
□ Key Points Covered in the Video:

1□ Introduction to Python:

- Python is a high-level, interpreted programming language.
- It was developed by **Guido van Rossum** in **1991**.

2□ Features of Python:

- **Easy to Learn & Readable:** Python syntax is simple and concise.
- **Extensive Libraries:** Python offers a vast collection of built-in libraries.
- **Cross-Platform Compatibility:** It runs on **Windows, Linux, and MacOS**.

3□ Applications of Python:

- **Web Development:** Using frameworks like Django and Flask.
- **Data Science & Machine Learning:** Libraries like Pandas, NumPy, and Scikit-Learn.
- **Automation & Scripting:** Used to automate repetitive tasks.

4□ Installing Python:

- Download and install Python from the official website (**python.org**).
- Set the **PATH variable** to access Python from the command line.

5□ First Python Program:

- Writing and running a simple `"Hello, World!"` program.
- Using Python Interpreter and script files (`.py`).

6□ Python Data Types:

- **Numeric Types:** `int`, `float`, `complex`.
- **Strings:** Used for text manipulation.
- **Lists:** Ordered and mutable collections.
- **Tuples:** Ordered but immutable collections.
- **Dictionaries:** Key-value pairs for data storage.

7□ Control Flow Statements:

- **Conditional Statements:** `if-else` statements for decision making.
- **Loops:** `for` and `while` loops for iteration.

8□ Functions in Python:

- Defining and calling functions.
- Built-in functions and user-defined functions.

9 Modules & Packages:

- Organizing code using modules (`import module_name`).
- Using third-party packages and libraries (`pip install package_name`).

Exception Handling:

- Handling errors using `try-except` blocks.

Python Basics - Notes

1. Python Variables

- **Variables** store data values.
- **Dynamic Typing**: No need to declare data type, Python detects it automatically.
- **Naming Rules**:
  - Must start with a letter (A-Z or a-z) or underscore (_).
  - Cannot start with a number (e.g., `1var` ✗
  - No special characters like `@, $, %`.

2. Python Data Types

Data Type	Description	Example
int	Integer numbers	`x = 10`
float	Decimal numbers	`y = 10.5`
str	String (text)	`name = "John"`
list	Ordered, mutable collection	`fruits = ["apple", "banana"]`
tuple	Ordered, immutable collection	`nums = (1, 2, 3)`
dict	Key-value pairs	`student = {"name": "Alex", "age": 20}`
set	Unordered, unique collection	`nums = {1, 2, 3, 4}`

3. Type Conversion

- **Implicit**: Python converts automatically.

- `a = 5 # int`
- `b = 2.5 # float`
- `c = a + b # float (Python converts int to float)`
- `print(c) # Output: 7.5`
- **Explicit**: Manually converting types.
- `x = "10"`
- `y = int(x) # Converts string "10" to integer 10`
- `print(y) # Output: 10`

🔹 *4. Operators in Python*

- **Arithmetic Operators:** `+ - * / % // **`
- **Comparison Operators:** `== != > < >= <=`
- **Logical Operators:** `and or not`
- **Assignment Operators:** `= += -= *= /=`
- **Bitwise Operators:** `& | ^ ~ << >>`

🔹 *5. Input & Output*

- Taking **user input**:
- `name = input("Enter your name: ")`
- `print("Hello, " + name)`
- Printing with **f-strings**:
- `age = 20`
- `print(f"I am {age} years old")`

🔹 *6. Comments in Python*

- **Single-line comment:** `# This is a comment`
- **Multi-line comment:**
- `'''`
- `This is a`
- `multi-line comment`
- `'''`

🔹 *7. Indentation in Python*

- Python uses **indentation** instead of `{}`.
- **Example:**
- `if 10 > 5:`
- `    print("10 is greater")  # Correct ✅`

🔹 *8. Conditional Statements*

```
age = 18
if age >= 18:
    print("You are an adult")
elif age >= 13:
    print("You are a teenager")
else:
    print("You are a child")
```

🔁 *9. Loops in Python*

- **For Loop:**
- `for i in range(5):`
- `    print(i)  # Output: 0 1 2 3 4`
- **While Loop:**
- `x = 0`
- `while x < 5:`
- `    print(x)`
- `    x += 1`

🔁 *10. Functions in Python*

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

print(greet("John"))  # Output: Hello, John!
```

🎯 Key Takeaways:

✅Python is easy to learn and use.
✅Variables store different types of data.
✅Indentation is important in Python.
✅Conditional statements (`if-else`) help in decision making.
✅Loops (`for`, `while`) help in iteration.
✅Functions make code reusable.

🚀 **These are the fundamental concepts you need to start coding in Python!** 🚀

🔁 Python Loops and Control Statements - Notes

🎯 1. Conditional Statements (if-else, elif)

✅*f Statement*

- Executes a block of code if a condition is **True**.
- Syntax:
- `age = 18`
- `if age >= 18:`
- `    print("You are an adult")`
- Output: `You are an adult`

✅*f-else Statement*

- Executes one block of code if the condition is **True**, and another if it is **False**.

- Syntax:
- `marks = 45`
- `if marks >= 40:`
- `    print("Pass")`
- `else:`
- `    print("Fail")`
- Output: `Pass`

✅ *if-elif-else Statement*

- Used when multiple conditions need to be checked.
- Syntax:
- `num = 0`
- `if num > 0:`
- `    print("Positive number")`
- `elif num < 0:`
- `    print("Negative number")`
- `else:`
- `    print("Zero")`
- Output: `Zero`

🔁 2. Loops in Python

✅ *for Loop*

- Used for iterating over a sequence (list, tuple, string, range, etc.).
- Syntax:
- `for i in range(5):`
- `    print(i)`
- Output:
- `0`
- `1`
- `2`
- `3`
- `4`

🔹 Looping through a list:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

- Output:
- `apple`
- `banana`
- `cherry`

✅ *while Loop*

- Repeats a block of code **as long as** the condition is true.
- Syntax:
- `x = 0`
- `while x < 5:`
- `    print(x)`
- `    x += 1`
- Output:
- `0`
- `1`
- `2`
- `3`
- `4`

🔁 3. Control Statements

✅ *break Statement*

- **Stops** the loop when a condition is met.
- Syntax:
- `for i in range(10):`
- `    if i == 5:`
- `        break`
- `    print(i)`
- Output:
- `0`
- `1`
- `2`
- `3`
- `4`

✅ *continue Statement*

- **Skips** the current iteration and moves to the next.
- Syntax:
- `for i in range(5):`
- `    if i == 2:`
- `        continue`
- `    print(i)`
- Output:
- `0`
- `1`
- `3`
- `4`

✅ *pass Statement*

- A placeholder when a statement is required but you **don't want to execute any code**.
- Syntax:
- `for i in range(5):`
- `    if i == 3:`
- `        pass`
- `    print(i)`

✅ Key Takeaways:

✅`if-else` helps make decisions in Python.
✅Loops (`for`, `while`) are used for repetition.
✅`break` stops the loop, `continue` skips an iteration, `pass` is a placeholder.

✅ **Practice writing loops and conditions to master Python programming!** ✅

📌 Python Operators - Notes

📌 *Video Title: Python Logical, Equality, Comparison, and Arithmetic Operators*

✅ **Video Link:** Watch Here

📌 1. Arithmetic Operators

Used for performing mathematical calculations.

Operator	Description	Example	Output
+	Addition	`5 + 3`	8
-	Subtraction	`10 - 4`	6
*	Multiplication	`6 * 7`	42
/	Division	`15 / 2`	7.5
//	Floor Division	`15 // 2`	7
%	Modulus (Remainder)	`10 % 3`	1

Operator	Description	Example	Output
`**`	Exponentiation	`2 ** 3`	`8`

---

2. Comparison Operators

Used to compare values and return `True` or `False`.

Operator	Description	Example	Output
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`10 > 5`	`True`
`<`	Less than	`2 < 7`	`True`
`>=`	Greater than or equal to	`8 >= 8`	`True`
`<=`	Less than or equal to	`6 <= 5`	`False`

---

3. Logical Operators

Used to combine multiple conditions.

Operator	Description	Example	Output
`and`	Returns `True` if both conditions are `True`	`(5 > 2) and (10 > 5)`	`True`
`or`	Returns `True` if at least one condition is `True`	`(5 > 10) or (10 > 5)`	`True`
`not`	Reverses the boolean result	`not (5 > 3)`	`False`

---

4. Identity Operators

Used to check if two variables refer to the same object in memory.

Operator	Description	Example	Output
`is`	Returns `True` if both variables refer to the same object	`a = [1,2,3]; b = a; print(a is b)`	`True`
`is not`	Returns `True` if variables refer to different objects	`a = [1,2,3]; b = [1,2,3]; print(a is not b)`	`True`

---

5. Membership Operators

Used to check whether a value exists in a sequence (like a list or string).

Operator	Description	Example	Output
`in`	Returns `True` if a value exists in a sequence	`'a' in 'apple'`	`True`
`not in`	Returns `True` if a value does not exist in a sequence	`'z' not in 'apple'`	`True`

Key Takeaways:

✅Arithmetic operators perform mathematical calculations.
✅Comparison operators check relationships between values.
✅Logical operators combine conditions.
✅Identity operators compare object references.
✅Membership operators check if a value exists in a sequence.

Practice in Python to strengthen your understanding!

I'll create structured notes for you based on the video. Give me a moment. ⏳

Python If-Else Statements - Notes

Video Title: Python If-Else Statements Explained

Video Link: Watch Here

I'll create structured notes for you based on the video. Give me a moment. ⏳

Python Loops - Notes

Video Title: Python Loops Explained

Video Link: Watch Here

---

1. Introduction to Loops

Loops are used in Python to execute a block of code multiple times. The two main types of loops in Python are:
✅**`for` loop** – Used for iterating over a sequence (list, tuple, string, range, etc.).
✅**`while` loop** – Repeats as long as a given condition is `True`.

---

2. `for` Loop

A `for` loop is used for iterating over a sequence like a list, tuple, string, or range.

✅Syntax:

```
for variable in sequence:
    # Code to execute in each iteration
```

✅Example:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

---

3. Using `range()` in `for` Loop

The `range()` function generates a sequence of numbers.

✅Example:

```
for i in range(5):  # Generates numbers from 0 to 4
    print(i)
```

Output:

```
0
1
2
3
4
```

✅Example with Custom Range:

```
for i in range(1, 10, 2):  # (start, stop, step)
    print(i)
```

Output:

```
1
3
5
7
9
```

🔁 4. `while` Loop

A `while` loop continues execution as long as a condition is `True`.

✅Syntax:

```
while condition:
    # Code to execute
```

✅Example:

```
x = 1
while x <= 5:
    print(x)
    x += 1
```

Output:

```
1
2
3
4
5
```

🔁 5. `break` Statement

The `break` statement is used to exit a loop when a condition is met.

✅Example:

```
for i in range(10):
    if i == 5:
        break  # Loop stops at 5
    print(i)
```

Output:

```
0
1
2
3
4
```

🔹 6. `continue` Statement

The `continue` statement skips the current iteration and moves to the next one.

✅Example:

```
for i in range(5):
    if i == 2:
        continue  # Skips 2
    print(i)
```

Output:

```
0
1
3
4
```

🔹 7. Nested Loops

A loop inside another loop is called a **nested loop**.

✅Example:

```
for i in range(3):
    for j in range(2):
        print(f"i = {i}, j = {j}")
```

Output:

```
i = 0, j = 0
i = 0, j = 1
i = 1, j = 0
i = 1, j = 1
i = 2, j = 0
```

```
i = 2, j = 1
```

⚠ Key Takeaways:

✅`for` loops are best for iterating over sequences.
✅`while` loops run as long as the condition remains `True`.
✅`break` stops the loop immediately.
✅`continue` skips the current iteration.
✅Nested loops allow complex iterations.

⚠ Practice writing loops in Python to master them! ⚠

⚠ 1. Introduction to Conditional Statements

Conditional statements in Python allow decision-making based on conditions. The `if`, `elif`, and `else` statements help control the flow of a program.

✅Basic Syntax of `if` Statement

```
if condition:
    # Code to execute if condition is True
```

✅Example:

```
age = 18
if age >= 18:
    print("You are eligible to vote.")
```

Output:

```
You are eligible to vote.
```

⚠ 2. `if-else` Statement

When a condition is `True`, one block of code executes; otherwise, the `else` block runs.

✅Syntax:

```
if condition:
    # Code executes if True
else:
```

```
    # Code executes if False
```

✅Example:

```
num = 10
if num % 2 == 0:
    print("Even Number")
else:
    print("Odd Number")
```

Output:

```
Even Number
```

🔹 3. `if-elif-else` Statement

The `elif` (else if) statement is used for multiple conditions.

✅Syntax:

```
if condition1:
    # Code executes if condition1 is True
elif condition2:
    # Code executes if condition2 is True
else:
    # Code executes if none of the above conditions are True
```

✅Example:

```
marks = 85

if marks >= 90:
    print("Grade: A")
elif marks >= 75:
    print("Grade: B")
elif marks >= 60:
    print("Grade: C")
else:
    print("Fail")
```

Output:

```
Grade: B
```

🔹 4. Nested `if` Statements

An `if` statement inside another `if` statement is called a nested `if`.

✅Example:

```
x = 10

if x > 5:
    print("x is greater than 5")
    if x > 8:
        print("x is also greater than 8")
```

Output:

```
x is greater than 5
x is also greater than 8
```

🔹 5. Using Logical Operators in `if` Statements

Logical operators (`and`, `or`, `not`) help combine conditions.

✅Example with `and`:

```
age = 25
income = 50000

if age > 18 and income > 30000:
    print("Eligible for Loan")
```

Output:

```
Eligible for Loan
```

✅Example with `or`:

```
age = 16
if age < 18 or age > 60:
    print("You are not eligible to work")
```

Output:

```
You are not eligible to work
```

🔹 Key Takeaways:

✅`if`, `elif`, and `else` are used for decision-making.
✅`if-else` handles two conditions, while `if-elif-else` handles multiple conditions.
✅Nested `if` statements allow checking multiple levels of conditions.
✅Logical operators help in complex conditions.

 Practice writing Python programs using `if-else` to understand better!

I'll create structured notes based on the video. Please wait a moment. ⏳

Python Functions - Notes

Video Title: Python Functions Explained

Video Link: Watch Here

---

1. What are Functions?

A **function** in Python is a reusable block of code that performs a specific task. Functions help in organizing code, improving readability, and avoiding repetition.

---

2. Defining a Function

A function is created using the `def` keyword.

✅Syntax:

```
def function_name(parameters):
    # Code to execute
    return value
```

✅Example:

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

print(greet("Alice"))
```

Output:

```
Hello, Alice!
```

---

3. Function with Multiple Parameters

You can pass multiple arguments to a function.

✅Example:

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

print(add_numbers(5, 10))
```

Output:

```
15
```

---

4. Default Parameters in Functions

If an argument is not provided, the default value is used.

✅Example:

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

print(greet())        # Uses default value
print(greet("Alice")) # Uses given argument
```

Output:

```
Hello, Guest!
Hello, Alice!
```

---

5. Returning Multiple Values

A function can return multiple values as a tuple.

✅Example:

```
def math_operations(a, b):
    return a + b, a - b, a * b, a / b

result = math_operations(10, 5)
print(result)  # (sum, difference, product, quotient)
```

Output:

```
(15, 5, 50, 2.0)
```

---

6. `*args` - Variable Length Arguments

The `*args` parameter allows a function to take any number of arguments.

✅Example:

```
def sum_all(*numbers):
    return sum(numbers)

print(sum_all(1, 2, 3, 4, 5))
```

Output:

```
15
```

---

7. `**kwargs` - Keyword Arguments

The `**kwargs` allows passing multiple key-value pairs.

✅Example:

```
def display_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

display_info(name="Alice", age=25, city="New York")
```

Output:

```
name: Alice
age: 25
city: New York
```

---

8. Lambda Functions (Anonymous Functions)

A **lambda function** is a small, one-line function without a name.

✅Example:

```
square = lambda x: x * x
print(square(5))
```

Output:

```
25
```

✅Example with Multiple Parameters:

```
multiply = lambda x, y: x * y
print(multiply(3, 4))
```

Output:

```
12
```

🔹 9. Recursive Functions

A function that calls itself is called a **recursive function**.

✅Example: Factorial Using Recursion

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))
```

Output:

```
120
```

🔹 Key Takeaways:

✅Functions help in code reusability and organization.
✅Use `def` to define a function.
✅Parameters allow passing values to functions.
✅`*args` handles multiple arguments, and `**kwargs` handles key-value pairs.
✅`lambda` functions are used for small, one-line functions.
✅Recursion allows a function to call itself.

🚀 Practice creating different types of functions in Python! 🚀

I'll create structured notes based on the video. Please wait a moment. ⏳

🚀 Python Classes and Objects - Notes

🔹 *Video Title: Python Classes and Objects Explained*

🚀 **Video Link:** Watch Here

---

1. What are Classes and Objects?

- **Class**: A blueprint for creating objects. It defines properties (attributes) and behaviors (methods).
- **Object**: An instance of a class with specific values assigned to attributes.

---

2. Defining a Class in Python

A class is created using the `class` keyword.

✅Example:

```
class Car:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year

    def display_info(self):
        return f"{self.year} {self.brand} {self.model}"
```

---

3. Creating Objects (Instances of a Class)

To create an object, we call the class with the required parameters.

✅Example:

```
car1 = Car("Toyota", "Corolla", 2022)
car2 = Car("Honda", "Civic", 2023)

print(car1.display_info())
print(car2.display_info())
```

Output:

```
2022 Toyota Corolla
2023 Honda Civic
```

---

4. The `__init__()` Method (Constructor)

- The `__init__` method is called automatically when an object is created.
- It initializes the attributes of the class.

✅Example:

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

student1 = Student("Alice", 20)
print(student1.name, student1.age)
```

Output:

```
Alice 20
```

---

5. Instance Variables vs Class Variables

- **Instance Variables:** Unique for each object (defined in `__init__`).
- **Class Variables:** Shared by all instances of a class.

✅Example:

```
class School:
    school_name = "XYZ High School"  # Class Variable

    def __init__(self, student_name):
        self.student_name = student_name  # Instance Variable

s1 = School("Alice")
s2 = School("Bob")

print(s1.school_name, s1.student_name)
print(s2.school_name, s2.student_name)
```

Output:

```
XYZ High School Alice
XYZ High School Bob
```

---

6. Adding Methods to a Class

Methods define the behavior of an object.

✅Example:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def get_salary(self):
        return f"{self.name}'s salary is {self.salary}"
```

```
emp1 = Employee("John", 50000)
print(emp1.get_salary())
```

Output:

```
John's salary is 50000
```

---

7. Using `self` Keyword

- `self` represents the instance of the class.
- It is used to access instance variables and methods.

✅Example:

```
class Dog:
    def __init__(self, name, breed):
        self.name = name
        self.breed = breed

    def bark(self):
        return f"{self.name} is barking!"

dog1 = Dog("Max", "Bulldog")
print(dog1.bark())
```

Output:

```
Max is barking!
```

---

8. Inheritance (Reusing a Class)

Inheritance allows one class to inherit properties and methods from another class.

✅Example:

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

    def speak(self):
        return "I make a sound"

class Dog(Animal):  # Inheriting from Animal
    def speak(self):
        return "Bark!"

dog = Dog("Buddy")
print(dog.name)
```

```
print(dog.speak())
```

Output:

```
Buddy
Bark!
```

---

9. Method Overriding

A child class can override a method from the parent class.

✅Example:

```
class Bird:
    def fly(self):
        return "Birds can fly"

class Penguin(Bird):
    def fly(self):
        return "Penguins cannot fly"

p = Penguin()
print(p.fly())
```

Output:

```
Penguins cannot fly
```

---

10. Encapsulation (Private and Public Variables)

Encapsulation restricts direct access to data and allows controlled modification.

- **Public variables:** Can be accessed anywhere.
- **Private variables:** Defined using __ (double underscore) and cannot be accessed directly.

✅Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private variable

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
print(account.get_balance())  # ✅Works
print(account.__balance)  # ❌Error (Cannot access private variable)
```

Output:

```
1000
AttributeError: 'BankAccount' object has no attribute '__balance'
```

🔹 11. Class Methods and Static Methods

- **Class Method (`@classmethod`)**: Works with class variables.
- **Static Method (`@staticmethod`)**: Independent of instance variables.

✅Example:

```
class MathOperations:
    @classmethod
    def class_method_example(cls):
        return "This is a class method"

    @staticmethod
    def static_method_example():
        return "This is a static method"

print(MathOperations.class_method_example())
print(MathOperations.static_method_example())
```

Output:

```
This is a class method
This is a static method
```

🔹 Key Takeaways:

✅Classes are blueprints for objects.
✅Objects are instances of classes.
✅Use `__init__` for initializing attributes.
✅Methods define the behavior of objects.
✅Inheritance allows reusing a class.
✅Encapsulation hides data using private variables.
✅Method overriding allows modifying inherited methods.
✅`@classmethod` and `@staticmethod` define specialized methods.

🚀 Practice creating your own classes and objects in Python! 🚀

I'll create structured notes based on the video. Please wait a moment. ⏳

🚀 Python OOP - Object-Oriented Programming Notes

Video Title: Python OOP Concepts Explained

Video Link: Watch Here

---

1. What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into reusable **objects** that contain **attributes** (data) and **methods** (functions).

Key Advantages of OOP:
✅Code reusability
✅Improved organization and structure
✅Easier debugging and maintenance
✅Better abstraction and modularity

---

2. Key Concepts of OOP

OOP is based on four fundamental principles:

1 Encapsulation

- Wrapping data (attributes) and methods inside a class.
- Prevents direct modification of data by making variables private.

2 Inheritance

- Allows one class to inherit methods and attributes from another class.
- Promotes code reuse and reduces redundancy.

3 Polymorphism

- Allows different classes to have methods with the same name but different implementations.

4 Abstraction

- Hides complex implementation details and only shows essential features.

🔹 3. Defining a Class in Python

A **class** is a blueprint for creating objects.

✅Example:

```
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model

    def show_details(self):
        return f"Car: {self.brand} {self.model}"
```

🔹 4. Creating Objects (Instances of a Class)

An **object** is an instance of a class.

✅Example:

```
car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Civic")

print(car1.show_details())
print(car2.show_details())
```

Output:

```
Car: Toyota Camry
Car: Honda Civic
```

🔹 5. Encapsulation (Data Hiding)

Encapsulation protects data from being modified directly.

✅Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private variable

    def get_balance(self):
        return self.__balance

account = BankAccount(5000)
print(account.get_balance())  # ✅Works
print(account.__balance)  # ❌Error (Cannot access private variable)
```

🔹 6. Inheritance (Reusing Code)

A child class can inherit attributes and methods from a parent class.

✅Example:

```
class Animal:
    def speak(self):
        return "Some sound"

class Dog(Animal):
    def speak(self):
        return "Bark!"

dog = Dog()
print(dog.speak())  # Output: Bark!
```

🔹 7. Polymorphism (Method Overriding)

Polymorphism allows different classes to use the same method name but with different behaviors.

✅Example:

```
class Bird:
    def fly(self):
        return "Flying high!"

class Penguin(Bird):
    def fly(self):
        return "Cannot fly!"

b = Bird()
p = Penguin()

print(b.fly())  # Output: Flying high!
print(p.fly())  # Output: Cannot fly!
```

🔹 8. Abstraction (Hiding Complexity)

Abstraction hides unnecessary details and provides a simple interface.

✅Example:

```
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
```

```
    def fuel_type(self):
        pass

class PetrolCar(Vehicle):
    def fuel_type(self):
        return "Petrol"

class ElectricCar(Vehicle):
    def fuel_type(self):
        return "Electric"

p = PetrolCar()
e = ElectricCar()

print(p.fuel_type())  # Output: Petrol
print(e.fuel_type())  # Output: Electric
```

✅ Key Takeaways:

✅OOP makes code more modular, reusable, and maintainable.
✅Classes define the structure, and objects are instances of a class.
✅Encapsulation hides data to prevent unauthorized access.
✅Inheritance allows code reuse between classes.
✅Polymorphism enables multiple classes to have methods with the same name but different behaviors.
✅Abstraction hides complex implementation details from users.

🚀 **Practice OOP concepts in Python to build scalable applications!** 🚀

I'll create structured notes for this video. Please wait a moment. ⏳

🐍 Python Functions - Notes

✅ *Video Title: Python Functions Explained*

🎥 **Video Link:** Watch Here

✅ 1. What are Functions in Python?

A **function** is a block of reusable code that performs a specific task. Functions help in:
✅Code Reusability
✅Better Organization
✅Easy Debugging
✅Modularity

❓ 2. Defining a Function

A function is defined using the `def` keyword.

✅Syntax:

```
def function_name(parameters):
    """Function Docstring"""
    # Function body
    return value
```

✅Example:

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

print(greet("John"))  # Output: Hello, John!
```

❓ 3. Types of Functions

1️⃣ Built-in Functions

These are predefined functions like `print()`, `len()`, `sum()`, etc.

2️⃣ User-Defined Functions

Functions created by users to perform specific tasks.

3️⃣ Lambda Functions

Anonymous functions that can be defined in one line using `lambda`.

✅Example:

```
square = lambda x: x * x
print(square(5))  # Output: 25
```

4️⃣ Recursive Functions

Functions that call themselves to solve problems.

✅Example:

```
def factorial(n):
    if n == 1:
        return 1
```

```
    return n * factorial(n - 1)

print(factorial(5))  # Output: 120
```

🔹 4. Function Arguments & Parameters

1️⃣ Positional Arguments

Arguments passed in the correct order.

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

print(add(2, 3))  # Output: 5
```

2️⃣ Default Arguments

Parameters with default values.

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

print(greet())  # Output: Hello, User!
print(greet("Alice"))  # Output: Hello, Alice!
```

3️⃣ Keyword Arguments

Arguments passed with parameter names.

```
def info(name, age):
    print(f"Name: {name}, Age: {age}")

info(age=25, name="John")  # Output: Name: John, Age: 25
```

4️⃣ Arbitrary Arguments (`*args`)

Allows passing multiple values.

```
def add_all(*numbers):
    return sum(numbers)

print(add_all(1, 2, 3, 4))  # Output: 10
```

5️⃣ Arbitrary Keyword Arguments (`**kwargs`)

Allows passing multiple key-value pairs.

```
def student_info(**details):
```

```
    for key, value in details.items():
        print(f"{key}: {value}")

student_info(name="Alice", age=22, course="CS")
```

---

5. Return vs Print

- `return` sends a value back and can be used later.
- `print` only displays the output.

✅Example:

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

result = add(4, 5)
print(result)  # Output: 9
```

Key Takeaways:

✅Functions improve code efficiency and readability.
✅Python supports different types of arguments.
✅`lambda` functions provide a concise way to define small functions.
✅Recursion helps solve problems like factorial and Fibonacci.

Practice functions in Python to master modular programming!

I'll create structured notes for this video. Please wait a moment. ⏳

Python Lists - Notes

Video Title: Python Lists Explained

Video Link: Watch Here

---

1. What is a List in Python?

A **list** is a built-in data structure in Python that allows you to store multiple values in a single variable. Lists are:
✅**Ordered** (Items have a specific position)
✅**Mutable** (Elements can be changed)
✅**Indexed** (Supports indexing and slicing)
✅**Heterogeneous** (Can store different data types)

✅Example:

```
# Creating a list
fruits = ["apple", "banana", "cherry"]
print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

---

2. Creating a List

Lists can contain numbers, strings, or even other lists.

✅Examples:

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# List of mixed data types
mixed = ["hello", 3.14, True]

# Nested list
nested_list = [[1, 2, 3], [4, 5, 6]]
```

---

3. Accessing List Elements

Lists support **indexing** and **slicing** to access elements.

✅Indexing:

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
print(fruits[-1]) # Output: cherry
```

✅Slicing:

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4])  # Output: [20, 30, 40]
print(numbers[:3])   # Output: [10, 20, 30]
print(numbers[::2])  # Output: [10, 30, 50]
```

---

4. Modifying a List

Lists are mutable, meaning elements can be updated, added, or removed.

✅Updating Elements:

```
fruits = ["apple", "banana", "cherry"]
fruits[1] = "mango"
```

```
print(fruits)  # Output: ['apple', 'mango', 'cherry']
```

✅Adding Elements:

```
# Using append()
fruits.append("orange")
print(fruits)  # Output: ['apple', 'mango', 'cherry', 'orange']

# Using insert()
fruits.insert(1, "grape")
print(fruits)  # Output: ['apple', 'grape', 'mango', 'cherry', 'orange']
```

✅Removing Elements:

```
# Using remove()
fruits.remove("mango")
print(fruits)  # Output: ['apple', 'grape', 'cherry', 'orange']

# Using pop()
fruits.pop(2)
print(fruits)  # Output: ['apple', 'grape', 'orange']

# Using del
del fruits[1]
print(fruits)  # Output: ['apple', 'orange']
```

---

5. List Operations

✅Concatenation & Repetition:

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

# Concatenation
result = list1 + list2
print(result)  # Output: [1, 2, 3, 4, 5, 6]

# Repetition
print(list1 * 2)  # Output: [1, 2, 3, 1, 2, 3]
```

✅Checking Membership:

```
numbers = [10, 20, 30, 40]
print(20 in numbers)   # Output: True
print(50 not in numbers)  # Output: True
```

---

6. List Methods

Python provides several built-in methods for list manipulation.

✅Common Methods:

```
numbers = [5, 2, 9, 1, 7]

numbers.sort()
print(numbers)  # Output: [1, 2, 5, 7, 9]

numbers.reverse()
print(numbers)  # Output: [9, 7, 5, 2, 1]

print(len(numbers))  # Output: 5
print(min(numbers))  # Output: 1
print(max(numbers))  # Output: 9

numbers.clear()
print(numbers)  # Output: []
```

❓ Key Takeaways:

✅Lists store multiple values in a single variable.
✅Lists support indexing, slicing, and iteration.
✅Lists are **mutable**, allowing modification of elements.
✅Built-in methods help in sorting, searching, and modifying lists.

 Practice list operations to master Python programming!

I'll create structured notes for this video. Please wait a moment. ⏳

Python Tuples - Notes

❓ *Video Title: Python Tuples Explained*

 Video Link: Watch Here

❓ 1. What is a Tuple in Python?

A **tuple** is a built-in data structure in Python that is:
✅**Ordered** (Items have a specific position)
✅**Immutable** (Elements cannot be changed after creation)
✅**Indexed** (Supports indexing and slicing)
✅**Heterogeneous** (Can store different data types)

✅Example:

```
# Creating a tuple
```

```
fruits = ("apple", "banana", "cherry")
print(fruits)  # Output: ('apple', 'banana', 'cherry')
```

---

2. Creating a Tuple

Tuples can contain numbers, strings, and even other tuples.

✅Examples:

```
# Tuple of numbers
numbers = (1, 2, 3, 4, 5)

# Tuple of mixed data types
mixed = ("hello", 3.14, True)

# Nested tuple
nested_tuple = ((1, 2, 3), (4, 5, 6))
```

Single-Element Tuple

A single-element tuple must have a **trailing comma**; otherwise, Python treats it as a normal variable.

```
single_element = ("hello",)  # Tuple
not_a_tuple = ("hello")  # String
print(type(single_element))  # Output: <class 'tuple'>
print(type(not_a_tuple))  # Output: <class 'str'>
```

---

3. Accessing Tuple Elements

Tuples support **indexing** and **slicing** like lists.

✅Indexing:

```
fruits = ("apple", "banana", "cherry")
print(fruits[0])  # Output: apple
print(fruits[-1]) # Output: cherry
```

✅Slicing:

```
numbers = (10, 20, 30, 40, 50)
print(numbers[1:4])  # Output: (20, 30, 40)
print(numbers[:3])   # Output: (10, 20, 30)
print(numbers[::2])  # Output: (10, 30, 50)
```

🔒 4. Immutability of Tuples

Tuples **cannot be modified** after creation.

```
fruits = ("apple", "banana", "cherry")
fruits[1] = "mango"  # ❌ TypeError: 'tuple' object does not support item assignment
```

If modification is necessary, convert the tuple into a list first:

```
fruits = ("apple", "banana", "cherry")
fruits_list = list(fruits)
fruits_list[1] = "mango"
fruits = tuple(fruits_list)
print(fruits)  # Output: ('apple', 'mango', 'cherry')
```

🔒 5. Tuple Operations

✅Concatenation & Repetition:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Concatenation
result = tuple1 + tuple2
print(result)  # Output: (1, 2, 3, 4, 5, 6)

# Repetition
print(tuple1 * 2)  # Output: (1, 2, 3, 1, 2, 3)
```

✅Checking Membership:

```
numbers = (10, 20, 30, 40)
print(20 in numbers)   # Output: True
print(50 not in numbers)  # Output: True
```

🔒 6. Tuple Methods

Tuples have fewer built-in methods than lists.

✅Common Methods:

```
numbers = (5, 2, 9, 1, 7)

print(len(numbers))  # Output: 5
print(min(numbers))  # Output: 1
print(max(numbers))  # Output: 9
```

```
# Count occurrences of an element
numbers = (1, 2, 3, 1, 1, 4)
print(numbers.count(1))  # Output: 3

# Find the index of an element
print(numbers.index(3))  # Output: 2
```

✅ Key Takeaways:

✅Tuples are **ordered and immutable**.
✅Tuples support **indexing and slicing**.
✅Tuples use **less memory** than lists, making them more efficient.
✅Use tuples when **data should not change**.

✅ **Practice tuple operations to improve Python skills!** ✅

I'll create structured notes for this video. Please wait a moment. ⏳

✅ Python Sets - Notes

✅ *Video Title: Python Sets Explained*

✅ **Video Link:** Watch Here

✅ 1. What is a Set in Python?

A **set** is a built-in data structure in Python that:
✅**Is Unordered** (Does not maintain insertion order)
✅**Contains Unique Elements** (No duplicate values)
✅**Is Mutable** (Can add or remove elements)
✅**Supports Mathematical Set Operations** (Union, Intersection, etc.)

✅Example:

```
# Creating a set
fruits = {"apple", "banana", "cherry"}
print(fruits)  # Output: {'banana', 'apple', 'cherry'}
```

✅ **Note:** The output order may vary because sets are unordered.

---

2. Creating a Set

Sets can contain numbers, strings, and even tuples, but **not lists or dictionaries** (since they are mutable).

✅Examples:

```
# Set of numbers
numbers = {1, 2, 3, 4, 5}

# Set of mixed data types
mixed = {10, "hello", 3.14}

# Set with duplicate elements (duplicates are removed automatically)
duplicates = {1, 2, 2, 3, 4, 4, 5}
print(duplicates)  # Output: {1, 2, 3, 4, 5}
```

❌Invalid Set Example (Using Mutable Elements)

```
invalid_set = {[1, 2, 3], "hello"}  # ❌ TypeError: unhashable type: 'list'
```

To create an **empty set**, use `set()` instead of `{}` (which creates an empty dictionary).

```
empty_set = set()
print(type(empty_set))  # Output: <class 'set'>
```

---

3. Accessing Set Elements

Sets **do not support indexing** because they are unordered.

```
fruits = {"apple", "banana", "cherry"}
print(fruits[0])  # ❌ TypeError: 'set' object is not subscriptable
```

To access elements, use a loop:

```
for fruit in fruits:
    print(fruit)
```

---

4. Modifying a Set

✅Adding Elements

Use `.add()` to insert a single item.

```
fruits = {"apple", "banana"}
fruits.add("cherry")
print(fruits)  # Output: {'banana', 'cherry', 'apple'}
```

Use `.update()` to add multiple elements.

```
fruits.update(["mango", "orange"])
print(fruits)  # Output: {'banana', 'cherry', 'apple', 'mango', 'orange'}
```

✅Removing Elements

```
fruits.remove("banana")  # ❌Raises error if item is missing
fruits.discard("banana")  # ✅No error if item is missing
print(fruits)
```

Use `.pop()` to remove a random element.

```
removed_item = fruits.pop()
print(removed_item)  # Output: Random element from set
```

Use `.clear()` to empty the set.

```
fruits.clear()
print(fruits)  # Output: set()
```

🧠 5. Set Operations

✅Union (Combine Two Sets - `|` or `.union()`)

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A | B)  # Output: {1, 2, 3, 4, 5}
print(A.union(B))  # Same output
```

✅Intersection (Common Elements - `&` or `.intersection()`)

```
print(A & B)  # Output: {3}
print(A.intersection(B))  # Same output
```

✅Difference (Elements in A but not in B - `-` or `.difference()`)

```
print(A - B)  # Output: {1, 2}
print(A.difference(B))  # Same output
```

✅Symmetric Difference (Elements in A or B, but not both - `^` or `.symmetric_difference()`)

```
print(A ^ B)  # Output: {1, 2, 4, 5}
print(A.symmetric_difference(B))  # Same output
```

---

6. Set Methods

✅Common Methods:

```
A = {1, 2, 3}
B = {2, 3}

print(A.issubset(B))    # False
print(B.issubset(A))    # True

print(A.issuperset(B)) # True
print(B.issuperset(A)) # False

A.add(5)
print(A)  # Output: {1, 2, 3, 5}

A.remove(3)
print(A)  # Output: {1, 2, 5}
```

Key Takeaways:

✅Sets are **unordered collections of unique elements**.
✅They **do not support indexing** but support **looping**.
✅Sets support **mathematical operations** like Union, Intersection, and Difference.
✅Use **sets when you need fast lookup and uniqueness of elements**.

Practice set operations to improve Python skills!

I'll create structured notes for this video. Please wait a moment. ⏳

Python Tuples - Notes

Video Title: Python Tuples Explained

Video Link: Watch Here

---

1. What is a Tuple in Python?

A **tuple** is a built-in data structure in Python that:
✅**Is Ordered** (Maintains the insertion order)
✅**Is Immutable** (Cannot be modified after creation)
✅**Can Store Different Data Types**
✅**Allows Duplicate Values**

✅Example:

```
# Creating a tuple
fruits = ("apple", "banana", "cherry")
print(fruits)  # Output: ('apple', 'banana', 'cherry')
```

---

2. Creating Tuples

✅Different Ways to Create Tuples:

```
# Tuple with multiple elements
numbers = (1, 2, 3, 4, 5)

# Tuple with different data types
mixed = (10, "hello", 3.14, True)

# Tuple with a single element (Needs a trailing comma)
single_element = (5,)
print(type(single_element))  # Output: <class 'tuple'>
```

 Note: Without a trailing comma, Python treats `(5)` as an integer.

```
not_a_tuple = (5)
print(type(not_a_tuple))  # Output: <class 'int'>
```

To create an **empty tuple**, use `()` or `tuple()`.

```
empty_tuple = ()
empty_tuple2 = tuple()
```

---

3. Accessing Tuple Elements

✅Using Indexing (0-based index)

```
fruits = ("apple", "banana", "cherry")
print(fruits[0])  # Output: apple
print(fruits[-1])  # Output: cherry (negative indexing)
```

✅Using Slicing

```
numbers = (10, 20, 30, 40, 50)
print(numbers[1:4])  # Output: (20, 30, 40)
print(numbers[:3])   # Output: (10, 20, 30)
print(numbers[::2])  # Output: (10, 30, 50)  # Every second element
```

---

4. Tuple Immutability

Tuples **cannot be modified** after creation.

```
numbers = (1, 2, 3)
numbers[0] = 10  # ❌ TypeError: 'tuple' object does not support item assignment
```

🔹 5. Tuple Operations

✅Concatenation (Merging Tuples)

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

merged = tuple1 + tuple2
print(merged)  # Output: (1, 2, 3, 4, 5, 6)
```

✅Repetition (Repeating Elements)

```
repeated = ("hello",) * 3
print(repeated)  # Output: ('hello', 'hello', 'hello')
```

✅Membership Test (`in` Operator)

```
fruits = ("apple", "banana", "cherry")
print("banana" in fruits)  # Output: True
```

🔹 6. Tuple Methods

✅Counting Elements

```
numbers = (1, 2, 3, 2, 2, 4, 5)
print(numbers.count(2))  # Output: 3 (number of times 2 appears)
```

✅Finding Index of an Element

```
print(numbers.index(3))  # Output: 2 (position of element 3)
```

🔹 7. Converting Tuple to List (If Modification is Needed)

Since tuples are **immutable**, convert them into a **list** to modify values.

```
fruits = ("apple", "banana", "cherry")
fruits_list = list(fruits)
fruits_list.append("orange")
fruits = tuple(fruits_list)
print(fruits)  # Output: ('apple', 'banana', 'cherry', 'orange')
```

🔑 Key Takeaways:

✅Tuples are **ordered and immutable**.
✅Use **indexing and slicing** to access elements.
✅Tuples **support concatenation, repetition, and membership testing**.
✅Convert a **tuple to a list** for modification.
✅Tuples are **faster and memory-efficient** compared to lists.

🚀 **Use tuples when data should remain constant!** 🚀

I'll create structured notes for this video. Please wait a moment. ⏳

🐍 Python Lists - Notes

🔑 *Video Title: Python Lists Explained*

🎥 **Video Link:** Watch Here

🔑 1. What is a List in Python?

A **list** is a built-in data structure in Python that:
✅**Is Ordered** (Maintains the insertion order)
✅**Is Mutable** (Can be modified after creation)
✅**Can Store Different Data Types**
✅**Allows Duplicate Values**

✅Example:

```
# Creating a list
fruits = ["apple", "banana", "cherry"]
print(fruits)  # Output: ['apple', 'banana', 'cherry']
```

🔑 2. Creating Lists

✅Different Ways to Create Lists:

```
# List with multiple elements
numbers = [1, 2, 3, 4, 5]

# List with different data types
mixed = [10, "hello", 3.14, True]

# Empty list
empty_list = []
```

```
empty_list2 = list()
```

---

3. Accessing List Elements

✅Using Indexing (0-based index)

```
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: apple
print(fruits[-1])  # Output: cherry (negative indexing)
```

✅Using Slicing

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4])  # Output: [20, 30, 40]
print(numbers[:3])   # Output: [10, 20, 30]
print(numbers[::2])  # Output: [10, 30, 50]  # Every second element
```

---

4. Modifying Lists

Since lists are **mutable**, we can change their values.

✅Changing Elements

```
numbers = [1, 2, 3]
numbers[0] = 10
print(numbers)  # Output: [10, 2, 3]
```

✅Adding Elements

```
# Append (Adds at the end)
fruits.append("orange")
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'orange']

# Insert (Adds at a specific index)
fruits.insert(1, "mango")
print(fruits)  # Output: ['apple', 'mango', 'banana', 'cherry', 'orange']
```

✅Removing Elements

```
# Remove specific element
fruits.remove("banana")
print(fruits)  # Output: ['apple', 'mango', 'cherry', 'orange']

# Pop (Removes last element by default)
fruits.pop()
print(fruits)  # Output: ['apple', 'mango', 'cherry']

# Pop at a specific index
fruits.pop(1)
```

```
print(fruits)  # Output: ['apple', 'cherry']
```

---

5. List Operations

✅Concatenation (Merging Lists)

```
list1 = [1, 2, 3]
list2 = [4, 5, 6]

merged = list1 + list2
print(merged)  # Output: [1, 2, 3, 4, 5, 6]
```

✅Repetition (Repeating Elements)

```
repeated = ["hello"] * 3
print(repeated)  # Output: ['hello', 'hello', 'hello']
```

✅Membership Test (`in` Operator)

```
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)  # Output: True
```

---

6. List Methods

✅Sorting a List

```
numbers = [5, 2, 8, 1, 9]
numbers.sort()
print(numbers)  # Output: [1, 2, 5, 8, 9]
```

✅Reversing a List

```
numbers.reverse()
print(numbers)  # Output: [9, 8, 5, 2, 1]
```

✅Finding the Index of an Element

```
print(fruits.index("cherry"))  # Output: 2
```

✅Counting Elements in a List

```
numbers = [1, 2, 3, 2, 2, 4, 5]
print(numbers.count(2))  # Output: 3
```

✅Copying a List

```
new_list = fruits.copy()
```

```
print(new_list)  # Output: ['apple', 'cherry']
```

✅Clearing a List

```
fruits.clear()
print(fruits)  # Output: []
```

🔑 Key Takeaways:

✅Lists are **ordered and mutable**.
✅Use **indexing and slicing** to access elements.
✅Lists **support concatenation, repetition, and membership testing**.
✅Lists have **built-in methods** for sorting, reversing, counting, etc.
✅Lists are **dynamic and can be modified easily**.

🚀 **Use lists when data needs to be modified frequently!** 🚀

I'll create structured notes for this video. Please wait a moment. ⏳

🐍 Python Tuples - Notes

🎥 *Video Title: Python Tuples Explained*

🔗 **Video Link:** Watch Here

🔹 1. What is a Tuple in Python?

A **tuple** is a built-in data structure in Python that:
✅**Is Ordered** (Maintains the insertion order)
✅**Is Immutable** (Cannot be modified after creation)
✅**Can Store Different Data Types**
✅**Allows Duplicate Values**

✅Example:

```
# Creating a tuple
fruits = ("apple", "banana", "cherry")
print(fruits)  # Output: ('apple', 'banana', 'cherry')
```

🔹 2. Creating Tuples

✅Different Ways to Create Tuples:

```
# Tuple with multiple elements
numbers = (1, 2, 3, 4, 5)

# Tuple with different data types
mixed = (10, "hello", 3.14, True)

# Empty tuple
empty_tuple = ()
empty_tuple2 = tuple()

# Tuple with one element (Comma is necessary)
single_element_tuple = (10,)
print(type(single_element_tuple))  # Output: <class 'tuple'>
```

---

3. Accessing Tuple Elements

✅Using Indexing (0-based index)

```
fruits = ("apple", "banana", "cherry")
print(fruits[0])  # Output: apple
print(fruits[-1])  # Output: cherry (negative indexing)
```

✅Using Slicing

```
numbers = (10, 20, 30, 40, 50)
print(numbers[1:4])  # Output: (20, 30, 40)
print(numbers[:3])   # Output: (10, 20, 30)
print(numbers[::2])  # Output: (10, 30, 50)  # Every second element
```

---

4. Why Use Tuples Instead of Lists?

Feature	Tuple	List
Mutability	Immutable ❌	Mutable ✅
Speed	Faster ✅	Slower ❌
Memory Usage	Less ✅	More ❌
Use Case	Fixed data ✅	Dynamic data ✅

Tuples are faster and take less memory compared to lists. Use them when the data should not be changed!

---

5. Tuple Operations

✅Concatenation (Merging Tuples)

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

merged = tuple1 + tuple2
print(merged)  # Output: (1, 2, 3, 4, 5, 6)
```

✅Repetition (Repeating Elements)

```
repeated = ("hello",) * 3
print(repeated)  # Output: ('hello', 'hello', 'hello')
```

✅Membership Test (`in` Operator)

```
fruits = ("apple", "banana", "cherry")
print("banana" in fruits)  # Output: True
```

---

6. Tuple Methods

✅Finding the Index of an Element

```
numbers = (10, 20, 30, 40)
print(numbers.index(30))  # Output: 2
```

✅Counting Elements in a Tuple

```
numbers = (1, 2, 3, 2, 2, 4, 5)
print(numbers.count(2))  # Output: 3
```

✅Length of a Tuple

```
print(len(numbers))  # Output: 7
```

✅Finding Maximum & Minimum

```
numbers = (5, 10, 20, 2, 8)
print(max(numbers))  # Output: 20
print(min(numbers))  # Output: 2
```

---

7. Tuple Packing & Unpacking

✅Packing a Tuple

```
student = ("John", 21, "CS")
```

✅Unpacking a Tuple

```
name, age, course = student
print(name)   # Output: John
print(age)    # Output: 21
print(course) # Output: CS
```

 Tuple unpacking is useful when returning multiple values from a function.

? Key Takeaways:

✅Tuples are **ordered and immutable**.
✅Use **indexing and slicing** to access elements.
✅Tuples **support concatenation, repetition, and membership testing**.
✅Tuples have **built-in methods** for counting, indexing, and length checking.
✅Tuples are **faster and memory-efficient** compared to lists.

 Use tuples when data should remain constant!

Python MySQL Quick Guide

1. **Install MySQL Connector:**

```
python -m pip install mysql-connector-python
```

2. **Test Installation:**

```
import mysql.connector
```

3. **Create Database Connection:**

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
     user="yourusername",
     password="yourpassword"
)

print(mydb)
```

Now, you can execute SQL queries using Python!

Python MySQL: Create & Check Database

1. **Create a Database:**

```
import mysql.connector

mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword"
)

mycursor = mydb.cursor()
mycursor.execute("CREATE DATABASE mydatabase")
```

2. **Check if Database Exists:**

```
mycursor.execute("SHOW DATABASES")
for x in mycursor:
    print(x)
```

3. **Connect to a Database:**

```
mydb = mysql.connector.connect(
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)
```

If the database doesn't exist, an error will occur.

Python MySQL: Create & Check Table

1. **Create a Table:**

```
import mysql.connector

mydb = mysql.connector.connect(
```

```
    host="localhost",
    user="yourusername",
    password="yourpassword",
    database="mydatabase"
)

mycursor = mydb.cursor()
mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address
VARCHAR(255))")
```

13. **Check if Table Exists:**

```
mycursor.execute("SHOW TABLES")
for x in mycursor:
    print(x)
```

Successfully creates and verifies the table! □

Python MySQL: Insert Data

1. **Insert Single Record:**

```
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)
mydb.commit()
print(mycursor.rowcount, "record inserted.")
```

7. **Insert Multiple Records:**

```
val = [('Peter', 'Lowstreet 4'), ('Amy', 'Apple st 652'), ('Hannah',
'Mountain 21')]
mycursor.executemany(sql, val)
mydb.commit()
print(mycursor.rowcount, "records inserted.")
```

12. **Get Inserted ID:**

```
print("Inserted ID:", mycursor.lastrowid)
```

Efficiently inserts data into MySQL! □

Python MySQL: Select Data

1. **Select All Records:**

```
mycursor.execute("SELECT * FROM customers")
myresult = mycursor.fetchall()
for x in myresult:
    print(x)
```

6. **Select Specific Columns:**

```
mycursor.execute("SELECT name, address FROM customers")
myresult = mycursor.fetchall()
```

9. **Fetch One Record:**

```
myresult = mycursor.fetchone()
print(myresult)
```

Quickly retrieve data from MySQL! □

Python MySQL: WHERE Clause

1. **Filter Records:**
```
mycursor.execute("SELECT * FROM customers WHERE address = 'Park Lane 38'")
```
3. **Use Wildcards (%):**
```
mycursor.execute("SELECT * FROM customers WHERE address LIKE '%way%'")
```
5. **Prevent SQL Injection:**
```
sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2",)
mycursor.execute(sql, adr)
```

Filter MySQL data efficiently! □

Python MySQL: ORDER BY

1. **Sort Ascending (Default):**
```
mycursor.execute("SELECT * FROM customers ORDER BY name")
```
3. **Sort Descending:**
```
mycursor.execute("SELECT * FROM customers ORDER BY name DESC")
```

Sort MySQL query results easily! □

Python MySQL: DELETE FROM

1. **Delete a specific record:**
```
mycursor.execute("DELETE FROM customers WHERE address = 'Mountain 21'")
mydb.commit()
```
4. **Prevent SQL Injection:**
```
sql = "DELETE FROM customers WHERE address = %s"
adr = ("Yellow Garden 2",)
mycursor.execute(sql, adr)
mydb.commit()
```

Always use `mydb.commit()` to apply changes! ⚡

Python MySQL: DROP TABLE

1. **Delete a table:**
```
mycursor.execute("DROP TABLE customers")
```
3. **Drop only if it exists:**
```
mycursor.execute("DROP TABLE IF EXISTS customers")
```

Avoid errors by using `IF EXISTS`! □

Python MySQL: UPDATE TABLE

1. **Update a record:**

```
mycursor.execute("UPDATE customers SET address = 'Canyon 123' WHERE
address = 'Valley 345'")
mydb.commit()
```

4. **Prevent SQL Injection:**

```
sql = "UPDATE customers SET address = %s WHERE address = %s"
val = ("Canyon 123", "Valley 345")
mycursor.execute(sql, val)
mydb.commit()
```

Use `WHERE` to avoid updating all records! □

Python MySQL: LIMIT

1. **Limit records:**

```
mycursor.execute("SELECT * FROM customers LIMIT 5")
```

3. **Start from a specific position:**

```
mycursor.execute("SELECT * FROM customers LIMIT 5 OFFSET 2")
```

Use `LIMIT` to control results and `OFFSET` to skip records! □

Python MySQL: JOIN

Use `JOIN` to combine rows from multiple tables based on a common column.

Example:

```
mycursor.execute("SELECT users.name, products.name FROM users JOIN products
ON users.fav = products.id")
```

Efficiently merge related data! □

Machine Learning Overview
Machine Learning enables computers to learn from data and predict outcomes, advancing AI.

Getting Started
This tutorial covers statistics, Python modules, and predictive functions.

Data Set
A data set is any collection of data, from arrays to databases.

Examples:

- **Array:** [99,86,87,88,111,86,103,87,94,78,77,85,86]
- **Database:** Car details (Name, Color, Age, Speed)

Mean, Median, Mode in Machine Learning

- **Mean**: The average value.
- `import numpy`
- `speed = [99,86,87,88,111,86,103,87,94,78,77,85,86]`
- `print(numpy.mean(speed))  # Output: 89.77`
- **Median**: The middle value after sorting.
- `print(numpy.median(speed))  # Output: 87`
- **Mode**: The most frequent value.
- `from scipy import stats`
- `print(stats.mode(speed))  # Output: 86`

These concepts are essential in Machine Learning.

Standard Deviation in Machine Learning

Standard deviation measures how spread out values are:

- **Low SD** → Values close to the mean.
- **High SD** → Values widely spread.

Example:

```
import numpy
speed1 = [86,87,88,86,87,85,86]
print(numpy.std(speed1))  # Output: 0.9

speed2 = [32,111,138,28,59,77,97]
print(numpy.std(speed2))  # Output: 37.85
```

A higher SD means greater data variability.

Percentiles in Machine Learning

Percentiles indicate the value below which a given percentage of data falls.

Example:

```
import numpy
ages = [5,31,43,48,50,41,7,11,15,39,80,82,32,2,8,6,25,36,27,61,31]

print(numpy.percentile(ages, 75))  # Output: 43 (75% are ≤ 43)
print(numpy.percentile(ages, 90))  # Output: Age below which 90% fall
```

Useful for data analysis!

Machine Learning - Data Distribution

Large data sets are essential for ML. NumPy helps generate random data, and Matplotlib visualizes it using histograms.

Example:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.random.uniform(0, 5, 100000)
plt.hist(x, 100)
plt.show()
```

Histograms help analyze data distribution efficiently!

Machine Learning - Normal Data Distribution

Normal distribution (Gaussian distribution) has values concentrated around a mean.

Example:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.random.normal(5.0, 1.0, 100000)
plt.hist(x, 100)
plt.show()
```

Most values are near 5.0, forming a bell curve!

Machine Learning - Scatter Plot

A scatter plot represents data points as dots on a graph.

Example:

```
import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

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

It shows car ages vs. speeds.

Random Data Example:

```
import numpy as np

x = np.random.normal(5.0, 1.0, 1000)
y = np.random.normal(10.0, 2.0, 1000)

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

Dots cluster around (5,10) with a wider spread on the y-axis.

Linear Regression in Machine Learning

Regression predicts relationships between variables to forecast future outcomes.

Linear Regression fits a straight line through data points to make predictions.

How It Works

Python provides methods to compute linear regression without manual calculations.

1. **Scatter Plot**

```
import matplotlib.pyplot as plt
x = [5,7,8,7,2,17,2,9,4,11,12,9,6]
y = [99,86,87,88,111,86,103,87,94,78,77,85,86]
plt.scatter(x, y)
plt.show()
```

7. **Linear Regression Line**

```
from scipy import stats
slope, intercept, r, p, std_err = stats.linregress(x, y)
 def myfunc(x): return slope * x + intercept
 mymodel = list(map(myfunc, x))
 plt.scatter(x, y)
 plt.plot(x, mymodel)
 plt.show()
```

R-Value (Relationship Measure)

- **r = -1 to 1** (Closer to ±1 → Stronger relationship).
- Example:

- `print(r)  # Output: -0.76 (usable relationship)`

Prediction

Predicting speed of a 10-year-old car:

```
speed = myfunc(10)
print(speed)  # Output: 85.6
```

Bad Fit Example

Some data sets are not suitable for linear regression if r is close to 0.

```
print(r)  # Output: 0.013 (poor fit)
```

Polynomial Regression - Summary

What is Polynomial Regression?
Polynomial regression is used when data points do not fit a straight line. It models the relationship between variables using a polynomial equation.

How It Works:

1. **Scatter Plot:** Visualize data distribution.
2. **Fit a Polynomial Curve:** Use `numpy.polyfit()` to create a polynomial model.
3. **Draw Regression Line:** Use `numpy.linspace()` for smooth plotting.
4. **Evaluate Fit:** Compute R^2 value (`r2_score()`) to check accuracy (0 = no relation, 1 = perfect fit).
5. **Make Predictions:** Use the model to predict future values.

Good vs. Bad Fit:

- A high R^2 value (e.g., 0.94) means a strong relationship.
- A low R^2 value (e.g., 0.00995) indicates poor fit, making predictions unreliable.

Polynomial Regression - Summary

What is Polynomial Regression?
Polynomial regression is used when data points do not fit a straight line. It models the relationship between variables using a polynomial equation.

How It Works:

1. **Scatter Plot:** Visualize data distribution.
2. **Fit a Polynomial Curve:** Use `numpy.polyfit()` to create a polynomial model.
3. **Draw Regression Line:** Use `numpy.linspace()` for smooth plotting.

4. **Evaluate Fit:** Compute R^2 value (`r2_score()`) to check accuracy (0 = no relation, 1 = perfect fit).
5. **Make Predictions:** Use the model to predict future values.

Good vs. Bad Fit:

- A high R^2 value (e.g., 0.94) means a strong relationship.
- A low R^2 value (e.g., 0.00995) indicates poor fit, making predictions unreliable.

Machine Learning - Feature Scaling

When data has different units (e.g., kg vs. meters), comparing values becomes difficult. **Scaling** helps by converting data into comparable values using **standardization**:

z=x−usz = \frac{x - u}{s}

where uu is the mean and ss is the standard deviation.

Example:

For **Weight = 790** and **Volume = 1.0**, the scaled values are **-2.1** and **-1.59**.

Using Python (StandardScaler from sklearn):

```
import pandas
from sklearn.preprocessing import StandardScaler

df = pandas.read_csv("data.csv")
scale = StandardScaler()
X = df[['Weight', 'Volume']]
scaledX = scale.fit_transform(X)
print(scaledX)
```

To **predict CO2 emission** for a **1.3L, 2300kg** car:

```
regr.fit(scaledX, df['CO2'])
scaled = scale.transform([[2300, 1.3]])
predictedCO2 = regr.predict([scaled[0]])
print(predictedCO2)
```

Output: `[107.2]`

Machine Learning - Train/Test & Model Evaluation

Train/Test is a method to measure model accuracy by splitting data into training (80%) and testing (20%) sets.

Steps:

1. **Prepare Data:** Generate a dataset of 100 customers with shopping habits.
2. **Split Data:** 80% for training, 20% for testing.
3. **Visualize Data:** Plot training and testing sets to ensure fairness.
4. **Fit Model:** Use polynomial regression to fit data.
5. **Evaluate Model:** Use R^2 score (0 to 1) to measure accuracy.
6. **Test on New Data:** Predict spending based on time in the shop.

Result: A reliable model can predict customer spending based on time spent in-store.

Machine Learning - Decision Tree (Short Version)

A **Decision Tree** is a flowchart-based model that helps make decisions using past data.

Example Scenario

A person decides whether to attend a comedy show based on:

- **Age**, **Experience**, **Rank**, and **Nationality** of the comedian.

Building the Decision Tree in Python

1. **Read Data:**

```
import pandas as pd
df = pd.read_csv("data.csv")
```

4. **Convert categorical data to numerical:**

```
df['Nationality'] = df['Nationality'].map({'UK': 0, 'USA': 1, 'N': 2})
df['Go'] = df['Go'].map({'YES': 1, 'NO': 0})
```

7. **Prepare features and target:**

```
X = df[['Age', 'Experience', 'Rank', 'Nationality']]
y = df['Go']
```

10. **Create & Train Decision Tree:**

```
from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier().fit(X, y)
```

13. **Predict Example:**

```
print(model.predict([[40, 10, 7, 1]]))  # Predict decision
```

Understanding Gini Index

- **Gini = 1 - (p^2 + q^2)** → Measures data purity at each split.
- **Lower Gini = Better split quality.**

Decision Tree Use

- Predicts whether to attend a comedy show based on learned patterns.
- Results may vary slightly depending on training data.

Confusion Matrix in Machine Learning

A **Confusion Matrix** is a table used to evaluate classification models by comparing actual vs. predicted values.

Key Components:

- **True Negative (TN)** – Correctly predicted negatives
- **False Positive (FP)** – Incorrectly predicted positives
- **False Negative (FN)** – Incorrectly predicted negatives
- **True Positive (TP)** – Correctly predicted positives

Metrics:

- **Accuracy** = (TP + TN) / Total Predictions
- **Precision** = TP / (TP + FP)
- **Recall (Sensitivity)** = TP / (TP + FN)
- **Specificity** = TN / (TN + FP)
- **F1-Score** = 2 * (Precision * Recall) / (Precision + Recall)

Python Code:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import metrics

actual = np.random.binomial(1, 0.9, 1000)
predicted = np.random.binomial(1, 0.9, 1000)

conf_matrix = metrics.confusion_matrix(actual, predicted)
metrics.ConfusionMatrixDisplay(conf_matrix, display_labels=[0,1]).plot()
plt.show()

print({
    "Accuracy": metrics.accuracy_score(actual, predicted),
    "Precision": metrics.precision_score(actual, predicted),
    "Recall": metrics.recall_score(actual, predicted),
    "Specificity": metrics.recall_score(actual, predicted, pos_label=0),
    "F1-Score": metrics.f1_score(actual, predicted)
})
```

Machine Learning - Confusion Matrix

A **Confusion Matrix** is a table used in classification problems to analyze model errors.

Creating a Confusion Matrix

- Use NumPy to generate actual & predicted values:
- `import numpy`
- `actual = numpy.random.binomial(1, 0.9, size=1000)`
- `predicted = numpy.random.binomial(1, 0.9, size=1000)`

- Compute confusion matrix using `sklearn.metrics`:
- `from sklearn import metrics`
- `confusion_matrix = metrics.confusion_matrix(actual, predicted)`
- Visualize with Matplotlib:
- `import matplotlib.pyplot as plt`
- `cm_display = metrics.ConfusionMatrixDisplay(confusion_matrix, display_labels=[0,1])`
- `cm_display.plot()`
- `plt.show()`

Confusion Matrix Components

- **True Negative (TN)** – Correctly predicted negatives
- **False Positive (FP)** – Incorrectly predicted positives
- **False Negative (FN)** – Incorrectly predicted negatives
- **True Positive (TP)** – Correctly predicted positives

Evaluation Metrics

- **Accuracy**: (TP + TN) / Total Predictions
- `Accuracy = metrics.accuracy_score(actual, predicted)`
- **Precision**: TP / (TP + FP)
- `Precision = metrics.precision_score(actual, predicted)`
- **Recall (Sensitivity)**: TP / (TP + FN)
- `Sensitivity = metrics.recall_score(actual, predicted)`
- **Specificity**: TN / (TN + FP)
- `Specificity = metrics.recall_score(actual, predicted, pos_label=0)`
- **F1-Score**: Harmonic mean of Precision & Recall
- `F1_score = metrics.f1_score(actual, predicted)`

All Metrics in One Print Statement

```
print({"Accuracy": Accuracy, "Precision": Precision, "Recall": Sensitivity, "Specificity": Specificity, "F1_score": F1_score})
```

Hierarchical clustering is an unsupervised learning method that groups data points based on their similarities. Using Agglomerative Clustering, it follows a bottom-up approach, merging the closest clusters iteratively until all points form a single cluster.

Key steps:

1. **Data Visualization:** Plot initial points.
2. **Dendrogram:** Compute linkage using Euclidean distance and Ward's method, then visualize hierarchy.
3. **Clustering with Scikit-Learn:** Use `AgglomerativeClustering` to classify points into clusters and plot results.

This method helps in understanding relationships between data points without prior labels.

Logistic Regression Overview

Logistic regression is used for classification problems, predicting categorical outcomes (e.g., whether a tumor is cancerous or not).

How It Works:

- Independent variable (X) represents tumor size.
- Dependent variable (y) indicates cancer presence (0 = No, 1 = Yes).
- Uses `LogisticRegression()` from `sklearn` to train the model and predict outcomes.

Example:

```
import numpy
from sklearn import linear_model

X = numpy.array([3.78, 2.44, 2.09, 0.14, 1.72, 1.65, 4.92, 4.37, 4.96, 4.52,
3.69, 5.88]).reshape(-1,1)
y = numpy.array([0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1])

logr = linear_model.LogisticRegression()
logr.fit(X, y)

predicted = logr.predict(numpy.array([3.46]).reshape(-1,1))
print(predicted)  # Output: [0] (Not cancerous)
```

Key Insights:

- **Coefficient:** A 1mm increase in tumor size increases the odds of being cancerous by ~4x.
- **Probability Calculation:**
- `def logit2prob(logr, X):`
- `    log_odds = logr.coef_ * X + logr.intercept_`
- `    odds = numpy.exp(log_odds)`
- `    return odds / (1 + odds)`
- `print(logit2prob(logr, X))`
    - Example: A tumor of 3.78cm has a **61% chance** of being cancerous.

*Logi***Preprocessing Categorical Data**
Machine learning models require numeric data, so categorical data (e.g., car brands) must be transformed.

One-Hot Encoding (OHE)

- Converts categories into binary columns (1 for presence, 0 for absence).
- Done easily using `pd.get_dummies()`.

Example:

```
import pandas as pd

cars = pd.read_csv('data.csv')
ohe_cars = pd.get_dummies(cars[['Car']])
print(ohe_cars.to_string())
```

This converts the "Car" column into multiple binary columns for each brand.

K-Means Clustering (ML)

K-means is an **unsupervised learning** method that groups data into **K clusters** by minimizing variance.

Steps:

1. **Random Assignment** – Data points are randomly assigned to K clusters.
2. **Centroid Calculation** – Compute the cluster centers.
3. **Reassignment** – Assign points to the nearest centroid.
4. **Repeat** – Until cluster assignments remain unchanged.

Finding Best K (Elbow Method)

- **Inertia** (sum of squared distances) is plotted for different K values.
- The "elbow" in the graph suggests the best K.

Python Example:

```
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans

x = [4, 5, 10, 4, 3, 11, 14, 6, 10, 12]
y = [21, 19, 24, 17, 16, 25, 24, 22, 21, 21]
data = list(zip(x, y))

# Elbow Method
inertias = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i)
    kmeans.fit(data)
    inertias.append(kmeans.inertia_)

plt.plot(range(1, 11), inertias, marker='o')
plt.xlabel('Clusters')
plt.ylabel('Inertia')
plt.show()
```

`# K-Means Clustering (Best K=2)` **Bootstrap Aggregation (Bagging) in Machine Learning**

Bagging is an ensemble method that reduces overfitting and improves accuracy in classification and regression tasks. It works by training multiple models on random subsets of the dataset (with replacement) and aggregating their predictions (majority vote for classification, averaging for regression).

Base Classifier Performance

Using a **Decision Tree Classifier** on the **Wine dataset**, we get:

- **Train Accuracy:** 100%
- **Test Accuracy:** 82.2% (Overfitting observed)

Bagging Classifier Implementation

A **Bagging Classifier** with multiple estimators is applied, showing improved accuracy.

- **Best Accuracy: 95.5%** with **12 estimators**
- **Out-of-Bag (OOB) Score:** Used for additional model evaluation

Bagging helps create more robust models by reducing variance and improving generalization.

```
kmeans = KMeans(n_clusters=2)
kmeans.fit(data)
plt.scatter(x, y, c=kmeans.labels_)
plt.show()
```

This clusters the data into **2 groups** based on the elbow method. □

stic regression helps predict classification outcomes with probability estimates.

Grid Search in Machine Learning

Grid Search helps find the best model parameters by testing different values. In logistic regression, the **C** parameter controls regularization. Higher **C** prioritizes training data, while lower **C** prevents overfitting.

Steps:

1. Train a logistic regression model on the **Iris dataset** with default **C = 1** (accuracy: 0.973).
2. Implement Grid Search with different **C** values (e.g., 0.25 to 2).
3. Store and compare scores to find the best value.
4. Higher **C** improved accuracy but beyond **1.75** had no benefit.

⚠Avoid Overfitting: Always use train/test split for better generalization.

Bootstrap Aggregation (Bagging) in Machine Learning

Bagging is an ensemble method that reduces overfitting and improves accuracy in classification and regression tasks. It works by training multiple models on random subsets of the dataset (with replacement) and aggregating their predictions (majority vote for classification, averaging for regression).

Base Classifier Performance

Using a **Decision Tree Classifier** on the **Wine dataset**, we get:

- **Train Accuracy:** 100%
- **Test Accuracy:** 82.2% (Overfitting observed)

Bagging Classifier Implementation

A **Bagging Classifier** with multiple estimators is applied, showing improved accuracy.

- **Best Accuracy: 95.5%** with **12 estimators**
- **Out-of-Bag (OOB) Score:** Used for additional model evaluation

Bagging helps create more robust models by reducing variance and improving generalization.

AUC - ROC Curve in Machine Learning

The **AUC - ROC Curve** evaluates classification models by plotting the **True Positive Rate (TPR) vs. False Positive Rate (FPR)** at different thresholds. **AUC (Area Under the Curve)** measures how well a model separates classes (closer to 1 is better).

Key Insights:

- **Accuracy** alone is unreliable for imbalanced data.
- **AUC is better** as it considers probability distribution.
- **Example:** A perfect model has **AUC ≈ 1**, while a random guess has **AUC ≈ 0.5**.
- **More confident models** (probabilities closer to 0 or 1) have **higher AUC** and perform better in real-world predictions.

K-Nearest Neighbors (KNN) is a simple supervised ML algorithm used for classification, regression, and missing value imputation. It classifies data points based on the majority class of their nearest neighbors, with K determining the number of neighbors considered.

Implementation:

1. **Data Visualization:**
   - Plotted sample data points using `matplotlib.pyplot`.
2. **KNN Model (K=1):**
   - Fitted `KNeighborsClassifier(n_neighbors=1)` on the dataset.
   - Classified a new point (8,21) → Result: Class 0.

3. **KNN Model (K=5):**
    - Increased neighbors to 5.
    - Reclassified the same point → Result: Class 1.

Larger K values provide more stable decision boundaries and reduce outliers' impact.

Python and MongoDB

Python can be utilized for database applications, with MongoDB being one of the most well-known NoSQL databases.

About MongoDB

MongoDB organizes data in JSON-like documents, providing a highly adaptable and scalable database structure.

To experiment with the examples in this guide, you will need access to a MongoDB database.

- You can obtain a free version of MongoDB from MongoDB's official site.
- Alternatively, you can start immediately using MongoDB Atlas, a cloud-based service.

Using PyMongo

To interact with MongoDB in Python, you need a database driver. The recommended driver for MongoDB in Python is **PyMongo**.

It is advisable to install PyMongo via PIP, which is likely pre-installed in your Python setup.

To install PyMongo, open the command line, navigate to the PIP directory, and execute the following command:

```
sh
CopyEdit
python -m pip install pymongo
```

Once the installation is complete, you have successfully installed the MongoDB driver.

Verifying PyMongo Installation

To check whether PyMongo is installed correctly, create a Python script containing the following line:

demo_mongodb_test.py

```
python
CopyEdit
import pymongo
```

If this script runs without errors, PyMongo has been installed and is ready for use.

Python MongoDB: Creating a Database

How to Create a Database

To establish a database in MongoDB using Python, first instantiate a `MongoClient` object, then specify the connection URL, including the appropriate IP address and the database name.

If the specified database does not already exist, MongoDB will automatically generate it upon establishing the connection.

Example: Creating a Database

Below is an example demonstrating how to create a database named **"mydatabase"**:

```
import pymongo

# Establish a connection to MongoDB
myclient = pymongo.MongoClient("mongodb://localhost:27017/")

# Define the database
mydb = myclient["mydatabase"]
```

Important Note:

In MongoDB, a database is **not** actually created until it contains data. This means you must first create a **collection** (similar to a table) and insert at least one **document** (record) before the database is officially created.

Checking if a Database Exists

Since databases in MongoDB do not materialize until they contain data, if you are creating a database for the first time, you should first learn how to **create a collection and insert a document** before verifying its existence.

Example: Listing All Databases

To view a list of all available databases in the system, use:

```
print(myclient.list_database_names())
```

Example: Checking for a Specific Database

To verify if a particular database (e.g., **"mydatabase"**) exists, use:

```
dblist = myclient.list_database_names()
if "mydatabase" in dblist:
    print("The database exists.")
```

Now, you're all set to work with MongoDB databases in Python!

Python MongoDB: Creating a Collection

Understanding Collections

In MongoDB, a **collection** functions similarly to a **table** in relational databases.

How to Create a Collection

To define a collection in MongoDB using Python, reference the **database object** and specify the desired collection name. If the collection does not already exist, MongoDB will generate it automatically when data is added.

Example: Creating a Collection

The following script sets up a **"customers"** collection within the **"mydatabase"** database:

```
import pymongo

# Establish a connection to MongoDB
myclient = pymongo.MongoClient("mongodb://localhost:27017/")

# Select the database
mydb = myclient["mydatabase"]

# Define the collection
mycol = mydb["customers"]
```

Important Note:

A collection **does not actually exist** until at least one document (record) has been inserted into it. MongoDB only creates the collection when it contains data.

Checking if a Collection Exists

Since collections are not officially created until they hold data, if this is your first time working with a collection, ensure you have inserted a document before verifying its existence.

Example: Listing All Collections

To retrieve a list of all available collections within a database, use:

```
print(mydb.list_collection_names())
```

Example: Checking for a Specific Collection

To determine whether a particular collection (e.g., **"customers"**) exists, execute:

```
collist = mydb.list_collection_names()
if "customers" in collist:
    print("The collection exists.")
```

With this setup, you can now effectively manage collections in MongoDB using Python!

Python MongoDB: Inserting Documents

Understanding Documents in MongoDB

In MongoDB, a **document** is equivalent to a **record** in relational databases like SQL.

Inserting a Single Document

To add a document to a collection, use the **`insert_one()`** method.
This method takes a dictionary as an argument, representing the fields and values of the document to be inserted.

Example: Inserting One Document

The following script inserts a record into the **"customers"** collection:

```
import pymongo

# Establish connection
myclient = pymongo.MongoClient("mongodb://localhost:27017/")

# Select database and collection
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

# Define document
mydict = { "name": "John", "address": "Highway 37" }

# Insert document
x = mycol.insert_one(mydict)
```

Retrieving the Inserted Document's ID

The **`insert_one()`** method returns an **InsertOneResult** object, which contains an **`inserted_id`** attribute holding the ID of the newly added document.

Example: Get the Inserted Document's ID

```
mydict = { "name": "Peter", "address": "Lowstreet 27" }

x = mycol.insert_one(mydict)

print(x.inserted_id)  # Prints the unique ID assigned by MongoDB
```

Note: If no `_id` field is explicitly defined, MongoDB automatically generates a unique identifier.

Inserting Multiple Documents

To insert multiple documents at once, use the **`insert_many()`** method.
This method takes a **list of dictionaries**, each representing a document.

Example: Inserting Multiple Documents

```
import pymongo

# Establish connection
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

# Define multiple documents
mylist = [
  { "name": "Amy", "address": "Apple St 652"},
  { "name": "Hannah", "address": "Mountain 21"},
  { "name": "Michael", "address": "Valley 345"},
  { "name": "Sandy", "address": "Ocean Blvd 2"},
  { "name": "Betty", "address": "Green Grass 1"},
  { "name": "Richard", "address": "Sky St 331"},
  { "name": "Susan", "address": "One Way 98"},
  { "name": "Vicky", "address": "Yellow Garden 2"},
  { "name": "Ben", "address": "Park Lane 38"},
  { "name": "William", "address": "Central St 954"},
  { "name": "Chuck", "address": "Main Road 989"},
  { "name": "Viola", "address": "Sideway 1633"}
]

# Insert multiple documents
x = mycol.insert_many(mylist)

# Print the IDs of inserted documents
print(x.inserted_ids)
```

The **`inserted_ids`** attribute contains the list of unique IDs assigned to each document.

Inserting Multiple Documents with Custom IDs

If you want to manually define `_id` values for documents, include them in the dictionary. Ensure that each `_id` is unique to prevent errors.

Example: Inserting Documents with Specified IDs

```
import pymongo

# Establish connection
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

# Define multiple documents with specified IDs
mylist = [
  { "_id": 1, "name": "John", "address": "Highway 37"},
  { "_id": 2, "name": "Peter", "address": "Lowstreet 27"},
  { "_id": 3, "name": "Amy", "address": "Apple St 652"},
  { "_id": 4, "name": "Hannah", "address": "Mountain 21"},
  { "_id": 5, "name": "Michael", "address": "Valley 345"},
  { "_id": 6, "name": "Sandy", "address": "Ocean Blvd 2"},
  { "_id": 7, "name": "Betty", "address": "Green Grass 1"},
  { "_id": 8, "name": "Richard", "address": "Sky St 331"},
  { "_id": 9, "name": "Susan", "address": "One Way 98"},
  { "_id": 10, "name": "Vicky", "address": "Yellow Garden 2"},
  { "_id": 11, "name": "Ben", "address": "Park Lane 38"},
  { "_id": 12, "name": "William", "address": "Central St 954"},
  { "_id": 13, "name": "Chuck", "address": "Main Road 989"},
  { "_id": 14, "name": "Viola", "address": "Sideway 1633"}
]

# Insert multiple documents
x = mycol.insert_many(mylist)

# Print list of inserted document IDs
print(x.inserted_ids)
```

With these methods, you can efficiently insert and manage documents in MongoDB using Python!

Here’s a reworded version of your text while keeping the original meaning intact:

Python MongoDB Querying Data

In MongoDB, data retrieval from a collection is accomplished using the `find()` and `find_one()` methods. These functions serve a similar purpose to the `SELECT` statement in MySQL databases.

Retrieving a Single Document

To fetch a single document from a MongoDB collection, the `find_one()` method is used. This function returns the first matching record based on the query criteria.

Example: Fetching the First Record

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

x = mycol.find_one()

print(x)
```

Retrieving Multiple Documents

To fetch all records from a MongoDB collection, the `find()` method is utilized. This function retrieves all occurrences matching the given query.

If no query is provided, it fetches all available documents, similar to using `SELECT *` in MySQL.

Example: Fetching All Documents

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find():
  print(x)
```

Selecting Specific Fields

The `find()` method also allows specifying which fields should be included in the output. This is done by passing an additional dictionary as a parameter.

If no field selection is provided, all fields are retrieved by default.

Example: Fetching Only Names and Addresses (Excluding `_id`)

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({}, { "_id": 0, "name": 1, "address": 1 }):
  print(x)
```

Field Inclusion and Exclusion Rules

MongoDB does not allow mixing inclusion (`1`) and exclusion (`0`) values in the same projection, except when excluding `_id`.

Example: Excluding Only the Address Field

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({}, { "address": 0 }):
  print(x)
```

Example: Incorrect Field Inclusion and Exclusion

The following code will result in an error since it attempts to include `name` and exclude `address` simultaneously:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({}, { "name": 1, "address": 0 }):
  print(x)
```

This version keeps the essence of your text while improving clarity, structure, and readability. □ Let me know if you need further modifications! □

Here's a refined and reworded version of your text while maintaining the original meaning:

Python MongoDB Querying with Filters

When retrieving documents from a MongoDB collection, filtering can be done using a query object.

The `find()` method accepts a query object as its first argument, which helps narrow down the search results.

Filtering Results

To retrieve specific documents based on a condition, a query object is used.

Example: Finding Documents with a Specific Address

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Park Lane 38" }

mydoc = mycol.find(myquery)

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

Performing Advanced Queries

For more complex queries, modifiers can be used within the query object.

For instance, to find documents where the "address" field starts with a letter **S** or comes later alphabetically, the **greater than** (`$gt`) modifier can be applied:

Example: Retrieving Addresses Starting from 'S' Onward

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$gt": "S" } }

mydoc = mycol.find(myquery)

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

Using Regular Expressions for Filtering

MongoDB also supports **regular expressions** to filter string-based fields.

To retrieve only those documents where the "address" field begins with the letter **S**, use the `$regex` operator:

Example: Filtering Addresses That Start with 'S'

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }

mydoc = mycol.find(myquery)

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

This refined version keeps the content clear, structured, and professional. □ Let me know if you need any further refinements! □

Python MongoDB Sorting

Sorting Query Results

In MongoDB, the `sort()` method is used to arrange query results in either **ascending** or **descending** order.

The method requires two parameters:

1. The **field name** by which to sort.
2. The **sorting order**, where **ascending (1) is the default** and **descending (-1) must be explicitly specified**.

Sorting in Ascending Order

The following example sorts the documents alphabetically based on the **"name"** field:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]
```

```
mydoc = mycol.find().sort("name")

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

Sorting in Descending Order

To sort results in **reverse order** (Z to A), use `-1` as the second parameter.

Example: Sorting Names in Descending Order

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name", -1)

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

✅**Ascending Sorting:** `sort("name", 1)`
✅**Descending Sorting:** `sort("name", -1)`

This approach ensures well-organized query results, making data retrieval more efficient!

Python MongoDB Ordering

Arranging Query Results

MongoDB allows sorting query results using the `sort()` method. This method organizes documents based on a specified field, in either **ascending** or **descending** order.

Sorting in Ascending Order (Default)

By default, sorting is done in **ascending order (A-Z, 0-9)**.

Example: Sorting Names Alphabetically

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name")
```

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

Sorting in Descending Order

To sort results in **reverse order (Z-A, 9-0)**, use `-1` as the second argument.

Example: Sorting Names in Descending Order

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name", -1)

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

✅**Ascending Sorting:** `sort("name", 1)`
✅**Descending Sorting:** `sort("name", -1)`

Sorting helps in efficiently retrieving structured and meaningful data!

Python MongoDB - Removing Documents

Deleting a Single Document

To remove a **specific** document, use the `delete_one()` method. The first parameter is a **query object** that defines which document should be deleted.

 Note: If multiple documents match the query, only **the first occurrence** will be deleted.

Example: Deleting a Document by Address

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Mountain 21" }

mycol.delete_one(myquery)
```

Deleting Multiple Documents

To remove multiple documents at once, use `delete_many()`.

Example: Deleting Documents Where Address Starts with 'S'

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": {"$regex": "^S"} }

x = mycol.delete_many(myquery)

print(x.deleted_count, "documents deleted.")
```

Deleting All Documents from a Collection

To **clear** an entire collection, pass an **empty query object** `{}` to `delete_many()`.

Example: Removing All Documents from a Collection

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

x = mycol.delete_many({})

print(x.deleted_count, "documents deleted.")
```

 Efficiently managing data deletion ensures optimal database performance!

Python Data Structures and Algorithms

1. Lists in Python

Overview

- Lists are **ordered and mutable** collections.
- Can store mixed data types.
- Implemented as **dynamic arrays**.

Operations

```
# Defining a list
my_list = [1, 2, 3, 4, 5]

# Adding elements
my_list.append(6)  # Appends 6
my_list.insert(2, 10)  # Inserts 10 at index 2

# Removing elements
my_list.pop()  # Removes the last element
my_list.remove(10)  # Removes the first occurrence of 10

# Slicing
print(my_list[1:4])  # Displays elements from index 1 to 3
```

---

2. Tuples in Python

Overview

- Tuples are **immutable and ordered** collections.
- Faster than lists due to immutability.

Operations

```
# Creating a tuple
tuple1 = (1, 2, 3, "Python")

# Accessing elements
print(tuple1[1])  # Outputs 2

# Slicing
tuple2 = tuple1[1:3]  # Result: (2, 3)
```

---

3. Sets and Dictionaries in Python

Sets

- Unordered collections with **unique** elements.

```
# Defining a set
my_set = {1, 2, 3, 4}
my_set.add(5)
my_set.remove(2)
print(my_set)
```

Dictionaries

- **Key-value pairs** that are unordered but highly efficient.

```
# Creating a dictionary
```

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

---

4. Queue in Python

Overview

- Works on **FIFO (First-In-First-Out)** principle.

```
from collections import deque
queue = deque([1, 2, 3])
queue.append(4)  # Enqueue
queue.popleft()  # Dequeue
print(queue)
```

---

5. Linked List in Python

Overview

- A **dynamic** structure where each node points to the next.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        temp = self.head
        while temp.next:
            temp = temp.next
        temp.next = new_node

    def display(self):
        temp = self.head
        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
        print("None")

ll = LinkedList()
ll.insert_at_end(1)
ll.insert_at_end(2)
ll.insert_at_end(3)
```

```
ll.display()
```

---

6. Stack in Python

Overview

- Follows **LIFO (Last-In-First-Out)** principle.

```
stack = []
stack.append(1)
stack.append(2)
stack.append(3)
print(stack.pop())  # Outputs 3
print(stack)
```

---

7. Graphs in Python

Overview

- Consists of **nodes (vertices) and edges**.
- Represented using an adjacency list.

```
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

print(graph['A'])  # Outputs ['B', 'C']
```

---

8. Trees in Python

Overview

- A hierarchical structure with **nodes**.

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def inorder_traversal(root):
    if root:
```

```
        inorder_traversal(root.left)
        print(root.value, end=' ')
        inorder_traversal(root.right)

root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
inorder_traversal(root)
```

Final Thoughts

This guide introduces **fundamental data structures** in Python along with example implementations. Each structure serves a different purpose and has **unique efficiency trade-offs**. Mastering these concepts is crucial for coding interviews and practical applications.

Here's the rewritten version of your text with the same meaning but different wording:

Python MongoDB: Modifying Documents

Modifying a Collection

In MongoDB, you can modify a document (also called a record) using the `update_one()` method.

- The first argument in `update_one()` is a query that specifies which document should be updated.
- **Important:** If multiple records match the query, only the **first** one is modified.
- The second argument defines the new values to be assigned to the document.

Example: Updating a Single Document

The following example updates the address field from **"Valley 345"** to **"Canyon 123"**:

```
import pymongo

# Establish a connection to MongoDB
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

# Define the query and update values
myquery = { "address": "Valley 345" }
newvalues = { "$set": { "address": "Canyon 123" } }

# Apply the update
mycol.update_one(myquery, newvalues)
```

```
# Print all documents after modification
for x in mycol.find():
    print(x)
```

Updating Multiple Records

To update **all** documents matching specific criteria, use the `update_many()` function.

Example: Modifying Multiple Documents

The following example updates every record where the address starts with "S" and changes the name field to **"Minnie"**:

```
import pymongo

# Connect to MongoDB
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

# Query to find addresses starting with "S"
myquery = { "address": { "$regex": "^S" } }
newvalues = { "$set": { "name": "Minnie" } }

# Apply bulk update
x = mycol.update_many(myquery, newvalues)

# Output number of modified records
print(x.modified_count, "documents updated.")
```

Stay on Track with Your Learning!

 Get Certified in MongoDB and track your progress easily.

 Try the Color Picker Tool to explore different color combinations!

This version is clearer, more engaging, and professionally structured while keeping the core meaning intact. Let me know if you need any further refinements!

Here’s a refined version of your text with improved clarity and structure:

Python MongoDB: Limiting Query Results

Using the `limit()` Method

MongoDB allows you to **limit** the number of documents returned in a query using the `limit()` method.

- The `limit()` function takes a **single parameter**, specifying how many documents to return.
- This is useful for optimizing queries and improving performance when dealing with large datasets.

Example: Sample "customers" Collection

Consider the following documents stored in the **"customers"** collection:

```
{ "_id": 1, "name": "John", "address": "Highway 37" }
{ "_id": 2, "name": "Peter", "address": "Lowstreet 27" }
{ "_id": 3, "name": "Amy", "address": "Apple st 652" }
{ "_id": 4, "name": "Hannah", "address": "Mountain 21" }
{ "_id": 5, "name": "Michael", "address": "Valley 345" }
{ "_id": 6, "name": "Sandy", "address": "Ocean blvd 2" }
{ "_id": 7, "name": "Betty", "address": "Green Grass 1" }
{ "_id": 8, "name": "Richard", "address": "Sky st 331" }
{ "_id": 9, "name": "Susan", "address": "One way 98" }
{ "_id": 10, "name": "Vicky", "address": "Yellow Garden 2" }
{ "_id": 11, "name": "Ben", "address": "Park Lane 38" }
{ "_id": 12, "name": "William", "address": "Central st 954" }
{ "_id": 13, "name": "Chuck", "address": "Main Road 989" }
{ "_id": 14, "name": "Viola", "address": "Sideway 1633" }
```

Example: Limiting Query Results

The following Python script retrieves **only the first 5 documents** from the collection:

```
import pymongo

# Connect to MongoDB
myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

# Retrieve only 5 documents
myresult = mycol.find().limit(5)

# Print the results
for x in myresult:
    print(x)
```

Why Use `limit()`?

✅**Faster Queries:** Reduces the amount of data retrieved, improving efficiency.
✅**Better Performance:** Saves memory by avoiding unnecessary document retrieval.
✅**Pagination:** Useful when implementing page-based navigation in web applications.

Stay on Track with Your Learning!

 Track Your Progress & Get Certified in MongoDB today!

 Explore the Color Picker Tool for designing visually appealing web applications.

This version presents the information in a **more structured, engaging, and professional** way while keeping the original meaning intact. Let me know if you need further refinements!

Data Structures and Algorithms in Python

---

1. Introduction to Data Structures & Algorithms (DSA)

□ What are Data Structures?

- **Definition:** A systematic way to store and organize data to facilitate efficient access and modifications.
- **Categories:**
  - **Linear**: Array, Linked List, Stack, Queue
  - **Non-linear**: Tree, Graph, Hash Table

□ What are Algorithms?

- **Definition:** A finite set of well-defined instructions to solve a problem.
- **Characteristics:** Correctness, Efficiency, Finiteness, Definiteness, Input/Output.

□ Algorithm Complexity

Complexity	Example Algorithm
O(1)O(1)O(1)	Accessing array element
O(log n)O(\log n)O(logn)	Binary Search
O(n)O(n)O(n)	Linear Search
O(nlog n)O(n \log n)O(nlogn)	Quick Sort
O(n2)O(n^2)O(n2)	Bubble Sort

---

2. Linear Data Structures

□ Linked Lists (with Diagram & Real-World Use Case)

□ Real-World Application:

- Used in **Music Playlists** where each song links to the next one.

□ Diagram:

```
rust
CopyEdit
Head -> [10 | * ] -> [20 | * ] -> [30 | * ] -> None
```

Python Implementation:

```
python
CopyEdit
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node
```

---

3. Advanced Data Structures

□ Trees (with Diagram & Real-World Use Case)

□ Real-World Application:

- Used in **file systems** (folders inside folders).

□ Diagram of a Binary Tree:

```
markdown
CopyEdit
        10
       /  \
      5    15
     / \   / \
    2   7 12 20
```

Python Implementation:

```
python
CopyEdit
class TreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

□ Tries (Used in Google Search Autocomplete)

□ Real-World Application:

- **Google Search suggestions** store prefixes in a Trie.

□ Diagram:

```
markdown
CopyEdit
       (Root)
      /   |   \
     c    h    t
    /     |     \
   a      e      r
  /      / \      \
 t      n   y      e
       /     \
      d       e
```

Python Implementation:

```
python
CopyEdit
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True
```

□ Segment Trees (Used in Range Queries)

□ Real-World Application:

- Used in **database query optimization** for fast range sums.

Python Implementation:

```
python
CopyEdit
```

```
class SegmentTree:
    def __init__(self, arr):
        self.n = len(arr)
        self.tree = [0] * (2 * self.n)
        self.build(arr)

    def build(self, arr):
        for i in range(self.n):
            self.tree[self.n + i] = arr[i]
        for i in range(self.n - 1, 0, -1):
            self.tree[i] = self.tree[2 * i] + self.tree[2 * i + 1]

    def query(self, left, right):
        res = 0
        left += self.n
        right += self.n
        while left < right:
            if left % 2:
                res += self.tree[left]
                left += 1
            if right % 2:
                right -= 1
                res += self.tree[right]
            left //= 2
            right //= 2
        return res
```

□ Dynamic Programming (Used in Path Finding & Games)

□ Real-World Application:

- Used in **Google Maps** to find the shortest route.

Example: Fibonacci using DP

```
python
CopyEdit
def fibonacci(n, memo={}):
    if n in memo:
        return memo[n]
    if n <= 2:
        return 1
    memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)
    return memo[n]
```
