Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

☑ LISTS, TUPLES & DICTIONARIES IN PYTHON

CHAPTER 1: INTRODUCTION TO DATA STRUCTURES IN PYTHON

Python provides **various data structures** to store and organize data. Three of the most commonly used are:

- **Lists**: Ordered, mutable (changeable), and allows duplicate values.
- **Tuples**: Ordered, immutable (unchangeable), and allows duplicate values.
- **Dictionaries**: Key-value pairs, unordered (before Python 3.7), mutable, and unique keys.

These structures help store and manipulate collections of data efficiently.

CHAPTER 2: LISTS IN PYTHON

2.1 What is a List?

A **list** in Python is an **ordered collection** that can hold multiple items. It is **mutable,** meaning you can modify it after creation.

2.2 Creating a List

Lists are defined using **square brackets [ ],** and elements are separated by commas.

📌 **Example:**

```
fruits = ["Apple", "Banana", "Cherry"]

numbers = [1, 2, 3, 4, 5]

mixed_list = ["Python", 3.14, True, 42]
```

2.3 Accessing Elements in a List

You can access list elements using **indexing,** where the index starts from **0**.

📌 **Example:**

```
fruits = ["Apple", "Banana", "Cherry"]

print(fruits[0])  # Output: Apple

print(fruits[1])  # Output: Banana

print(fruits[-1]) # Output: Cherry (Negative Indexing)
```

2.4 Modifying a List

Lists are **mutable,** meaning you can change their elements.

📌 **Example:**

```
fruits = ["Apple", "Banana", "Cherry"]

fruits[1] = "Mango"
```

```
print(fruits)  # Output: ['Apple', 'Mango', 'Cherry']
```

2.5 Adding & Removing Elements in a List

✓ Adding Elements:

- append() – Adds an item to the end.
- insert() – Adds an item at a specific index.

📌 Example:

```
fruits.append("Orange")  # Adds at the end
fruits.insert(1, "Grapes")  # Adds at index 1
print(fruits)  # Output: ['Apple', 'Grapes', 'Mango', 'Cherry', 'Orange']
```

✓ Removing Elements:

- remove() – Removes the first occurrence of an item.
- pop() – Removes an element by index or the last element if no index is given.

📌 Example:

```
fruits.remove("Cherry")
fruits.pop(0)
print(fruits)  # Output: ['Grapes', 'Mango', 'Orange']
```

2.6 Looping Through a List

📌 **Example:** Using a **for loop** to print all items in a list.

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

CHAPTER 3: TUPLES IN PYTHON

3.1 What is a Tuple?

A **tuple** is similar to a list but **immutable**, meaning it **cannot be changed** after creation.

3.2 Creating a Tuple

Tuples are defined using **parentheses ( )**.

📌 **Example:**

```
fruits_tuple = ("Apple", "Banana", "Cherry")
numbers_tuple = (1, 2, 3, 4, 5)
single_item_tuple = ("Python",)  # A comma is required for a single-item tuple
```

3.3 Accessing Elements in a Tuple

Tuple elements are accessed using **indexing**, just like lists.

📌 **Example:**

```
print(fruits_tuple[0])  # Output: Apple
print(fruits_tuple[-1]) # Output: Cherry
```

3.4 Why Use Tuples Instead of Lists?

✓ **Faster:** Tuples are stored more efficiently than lists.
✓ **Immutable:** Data security is higher as values cannot be changed.
✓ **Hashable:** Tuples can be used as keys in dictionaries.

📌 **Example:**

```
coordinates = (10, 20)

print(coordinates)  # Output: (10, 20)
```

3.5 Tuple Packing & Unpacking

Tuples allow **packing multiple values** and extracting them easily.

📌 **Example:**

```
person = ("Alice", 25, "Engineer")  # Tuple Packing

name, age, profession = person  # Tuple Unpacking

print(name)  # Output: Alice

print(age)   # Output: 25

print(profession)  # Output: Engineer
```

CHAPTER 4: DICTIONARIES IN PYTHON

4.1 What is a Dictionary?

A **dictionary** stores data in **key-value pairs**. Unlike lists and tuples, dictionary elements are accessed using **keys instead of index numbers**.

4.2 Creating a Dictionary

Dictionaries are defined using **curly braces {}** with key-value pairs.

📌 **Example:**

```
student = {
  "name": "Alice",
  "age": 25,
  "grade": "A"
}
```

4.3 Accessing Dictionary Values

Values are accessed using **keys** instead of indexes.

📌 **Example:**

```
print(student["name"])  # Output: Alice
print(student.get("age"))  # Output: 25
```

4.4 Modifying a Dictionary

Dictionaries are **mutable**, so you can change, add, and remove elements.

✓ **Modifying Values:**

📌 **Example:**

```
student["age"] = 26
print(student)  # Output: {'name': 'Alice', 'age': 26, 'grade': 'A'}
```

✓ **Adding New Key-Value Pairs:**

📌 **Example:**

```
student["city"] = "New York"

print(student)  # Output: {'name': 'Alice', 'age': 26, 'grade': 'A', 'city': 'New York'}
```

✓ **Removing Elements:**

📌 **Example:**

```
del student["grade"]

print(student)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

4.5 Looping Through a Dictionary

You can loop through **keys, values, or both** using a for loop.

📌 **Example:**

```
for key, value in student.items():
   print(key, ":", value)
```

✓ **Output:**

```
name : Alice
age : 26
city : New York
```

CHAPTER 5: EXERCISE

5.1 Multiple Choice Questions

1. What is a key difference between a list and a tuple?
    - (a) Lists are immutable, tuples are mutable
    - (b) Lists use parentheses, tuples use square brackets
    - (c) Lists are mutable, tuples are immutable
    - (d) Lists store key-value pairs
2. How do you access a value in a dictionary?
    - (a) Using an index number
    - (b) Using a key
    - (c) Using .getValue()
    - (d) Using .index()
3. What will print(fruits_tuple[1]) output if fruits_tuple = ("Apple", "Banana", "Cherry")?
    - (a) Apple
    - (b) Banana
    - (c) Cherry
    - (d) Error

5.2 Practical Tasks

📌 **Task 1: Create a list of five favorite movies and print each one using a loop.**

📌 **Task 2: Create a tuple containing three city names and access the second city.**

📌 **Task 3: Create a dictionary with information about a book (title, author, year) and print the book's title.**

CHAPTER 6: SUMMARY

✅ **Lists** are ordered, mutable, and allow duplicates.
✅ **Tuples** are ordered, immutable, and allow duplicates.
✅ **Dictionaries** store key-value pairs and provide fast lookups.

☑ File Handling – Reading & Writing Files in Python

Chapter 1: Introduction to File Handling

File handling in Python allows us to **create, read, update, and delete files** from a program. It is useful for storing data **permanently**, even after a program stops running.

Python provides built-in functions to work with files, such as:

✓ open() – Opens a file
✓ read() – Reads data from a file
✓ write() – Writes data to a file
✓ close() – Closes a file after use

Understanding file handling is **crucial** for working with large datasets, configurations, or saving user-generated content.

Chapter 2: Opening a File in Python

Before reading or writing to a file, we must **open it** using the open() function.

📌 **Syntax:**

file = open("filename.txt", "mode")

- "filename.txt" is the name of the file.
- "mode" specifies how the file should be opened (read, write, append, etc.).

CHAPTER 3: FILE OPENING MODES

Mode	Description
r	Read mode (default) – Opens file for reading, gives an error if the file does not exist.
w	Write mode – Opens file for writing, **overwrites** existing content or creates a new file.
a	Append mode – Opens file for writing, **adds new content** without removing old content.
x	Create mode – Creates a new file, gives an error if the file exists.
rb, wb	Read and write binary files (images, videos, etc.).

CHAPTER 4: READING A FILE IN PYTHON

4.1 Reading the Entire File

📌 Example: Reading a file using read()

```
file = open("example.txt", "r")  # Open file in read mode
content = file.read()  # Read the entire file content
print(content)
file.close()  # Always close the file
```

✓ Output (if example.txt contains "Hello, Python!")

Hello, Python!

4.2 Reading a File Line by Line

📌 **Example: Using readline() to read one line at a time**

```
file = open("example.txt", "r")

line1 = file.readline()  # Reads first line

line2 = file.readline()  # Reads second line

print(line1, line2)

file.close()
```

This is useful for reading large files **line by line** instead of loading the whole file into memory.

📌 **Example: Using readlines() to read all lines into a list**

```
file = open("example.txt", "r")

lines = file.readlines()  # Stores all lines in a list

for line in lines:

   print(line.strip())  # Print each line after removing extra spaces

file.close()
```

CHAPTER 5: WRITING TO A FILE IN PYTHON

5.1 Writing Content to a File

To write data to a file, we use **write mode (w) or append mode (a)**.

📌 **Example: Writing to a File using write()**

```
file = open("example.txt", "w")  # Open file in write mode
file.write("Hello, Python!\n")
file.write("This is a new file.\n")
file.close()
```

✓ **Content of example.txt after execution:**

```
Hello, Python!
This is a new file.
```

📌 **Warning:** Using "w" mode **overwrites** the existing content of the file.

5.2 Appending Content to a File

If you want to add new content **without erasing existing content,** use **append mode (a)**.

📌 **Example: Adding More Text to an Existing File**

```
file = open("example.txt", "a")  # Open file in append mode
file.write("Appending new content.\n")
file.close()
```

✓ **Updated Content of example.txt:**

```
Hello, Python!
```

This is a new file.

Appending new content.

CHAPTER 6: USING WITH STATEMENT FOR FILE HANDLING

Using the with statement is **recommended** because it **automatically closes the file** after operations.

📌 **Example: Reading a File Using with open()**

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)  # File is automatically closed after this block
```

📌 **Example: Writing to a File Using with open()**

```
with open("example.txt", "w") as file:
    file.write("This file was written using 'with' statement.\n")
```

CHAPTER 7: WORKING WITH DIFFERENT FILE TYPES

7.1 Handling CSV Files

CSV (Comma-Separated Values) files are widely used for **data storage**.

📌 **Example: Writing to a CSV File**

```
import csv
```

```
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age", "City"])
    writer.writerow(["Alice", 25, "New York"])
```

📌 **Example: Reading from a CSV File**

```
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

7.2 Handling JSON Files

JSON (JavaScript Object Notation) is used for storing and exchanging data.

📌 **Example: Writing to a JSON File**

```
import json

data = {"name": "Alice", "age": 25, "city": "New York"}

with open("data.json", "w") as file:
    json.dump(data, file)
```

📌 **Example: Reading from a JSON File**

```
with open("data.json", "r") as file:
    data = json.load(file)
    print(data)
```

Chapter 8: Error Handling in File Operations

When working with files, errors can occur if a file **doesn't exist** or is **not accessible**.

📌 **Example: Handling File Not Found Error**

```
try:
    with open("missing_file.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("Error: The file does not exist.")
```

✓ **Output:**

```
Error: The file does not exist.
```

Chapter 9: Exercise

9.1 Multiple Choice Questions

1. What mode is used to open a file for reading?
    - (a) "r"
    - (b) "w"

- (c) "a"
- (d) "x"

2. Which function is used to read the entire content of a file?
   - (a) readall()
   - (b) fetch()
   - (c) read()
   - (d) get()
3. What happens when you open a file in "w" mode?
   - (a) It appends content to the file.
   - (b) It overwrites the file if it exists.
   - (c) It reads the file.
   - (d) It only opens the file without making changes.

9.2 Practical Tasks

📌 **Task 1: Create a text file and write your name and age into it.**

```
with open("info.txt", "w") as file:
  file.write("Name: Alice\nAge: 25")
```

📌 **Task 2: Read and print the content of the file you just created.**

```
with open("info.txt", "r") as file:
  print(file.read())
```

📌 **Task 3: Append a new line to the file.**

```
with open("info.txt", "a") as file:
    file.write("\nCity: New York")
```

CHAPTER 10: SUMMARY

✅ Python provides open(), read(), write(), and close() functions for file handling.
✅ "r", "w", and "a" modes allow reading, writing, and appending.
✅ with open() ensures files are **automatically closed** after use.
✅ Python can handle **CSV, JSON, and text files** efficiently.

☑ Introduction to Object-Oriented Programming (OOP) Basics

Chapter 1: What is Object-Oriented Programming (OOP)?

Object-Oriented Programming (**OOP**) is a programming paradigm based on the concept of **objects**.

- An **object** is a real-world entity with **attributes (properties)** and **behaviors (methods)**.
- OOP allows programmers to model real-world scenarios efficiently by grouping **data and functions together**.

📌 **Example of Real-World Object:**
A **Car** has:
✓ **Attributes (Properties)**: Color, Brand, Model, Speed
✓ **Methods (Behaviors)**: Start, Stop, Accelerate, Brake

In Python, OOP helps in writing **modular, reusable, and scalable** code.

Chapter 2: Key Concepts of OOP

2.1 Classes & Objects

- A **class** is a **blueprint** for creating objects.
- An **object** is an **instance** of a class.

📌 **Example:** Defining a Car class and creating an object.

class Car:

```
  def __init__(self, brand, color):
    self.brand = brand
    self.color = color

car1 = Car("Toyota", "Red")  # Creating an object
print(car1.brand)  # Output: Toyota
print(car1.color)  # Output: Red
```

2.2 Encapsulation

Encapsulation is the concept of **restricting direct access** to data by **using methods** to control modifications.

- It is achieved using **private variables** and **getter/setter methods**.

📌 **Example: Encapsulation in Python**

```
class BankAccount:
  def __init__(self, balance):
    self.__balance = balance  # Private variable (Cannot be accessed directly)

  def get_balance(self):
    return self.__balance
```

```
    def deposit(self, amount):
        self.__balance += amount


account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

✓ The variable __balance is **private** and cannot be accessed directly.

2.3 Inheritance

Inheritance allows one class (**child class**) to **inherit properties and methods** from another class (**parent class**).

- This helps in **code reusability** and **reducing redundancy**.

📌 **Example: Inheritance in Python**

```
class Animal:
    def make_sound(self):
        print("Animal makes a sound")


class Dog(Animal):  # Dog class inherits from Animal
    def bark(self):
        print("Dog barks")
```

```
dog = Dog()
dog.make_sound()  # Output: Animal makes a sound
dog.bark()       # Output: Dog barks
```

✓ The Dog class **inherits** the make_sound() method from Animal.

2.4 Polymorphism

Polymorphism means **one function behaves differently based on the object calling it**.

- This allows **method overriding** and **function overloading**.

📌 **Example: Polymorphism in Python**

```
class Bird:
    def sound(self):
        print("Bird chirps")

class Cat:
    def sound(self):
        print("Cat meows")

# Using the same function for different objects
def make_sound(animal):
    animal.sound()
```

```
bird = Bird()

cat = Cat()


make_sound(bird)  # Output: Bird chirps

make_sound(cat)   # Output: Cat meows
```

✓ The sound() method works differently for Bird and Cat objects.

2.5 Abstraction

Abstraction hides the **implementation details** and only shows the **essential features** to the user.

- It is achieved using **abstract classes** and **abstract methods**.

📌 **Example: Abstraction in Python (Using abc module)**

```
from abc import ABC, abstractmethod


class Vehicle(ABC):  # Abstract class
   @abstractmethod
   def fuel_type(self):
      pass  # Method without implementation


class Car(Vehicle):
```

```
  def fuel_type(self):
    return "Petrol"

car = Car()
print(car.fuel_type())  # Output: Petrol
```

✓ fuel_type() is defined but **not implemented** in Vehicle, ensuring every subclass must define it.

CHAPTER 3: WHY USE OOP?

✓ **Code Reusability** – Inheritance allows the reuse of code, reducing duplication.
✓ **Encapsulation** – Protects data from accidental modification.
✓ **Modularity** – Organizes code into smaller, manageable parts.
✓ **Scalability** – Allows easy expansion of functionality.

📌 **Real-World Example:**

- **E-Commerce Application**: A User class can be inherited by Customer and Seller.
- **Gaming Development**: A Character class can be inherited by Warrior, Mage, and Archer.

CHAPTER 4: EXERCISE

4.1 Multiple Choice Questions

1. What is a class in OOP?

- (a) A variable type
- (b) A blueprint for creating objects
- (c) A function
- (d) A library

2. What is the concept of hiding implementation details called?
   - (a) Inheritance
   - (b) Encapsulation
   - (c) Abstraction
   - (d) Polymorphism
3. Which OOP principle allows different classes to use the same method name?
   - (a) Inheritance
   - (b) Abstraction
   - (c) Polymorphism
   - (d) Encapsulation

4.2 Practical Tasks

📌 Task 1: Create a Class for a Bank Account with Deposit and Withdraw Methods

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

```
  def deposit(self, amount):
    self.__balance += amount

  def withdraw(self, amount):
    if amount <= self.__balance:
      self.__balance -= amount
    else:
      print("Insufficient funds!")

  def get_balance(self):
    return self.__balance

account = BankAccount(1000)
account.deposit(500)
account.withdraw(300)
print(account.get_balance())  # Output: 1200
```

📌 Task 2: Implement a Class Hierarchy Using Inheritance (Vehicle → Car, Bike)

```
class Vehicle:
  def start_engine(self):
```

```
    print("Engine started")


class Car(Vehicle):
  def drive(self):
    print("Car is driving")


class Bike(Vehicle):
  def ride(self):
    print("Bike is riding")


car = Car()
bike = Bike()


car.start_engine()
car.drive()


bike.start_engine()
bike.ride()
```

✓ Expected Output:

Engine started

Car is driving

Engine started

Bike is riding

CHAPTER 5: SUMMARY

✅ **OOP** is a programming paradigm based on **objects and classes**.
✅ **Encapsulation** protects data using private variables and methods.
✅ **Inheritance** allows one class to inherit from another, improving **code reuse**.
✅ **Polymorphism** enables different classes to have methods with the same name but different behaviors.
✅ **Abstraction** hides the implementation details and focuses on essential features.

☑ Building Small Python Projects

Python is a beginner-friendly yet powerful programming language that can be used to build **fun and useful projects**. In this chapter, we will build **two small Python projects**:

1. **Password Generator** – A program that creates random passwords for users.
2. **Guess the Number Game** – A game where the player guesses a randomly chosen number.

These projects will help you practice **user input, loops, conditional statements, and the random module** in Python.

📌 Project 1: Password Generator

🎯 Objective:

Create a Python program that generates a **strong random password** with uppercase and lowercase letters, numbers, and special symbols.

Step 1: Import the Required Modules

Python has a built-in random module that helps in generating random numbers and characters.

```
import random

import string
```

Step 2: Define the Password Generator Function

The function will:

- Accept user input for password **length**.
- Use letters, numbers, and symbols.
- Generate a **random password**.

📌 **Code:**

```
def generate_password(length):
    characters = string.ascii_letters + string.digits + string.punctuation  # All possible characters
    password = ''.join(random.choice(characters) for _ in range(length))  # Generate password
    return password
```

Step 3: Ask the User for Input and Generate a Password

📌 **Code:**

```
length = int(input("Enter password length: "))
new_password = generate_password(length)
print("Generated Password:", new_password)
```

✓ **Example Output:**

```
Enter password length: 12
Generated Password: d@9X^f7&h2L!
```

Step 4: Full Code of the Password Generator

```
import random

import string


def generate_password(length):

    characters = string.ascii_letters + string.digits + string.punctuation

    password = ''.join(random.choice(characters) for _ in range(length))

    return password


length = int(input("Enter password length: "))

new_password = generate_password(length)

print("Generated Password:", new_password)
```

🔍 Challenge Yourself!

✅ Add an option for **user-defined password strength** (weak, medium, strong).
✅ Save the generated password to a **file** for later use.
✅ Create a **GUI-based password generator** using Tkinter.

📌 Project 2: Guess the Number Game

🎯 Objective:

Create a game where:

- The computer **chooses a random number** between 1 and 100.
- The player has to **guess the number**.
- The program provides **hints** if the guess is too high or too low.

Step 1: Import the Random Module

The random.randint() function helps generate a random number.

import random

Step 2: Define the Game Logic

📌 Code:

```
def guess_the_number():
  secret_number = random.randint(1, 100)  # Generate a random number
  attempts = 0

  print("Welcome to Guess the Number!")
  print("Try to guess the number between 1 and 100.")
```

```
while True:
    guess = int(input("Enter your guess: "))
    attempts += 1

    if guess < secret_number:
        print("Too low! Try again.")
    elif guess > secret_number:
        print("Too high! Try again.")
    else:
        print(f"Congratulations! You guessed it in {attempts} attempts.")
        break  # Exit the loop
```

Step 3: Start the Game

📌 **Code:**

```
guess_the_number()
```

✓ **Example Output:**

Welcome to Guess the Number!

Try to guess the number between 1 and 100.

Enter your guess: 50

Too low! Try again.

Enter your guess: 75

Too high! Try again.

Enter your guess: 62

Congratulations! You guessed it in 3 attempts.

Step 4: Full Code for the Guess the Number Game

```
import random


def guess_the_number():
    secret_number = random.randint(1, 100)
    attempts = 0


    print("Welcome to Guess the Number!")
    print("Try to guess the number between 1 and 100.")


    while True:
        guess = int(input("Enter your guess: "))
        attempts += 1


        if guess < secret_number:
            print("Too low! Try again.")
```

```
    elif guess > secret_number:
        print("Too high! Try again.")
    else:
        print(f"Congratulations! You guessed it in {attempts} attempts.")
        break

guess_the_number()
```

🔍 Challenge Yourself!

✅ Add a **limit on the number of attempts** (e.g., 5 attempts).
✅ Make the game **multiplayer** (allow two players to take turns).
✅ Display the **score and fastest time** to track the best player.

📌 Summary of What You Learned

✅ **Password Generator:**
✓ Uses the random and string modules.
✓ Generates **random secure passwords**.
✓ Can be customized for **different password strengths**.

✅ **Guess the Number Game:**
✓ Uses random.randint() to **generate a random number**.
✓ Implements a **loop with conditions** to check user guesses.
✓ Improves **logical thinking and debugging skills**.

ASSIGNMENT 1:

DEVELOP A NUMBER GUESSING GAME USING PYTHON.

💡 Assignment Solution 1: Develop a Number Guessing Game Using Python

---

🎯 Objective:

✓ Learn how to **generate random numbers** in Python.
✓ Use **loops and conditional statements** to check the user's guess.
✓ Provide **feedback to the user** (Too high, Too low, Correct).
✓ Keep track of the number of attempts.

🛠 Step-by-Step Guide

Step 1: Import the Random Module

To generate a random number, you need to **import the random module**.

📌 Code:

```
import random
```

✓ Explanation:

- The random module allows us to generate a **random number** within a given range.

Step 2: Generate a Random Number

Use the randint() function to generate a **random number between 1 and 100**.

📌 Code:

```
secret_number = random.randint(1, 100)
```

✓ **Explanation:**

- randint(1, 100) generates a random number between **1 and 100**.
- The player will try to **guess this number**.

Step 3: Set Up the User Input and Loop

Since the user needs multiple attempts, use a **while loop** to keep asking for input until they guess correctly.

📌 **Code:**

```
attempts = 0


while True:
    guess = int(input("Guess a number between 1 and 100: "))
    attempts += 1  # Track number of attempts
```

✓ **Explanation:**

- attempts keeps track of the number of guesses.
- while True runs the loop **indefinitely** until the correct guess is made.
- int(input()) ensures the user input is converted to an **integer**.

Step 4: Compare the Guess with the Secret Number

Use **if-elif-else statements** to check if the guess is **too high, too low, or correct**.

📌 **Code:**

```
if guess < secret_number:
    print("Too low! Try again.")
elif guess > secret_number:
    print("Too high! Try again.")
else:
    print(f"Congratulations! You guessed the number in {attempts} attempts.")
    break  # Exit the loop when guessed correctly
```

✓ **Explanation:**

- If the guess is **lower** than the secret number, print "Too low!".
- If the guess is **higher**, print "Too high!".
- If the guess is **correct**, print a **congratulatory message** and **exit the loop** using break.

Step 5: Complete Python Program

Now, let's combine all the steps into a **single Python script**.

📌 **Final Code:**

```
import random

# Step 1: Generate a random number
```

```
secret_number = random.randint(1, 100)
attempts = 0

print("Welcome to the Number Guessing Game!")
print("Try to guess the number between 1 and 100.")

# Step 2: Loop until the user guesses correctly
while True:
  try:
    guess = int(input("Enter your guess: "))
    attempts += 1  # Increment attempt count

    if guess < secret_number:
      print("Too low! Try again.")
    elif guess > secret_number:
      print("Too high! Try again.")
    else:
      print(f"🎉 Congratulations! You guessed the number {secret_number} in {attempts} attempts.")
      break  # Exit the loop when guessed correctly

  except ValueError:
    print("Invalid input! Please enter a number.")
```

Step 6: Run and Test Your Game

Run the program and try guessing the number.

✓ Example Gameplay:

Welcome to the Number Guessing Game!

Try to guess the number between 1 and 100.

Enter your guess: 50

Too low! Try again.

Enter your guess: 75

Too high! Try again.

Enter your guess: 63

Too low! Try again.

Enter your guess: 70

🎉 Congratulations! You guessed the number 70 in 4 attempts.

🔍 Bonus: Enhancing the Game

To make the game **more interactive,** try adding:

✅ A **difficulty level** (Easy: 1-50, Hard: 1-200).

✅ A **maximum number of attempts** (e.g., **10 tries before losing**).

✅ A **score system** where the player earns points based on attempts.

📌 Example: Adding a Maximum Number of Attempts

max_attempts = 10

```
while attempts < max_attempts:
    guess = int(input("Enter your guess: "))
    attempts += 1

    if guess == secret_number:
        print(f" 🎉 You guessed it in {attempts} attempts!")
        break
    elif guess < secret_number:
        print("Too low!")
    else:
        print("Too high!")

if attempts == max_attempts:
    print(f"Game Over! The secret number was {secret_number}.")
```

✓ **Now, the player gets only 10 tries before losing!**

🎯 Summary

- ✅ Used random.randint() to generate a **random number**.
- ✅ Used a **while loop** to keep asking for guesses.
- ✅ Used if-elif-else to compare the guess and provide feedback.
- ✅ Included **error handling** using try-except to prevent input errors.

ASSIGNMENT 2:

CREATE A SIMPLE CHATBOT THAT RESPONDS TO BASIC USER QUERIES.

Assignment Solution 2: Create a Simple Chatbot

🎯 Objective:

✓ Learn how to use **user input, conditional statements, and functions** in Python.
✓ Create a chatbot that can answer **basic queries like greetings, weather, and name recognition**.
✓ Make the chatbot **interactive and customizable**.

🛠 Step-by-Step Guide

Step 1: Define the Chatbot's Purpose

Before writing code, decide what your chatbot should do. This chatbot will:

- Greet users
- Answer basic questions (e.g., "How are you?")
- Respond to name-based queries (e.g., "What is your name?")
- Exit when the user types **"bye"**

Step 2: Write the Chatbot Code

1. **Use an infinite loop** (while True) to keep the chatbot running.
2. **Take user input** using input().
3. **Use if-elif statements** to check what the user said and provide appropriate responses.

4. **Break the loop** when the user types "bye".

📌 **Code:**

```
def chatbot():
    print("Hello! I am ChatBot. Type 'bye' to exit.")

    while True:  # Infinite loop to keep the chatbot running
        user_input = input("You: ").lower()  # Convert input to lowercase for easier comparison

        if user_input in ["hello", "hi", "hey"]:
            print("ChatBot: Hello! How can I assist you today?")
        elif user_input in ["how are you?", "how are you"]:
            print("ChatBot: I'm just a bot, but I'm doing great! How about you?")
        elif user_input in ["what is your name?", "who are you?"]:
            print("ChatBot: I am ChatBot, your virtual assistant!")
        elif user_input in ["what can you do?", "help"]:
            print("ChatBot: I can chat with you and answer basic queries!")
        elif user_input in ["bye", "exit"]:
            print("ChatBot: Goodbye! Have a great day!")
            break  # Exit the loop
        else:
```

```
        print("ChatBot: Sorry, I don't understand that. Can you ask something else?")


chatbot()  # Call the chatbot function to start the conversation
```

Step 3: Understanding the Code

✓ while True – Keeps the chatbot running until the user types "bye".
✓ input("You: ") – Takes input from the user.
✓ .lower() – Converts the input to lowercase to avoid case-sensitive issues.
✓ if-elif – Checks the user's input and gives the correct response.
✓ break – Stops the chatbot when the user types "bye".

Step 4: Running the Chatbot

1. **Copy and paste** the code into a Python editor (IDLE, VS Code, or any Python environment).
2. **Run the script**.
3. **Chat with the bot** by typing different queries.
4. **Exit** by typing "bye".

✓ **Example Conversation:**

Hello! I am ChatBot. Type 'bye' to exit.

You: hi

ChatBot: Hello! How can I assist you today?

You: what is your name?

ChatBot: I am ChatBot, your virtual assistant!

You: bye

ChatBot: Goodbye! Have a great day!

🔍 Challenge Yourself!

✅ Add **more responses** for different queries (e.g., "Tell me a joke").
✅ Use **random module** to give different responses for the same input.
✅ Integrate **AI-based responses** using the openai or ChatterBot library.

📌 Summary

✅ The chatbot takes **user input** and provides **predefined responses**.
✅ It **understands basic greetings and questions**.
✅ It keeps running **until the user types "bye"**.
✅ The chatbot can be **enhanced with more responses and AI features**.