50 Days of Code - The Complete Python Pro Bootcamp

This comprehensive Python bootcamp offers an in-depth journey into Python programming, starting from beginner-level concepts and advancing to professional-grade development. Over the course of 50 days, learners are guided through hands-on projects and real-world applications, ensuring mastery of both foundational and advanced Python topics.

Key areas covered include:

- **Core Python:** Learning Python 3, including scripting, automation, and object-oriented programming.
- **Game Development:** Build interactive games using libraries like Turtle.
- **Web Scraping:** Extract data from websites with tools like Beautiful Soup and Selenium.
- **Data Science:** Dive into data analysis using Pandas, NumPy, Matplotlib, Seaborn, and more.
- **Web Development:** Develop both front-end and back-end applications using Flask, REST APIs, and databases like SQL, SQLite, and PostgreSQL.
- **Automation & GUI Development:** Automate tasks and build GUI desktop applications using Tkinter.
- **Deployment & Version Control:** Learn to deploy web apps and manage code with Git, GitHub, and Heroku.

The course emphasizes project-based learning, culminating in a portfolio of 50 Python projects, ranging from games to web apps, data analysis tools, and automation scripts. It provides a solid foundation for anyone looking to become a professional Python developer or enhance their programming skill set.

Starting the 50 Days of Code Challenge

Embarking on the 50 Days of Code challenge has been an exciting personal journey to enhance my Python skills. With a commitment to coding for at least one hour a day, I'm focusing on building practical projects that reinforce the core concepts of programming. Each day introduces new challenges, helping me to progressively improve my problem-solving abilities and deepen my understanding of Python. This journey is not just about writing code, but about building a strong foundation that can be applied to real-world applications.

Day 1: Band Name Generator

Concepts Learned:

- User Input: Gathering user information using the input() function.
- String Manipulation: Concatenating strings with variables to form a custom message.
- Basic Output: Displaying results with the print() function.

The project allows users to generate a band name by combining two inputs (a city and a pet's name), emphasizing user interaction through simple input and output operations.

Day 2: Tip Calculator

Concepts Learned:

- Data Types: Converting string inputs into numeric data types using float() and int().
- Basic Arithmetic: Performing calculations to determine the total bill, tip, and individual contributions.
- Formatting Output: Using round() for precision in financial calculations and f-strings to format the output message neatly.

This project introduces basic mathematical operations and string formatting, making it practical for solving everyday problems like splitting a restaurant bill with tips.

Day 3: Treasure Island Adventure

Concepts Learned:

- Conditional Logic: Implementing decision trees using if, elif, and else statements.
- Control Flow: Controlling the flow of the game based on user input and determining the game's outcome.
- String Comparison: Handling user input with case-insensitive comparisons using .lower().

This interactive game demonstrates the use of conditional statements to guide users through a simple text-based adventure, enhancing problem-solving and decision-making logic.

Day 4: Rock, Paper, Scissors Game

Concepts Learned:

- Lists: Using lists to store multiple options (rock, paper, scissors) and randomly selecting from them.
- Randomization: Leveraging the random module to generate random choices for the computer.
- User Input Validation: Ensuring the user's input is within the valid range (0 to 2) to prevent errors.
- Conditional Logic: Implementing nested if statements to determine the game's outcome (win, lose, or draw).

This project introduces the combination of randomness with conditional logic to create an interactive game. It emphasizes decision-making through comparisons and enhances the understanding of working with lists and user input validation.

Day 5: PyPassword Generator

Concepts Learned:

- Random Selection: Using random.choice() to randomly pick letters, numbers, and symbols for the password.
- Loops and Ranges: Looping through the desired number of letters, numbers, and symbols specified by the user.
- Shuffling: Leveraging random.shuffle() to randomize the final password.
- String Manipulation: Constructing the password by concatenating characters and building it dynamically.

This project challenges you to generate a random and strong password based on user input. It demonstrates the importance of randomness in password security and practices skills in loops, list manipulation, and string concatenation.

Day 6: Hurdle Jumper Game (Reeborg's World)

Concepts Learned:
Defining Functions: Custom functions help simplify repetitive actions, improving code readability and re-usability.
While Loops: Using while loops to perform actions until a condition is met..
Conditional Statements: Using if-else to check and act based on the environment

In this project, you wrote code to navigate a robot through a series of hurdles. The robot must either move forward if the path is clear or jump over obstacles if it encounters a wall. This requires combining function definitions, loops, and conditionals for problem-solving and automation.

Day 7: Hangman Game

In this project, you've built the classic Hangman game. The player must guess the hidden word letter by letter, with a limited number of wrong guesses before the game ends in a loss.

Key Components:

1. **External Modules:**
utilized hangman_words.py for a list of possible words (word_list).
also imported artwork from hangman_art.py, including the stages of the hangman drawing and the game logo.

2. **Random Word Selection:**
The game begins by selecting a random word from word_list using random.choice().

3. **Displaying the Game Progress:**
A list (placeholder) is used to track the letters guessed by the player. Underscores (_) represent unguessed letters, and correct guesses replace the underscores.

4. **Lives and Stages:**
The player starts with 6 lives. Every incorrect guess decreases the lives, and a different hangman stage is displayed (using the stages list).

5. **Game Logic:**
The game loops until the player either guesses the word correctly or runs out of lives.

6. **Each guess is validated:**
If correct, the guessed letter appears in the placeholder.
If incorrect, the player loses a life, and a new stage of the hangman is displayed.
The game ends when all letters are guessed or lives reach zero.

Day 8: Caesar Cipher Project

In this project, you've implemented a Caesar Cipher, an ancient encryption technique where each letter in the input text is shifted by a specified number of positions in the alphabet. The project allows both encryption (shifting letters forward) and decryption (shifting letters backward).

Key Components:

External Module:
The logo is imported from art.py and displayed when the program starts.

Caesar Cipher Functionality:

- Encryption: Takes the user input, shifts each letter forward by the specified number (shift), and outputs the encrypted text.
- Decryption: Reverses the process by shifting each letter backward by the specified number, revealing the original text.

Input Validation:
The program ensures that only alphabetic characters are processed. If the user enters a number, symbol, or space, a message informs them that only alphabet letters are valid for encoding/decoding.

Program Flow:
The user is prompted to choose between encrypting or decryption a message. After processing, the user is asked if they'd like to run the cipher again. The program repeats based on their response (yes to continue, no to exit).

Day 9: Secret Auction Program

This project implements a secret auction program where multiple bidders can submit their bids. The bids are stored in a dictionary, and once all participants have entered their bids, the program determines and announces the winner with the highest bid.

Key Components:

Logo Display:
The program begins by displaying the logo imported from the art.py file.

Bid Collection:
The user is prompted to enter their name and bid amount.
The input validation ensures that the bid is numeric.
The bids are stored in a dictionary with the name as the key and the bid as the value.

Handling Multiple Bidders:
After each bid, the program asks if there are more bidders. If the user answers "yes," it clears the screen and waits for the next bidder. If "no," it proceeds to find the winner.

Winner Determination:
The program identifies the highest bid and prints the name of the winning bidder along with their bid amount.

Day 10: Basic Calculator App

This project implements a basic calculator app where users can perform multiple arithmetic operations: addition, subtraction, multiplication, and division. The program supports continuous calculations using the result of the previous operation, offers a new calculation option, and allows the user to exit the app.

Key Components:

Functions for Arithmetic Operations:
The app defines separate functions for addition, subtraction, multiplication, and division:
add(), subtract(), multiply(), and divide().
Each function takes two numbers as arguments and returns the result of the operation.

Dynamic Operator Handling:
A dictionary, calculator, is used to map mathematical operators (+, -, *, /) to their respective functions. This design allows flexible operator handling.

Input Handling and Validation:
The app prompts the user to enter the first and second numbers and ensures the inputs are numeric.
The operator is also validated to ensure it is one of the supported mathematical symbols.

Loop for Continuous Operation:
The program is structured with a loop that allows the user to either:
Continue calculating with the result of the previous operation.
Start a new calculation.
Exit the calculator.

Day 11: Blackjack Game Documentation

This is a command-line implementation of the classic card game Blackjack, allowing a user to play against the computer. The game follows typical Blackjack rules where the player and the computer (dealer) are dealt two cards, and the goal is to get as close to 21 points without exceeding it. The player can request additional cards or stand, and the dealer follows basic rules to either draw or stop based on their score.

Modules and Files

1. art.py

Contains art/logo for the game.
Displays the game logo at the start of the game when the player decides to play.

2. cards

Contains a list of cards available in the game.
Card values include Aces ('A'), number cards (2 to 10), and face cards ('J', 'Q', 'K').
Aces can have values of either 1 or 11, while face cards are valued at 10.

3. function

Contains the game() function, which manages the core game logic, including:
Determining if the player or computer has a Blackjack.
Handling Ace card values (1 or 11 based on score).
Calculating player and computer scores.
Checking for win conditions and ending the game when applicable.

4. winner_function

Contains the winner() function that determines the winner after both the player and computer finish drawing cards.
Compares the player's and the computer's final scores and prints the result.

Main Game Loop

- **Initialization**

The game begins by asking the player whether they want to play a game of Blackjack ('y' or 'n').
If the player selects 'y', two cards are dealt to both the player and the computer.

- **Player's Turn**

The player's initial two cards are displayed along with the computer's first card (the second card remains hidden).
The player is asked if they want to draw another card by typing 'y' or pass by typing 'n'.
If the player draws another card and their score exceeds 21, the player loses.
If the player stands, the computer's turn begins.

- **Computer's Turn**

The computer draws cards until its score reaches at least 16 points.
If the computer's score exceeds 21, the player automatically wins.

- **End of Game**

The final scores for both the player and the computer are displayed.
The game checks for Blackjack conditions (where either the player or the computer has a Blackjack "both Ace and face card" from the start the computer wins if it has both even if player has Blackjack as well).
The winner() function is called to decide the outcome based on both players' scores.

Key Functions

1. game(computer_cards, user_cards)

This function processes the gameplay.
it Checks if any of the initial hands (player or computer) contains a Blackjack (21).
Calculates and adjusts the value of the Ace based on whether it should count as 1 or 11.
Calculates and adjusts the value of the face cards to count as 10.
Returns the user's score, computer's score, and a flag indicating if the game is over.

2. winner(user_cards, computer_cards)

This function determines the winner at the end of the game:
If the player’s score exceeds 21, they lose.
If the computer’s score exceeds 21, the player wins.
Otherwise, the scores are compared to decide the winner.
The player can choose to start a new game or exit.

3. Card Value Logic

Ace ('A'): Can be either 1 or 11. The value is automatically adjusted based on the total score (11 if the score doesn’t exceed 21, else 1).
Face Cards ('J', 'Q', 'K'): These are valued at 10.
Number Cards (2 to 10): Retain their numeric value.

Day 12 Project: Number Guessing Game Documentation

The goal was to create a number guessing game similar to the provided demo. The game should allow the user to guess a randomly chosen number between 1 and 100, while limiting the number of attempts based on the difficulty level selected by the player. There were no specific hints or detailed instructions provided, so the task was to replicate the demo using creativity and problem-solving.

Key Features:

- Random Number Generation: A random number is generated between 1 and 100 at the start of the game.
- Difficulty Levels: The user can select between 'easy' (10 attempts) and 'hard' (5 attempts).
- User Interaction: The user makes guesses, and the game responds with feedback ("Too High", "Too Low") until the correct number is guessed or the attempts run out.
- Input Validation: Input is validated to ensure it is a valid number within the specified range.

Files in the Project:

1. **Intro.py:** This module contains the ASCII art for the game logo and an introductory statement.
2. **Function.py:** This module contains the guess_game function, which implements the core logic of the game.
3. **Main Program:** This is where the game starts, providing the user with the option to start a new game, choose the difficulty level, and play the number guessing game.

Step-by-Step Explanation:

- **Generating the Logo (Intro.py)**

started by creating a logo using an ASCII art generator from the website patorjk.com, which adds a fun visual element to the game. This logo was stored in the Intro.py module.

- **Core Game Logic (Function.py)**

In Function.py, wrote the core of the number guessing game with the guess_game() function.

This function handles:

1. Generating a random number between 1 and 100.
2. Checking if the user's input is a valid guess (numeric and within range).
3. Giving feedback if the guess is too high or too low.
4. Handling the number of attempts based on the chosen difficulty level.

Key parts include:

- Random Number Generation: Using random.choice(range(1, 101)) to generate a number between 1 and 100.
- Input Validation: Checking if the user's input is a valid number using .isnumeric().
- Feedback: Providing feedback if the guess is "Too High" or "Too Low" and reducing the number of attempts accordingly.
- Game Over Condition: If the player runs out of attempts or guesses the correct number, the game ends.

- **Main Game Logic (Main Program)**

In the main part of your program, is the set up the overall flow for the game. This part allows users to start a new game, select a difficulty level, and exit the game.

Day 13 : Debugging

Day 13 focused on learning the concept of debugging, which is the process of identifying and fixing errors or bugs in code to ensure it runs as expected. It involves understanding error messages, using logical thinking to trace issues, and systematically resolving them to make the program function correctly. Debugging is essential in development to ensure code accuracy and reliability.

Day 14: The Higher Lower GAME :

the project involved creating a "Higher Lower" game where players compare follower counts of two random celebrities, with the objective of guessing which one has more followers.

The project required multiple elements:

- **Game Structure:**

The game introduces two random celebrities selected from a dataset containing their names, descriptions, countries, and follower counts in a dictionary format.
Players are prompted to choose between two options, "A" or "B," based on their guess of who has the higher follower count.

- **Data and Comparison:**

The celebrity data is stored in a separate module (game_data), containing details like their name, profession, country, and follower count.
Random selections are made from this dataset for comparison. The player's task is to correctly guess which celebrity has more followers.

- **Game Continuity:**

If the player guesses correctly, the score increases, and the game continues with a new comparison using the celebrity (A) against a new random celebrity (B).
If the player guesses incorrectly, the game ends, and the final score is displayed.

- **User Input and Validation:**

The program handles invalid inputs (anything other than 'A' or 'B') by prompting the player to choose again.
A user-friendly loop structure is used to allow continuous gameplay until the player chooses to exit or makes a wrong guess.

- **Visual Design:**

The game uses ASCII art from the art module to enhance the visual appeal. Logos and a visual separator (VS) are displayed to distinguish between the two celebrity comparisons.
This project emphasizes core programming concepts such as loops, conditionals, random selections, and user input handling, while also making the game interactive and fun for the player.

Day 15 : Coffee Machine CODE :

For Day 15, the project revolves around building a Coffee Machine Simulator. This interactive program allows users to "buy" coffee by inserting coins and selecting their desired drink. The project involves several components:

Key Features:

Menu Options:
The program presents three drink options: espresso, latte, and cappuccino, each with distinct ingredients and costs.

1. Resource Management:
The coffee machine starts with a set amount of water, milk, and coffee beans. Each time a drink is made, the machine checks if there are sufficient resources available. If not, it informs the user of the deficiency.

2. Coin Processing:
The program simulates coin insertion by prompting the user to input the number of quarters, dimes, nickels, and pennies.
It calculates whether the inserted amount is enough to cover the cost of the selected drink.
If the amount exceeds the required cost, the program returns the change to the user.

3. Transaction Flow:
If sufficient resources and payment are available, the drink is prepared, and the resources are deducted from the machine's supply.
The machine keeps track of the total money it collects from transactions.

Administrative Commands:
Users can turn the machine off by typing "off" or check the remaining resources by typing "report."

This project reinforces concepts such as modular code structure, user input handling, condition checks, loops, and resource management, providing an engaging and real-world-like simulation of a coffee vending machine.

Day 16: OOP principles :

For Day 16, the project extends the previous Coffee Machine Simulator into a more modular and refined version by utilizing object-oriented programming (OOP) principles. It incorporates classes like Menu, MenuItem, CoffeeMaker, and MoneyMachine to encapsulate functionalities.

Key Concepts:

Class Structure:

- Menu Class: Manages the available drinks and retrieves details about them.
- MenuItem Class: Represents individual drink items with specific ingredients and costs.
- CoffeeMaker Class: Handles the coffee-making process, including checking resource sufficiency and updating resources after each order.
- MoneyMachine Class: Processes payments, calculates coins inserted, and ensures enough money is provided.

Features:

Improved Code Modularity:

The code is split into smaller, more focused classes. This makes the logic cleaner, easier to maintain, and reusable.

Enhanced User Flow:

Users can select drinks from a refined menu.
The program verifies if there are sufficient resources to make the requested drink, processes the payment, and updates the machine's resource levels.

The main concept of Day 16's project was to dive deeper into Object-Oriented Programming (OOP), emphasizing how to structure and organize code for better maintainability, reusability, and clarity.

Object-Oriented Programming Concepts Highlighted:

Encapsulation:

Each class (Menu, MenuItem, CoffeeMaker, and MoneyMachine) encapsulates specific functionality. This means all related data and behavior are bundled into distinct objects, hiding the internal workings from the outside world and exposing only what's necessary (e.g., methods like is_resource_sufficient and make_payment).

Abstraction:
The complexity of processes like handling payments, checking resource availability, and making coffee is abstracted away into easy-to-understand methods within each class. For example, MoneyMachine.make_payment() handles all the coin input and payment processing, so the main logic doesn't need to worry about the details.

Modularity:
By breaking the program into separate classes, the code becomes more modular. Each class has its specific role, making the code easier to manage, test, and debug. For instance, changes to the coffee-making logic can be handled within the CoffeeMaker class without affecting how the payment system works in MoneyMachine.

Reusability:
The OOP structure encourages reusability. If you wanted to build a similar machine or expand its functionality (e.g., add more drinks), you wouldn't need to rewrite all the logic from scratch. You can extend or reuse existing classes and methods. For example, adding a new drink would only require adding it to the Menu class.

Day 17 : quiz game:

In Day 18's project, the main focus is on reinforcing the concepts of Object-Oriented Programming (OOP), specifically through creating a quiz game that handles questions and tracks the user's progress.

Key OOP Concepts in this Project:

Class Definitions:
Two primary classes are created: Question and QuizBrain.
The Question class is simple and represents individual questions, holding both the question text and the correct answer.
The QuizBrain class handles the core logic of the quiz, including asking questions, keeping track of the score, and determining if there are more questions left.

Encapsulation:
The data and behavior of each question (text and answer) are encapsulated within the Question class. This allows the QuizBrain class to interact with Question objects without worrying about the internal representation of each question.
Similarly, the QuizBrain class encapsulates all logic related to managing the quiz—tracking question numbers, user responses, and calculating scores.

Abstraction:
The QuizBrain class abstracts away the complexity of interacting with the quiz data. The main program doesn't need to know the details of how questions are stored or how the score is calculated. Instead, the program can rely on methods like next_question() and still_has_questions() to manage the quiz flow.
This means the user interacts with simple and clear methods, without having to deal with the underlying logic of question handling and answer checking.

Modularity:
The Question and QuizBrain classes are independent modules. This modularity makes the code more organized and easier to maintain. If you wanted to update how questions are managed, you could modify the Question class without affecting the quiz logic in QuizBrain.
Similarly, the quiz logic could be updated or extended without touching the way individual questions are defined or stored.

Reusability:
The project demonstrates how code can be reused by breaking it down into meaningful objects. The Question class, for instance, can be used anywhere a question/answer pair is needed, beyond just this quiz game.
The QuizBrain class could also be easily extended to handle more advanced quizzes (e.g., adding different question types, allowing multiple choice, etc.).

Control Flow:
The QuizBrain class contains the logic to move through the quiz using methods like next_question() and check_answer(). This class also controls the flow of the quiz using loops and conditionals (e.g., checking if there are more questions, updating the score based on user answers).

Data Processing:
The quiz is based on a collection of dictionaries (question_data), and the Question class is used to transform that data into objects that the quiz can use. This is a good example of how OOP can be used to process and manage data in a more structured and flexible way.

Day 18 : graphics in Python :

In Day 18's project, the focus shifts to working with graphics in Python using the turtle library and experimenting with randomness and color extraction from an image. This project adds an artistic layer to programming by using turtle graphics to generate a grid of colored dots.

Key Concepts Learned:

Using External Libraries:
The project makes use of the colorgram library to extract colors from an image. The extracted colors are then used to create a color palette (color_pal) that is used in turtle graphics.

Turtle Graphics:
Turtle is a graphical module in Python used for drawing shapes, patterns, and other visuals.

Working with Color:
After extracting a list of RGB color tuples using colorgram.extract(), the project demonstrates how to manipulate and use colors with the turtle library. The turtle color mode is set to RGB using turtle.colormode(255) to support custom RGB values.

Randomization:
random.choice() is used to randomly select a color from the extracted palette for each dot. This introduces randomness in the color pattern, creating a visually appealing effect.

Control Flow with Loops:
A for loop is used to iterate over the number of dots (number_of_dots). The turtle draws a dot, moves forward, and when it reaches the end of a row, it moves down and starts a new row. This grid-like structure is accomplished by adjusting the turtle's direction after drawing a certain number of dots.

Turtle Positioning:
The penup() method is used to move the turtle without drawing lines. After drawing each dot, the turtle moves to the next position using forward() and changes direction when the end of a row is reached.
goto() Method: The turtle is initially positioned at (-300, 300) to ensure that the drawing begins in the top-left corner of the screen.

Modularity and Reusability:
The code is written in a modular way, with configurable parameters like number_of_dots, dots_per_row, dot_size, and spacing. This allows you to easily adjust the appearance of the drawing by changing these values without altering the underlying logic.

Event-Driven Programming:
screen.exitonclick() ensures that the turtle graphics window remains open until a user clicks on it. This introduces a basic form of event-driven programming, where the program waits for user interaction before closing.

Day 19 : Event-driven programming :

In Day 19's project, the focus is on simulating a turtle race using the turtle module in Python. This project introduces interactive elements and randomness, enhancing the understanding of event-driven programming and the use of objects (Turtles) in a graphical context.

Key Concepts Covered:

Turtle Setup and Positioning:
The project involves creating multiple turtle objects, each assigned a unique color and starting position. This demonstrates how to manage multiple objects in a graphical environment.

User Interaction:
textinput() Function: This function is used to prompt the user to place a bet by selecting the color of the turtle they think will win. This adds interactivity to the program, allowing for dynamic user input and creating an engaging experience.

Creating Multiple Turtle Objects:
The Turtle class is used to create six turtle racers, each assigned a different color from the colors list. These turtles are placed at predefined y-positions to ensure they start on separate tracks.

Race Logic and Movement:
The race starts when the user places a bet.
Random distances are generated for each turtle to simulate variable speed, adding unpredictability to the race.

Checking for the Winner:
As the turtles move forward, their x-coordinate is checked to determine if they've crossed the finish line. Once a turtle crosses, the race ends, and the color of the winning turtle is compared to the user's bet. Depending on the result, a message is printed to inform the user whether they won or lost.

Event-Driven Programming:
This project reinforces event-driven programming concepts, where the race only begins once the user has placed a bet. The turtle race loop is dependent on continuous updates to the turtle's position and user input, giving a real-time feel to the game.

Day 20&21 : Snake Game :

In Day 20 and 21's project, the focus is on creating a classic Snake Game using the turtle module in Python. This project demonstrates how to manage game loops, handle user input, and work with object-oriented programming principles by breaking down the game into manageable classes: Snake, Food, and Scoreboard.

1. Snake Initialization:
- The snake is created using the `Snake` class, which initializes the snake with a few segments. The `move()` method ensures that the snake moves forward while maintaining its body structure.

2. Game Loop:
- The game loop continuously updates the screen and checks for interactions (eating food, colliding with walls, or hitting itself). If a condition ends the game, such as the snake hitting a wall, the game stops, and the score is displayed.

3. Object-Oriented Design:
- The game is built using three key classes (`Snake`, `Food`, `Scoreboard`), making the code modular and easy to manage. Each class has its own responsibility, and they interact with each other through method calls.

- Level Progression: Increase the speed of the game as the snake gets longer or with each score increase.
- Power-ups: Introduce random food items that grant temporary speed boosts or slow down the game.
- Obstacle Addition: screen four walls are obstacles that the snake must avoid.
- High Score Tracking: Keep track of the highest score across multiple game sessions.

Day 22 : Pang Game :

This Pong game is a simple yet engaging implementation of the classic arcade game using Python's turtle module. The game features two paddles, controlled by two players, and a ball that bounces between them. The objective is to prevent the ball from passing your paddle while trying to score points by making the ball go past your opponent's paddle.

Two Player Controls: Players can control their paddles using the keyboard. The right paddle is controlled using the "Up" and "Down" arrow keys, while the left paddle uses the "W" and "S" keys for upward and downward movement, respectively.

Dynamic Scoring: The game tracks the scores of both players in real-time. Points are awarded when the opposing player fails to return the ball.

Ball Movement: The ball moves continuously across the screen and bounces off the top and bottom walls, as well as the paddles. The ball resets to the center after a score is made.

Collision Detection: The game includes simple collision detection logic that allows the ball to bounce off the paddles and walls, enhancing the gameplay experience.

Components

Main Game Logic: This component initializes the game screen, sets up paddles, and controls the main game loop where the ball moves, scores are updated, and user inputs are processed.

Paddle Class: Defines the paddles' properties and movement. Each paddle is created at specific positions on the left and right sides of the screen and can move up and down within set boundaries.

Ball Class: Manages the ball's movement and bouncing mechanics. It controls how the ball interacts with the paddles and walls, as well as its position resets when a player scores.

Score Class: Keeps track of the players' scores and updates the displayed score on the screen.

Day 23 : The Turtle Crossing Game :

The Turtle Crossing Game is a Python-based project that simulates a simplified version of the classic "Frogger" game, where the player controls a turtle trying to cross a road filled with moving cars. The goal is to reach the top of the screen while avoiding collisions with the cars, which increase in speed with each successful crossing. The game is built using Python's turtle module, and it includes player movement, dynamically generated cars, and a scoring system that tracks the player's progress.

Key Features

Player Control: The player controls a turtle that moves up the screen. Each time the player reaches the top (finish line), they return to the starting position and the level increases, making the game progressively more challenging.

Car Movement: Cars are randomly generated and move horizontally across the screen. The speed of the cars increases with each level, providing a dynamic challenge for the player.

Collision Detection: The game detects when the turtle collides with a car. If the turtle is too close to a car, the game ends, and a "Game Over" message is displayed.

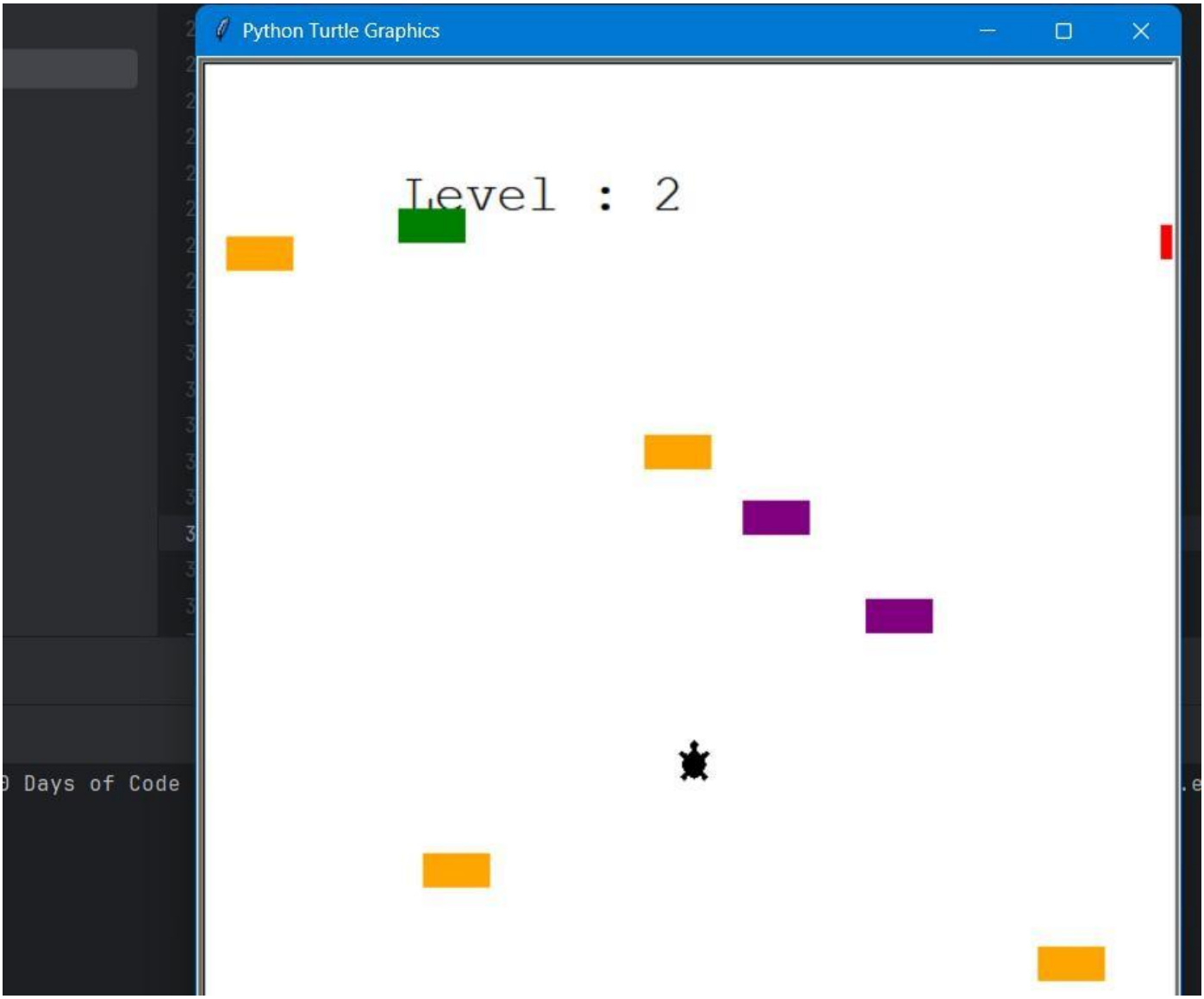
Scoring System: The scoreboard tracks the player's progress by displaying the current level. Each time the player reaches the finish line, they level up, and the cars speed up accordingly.

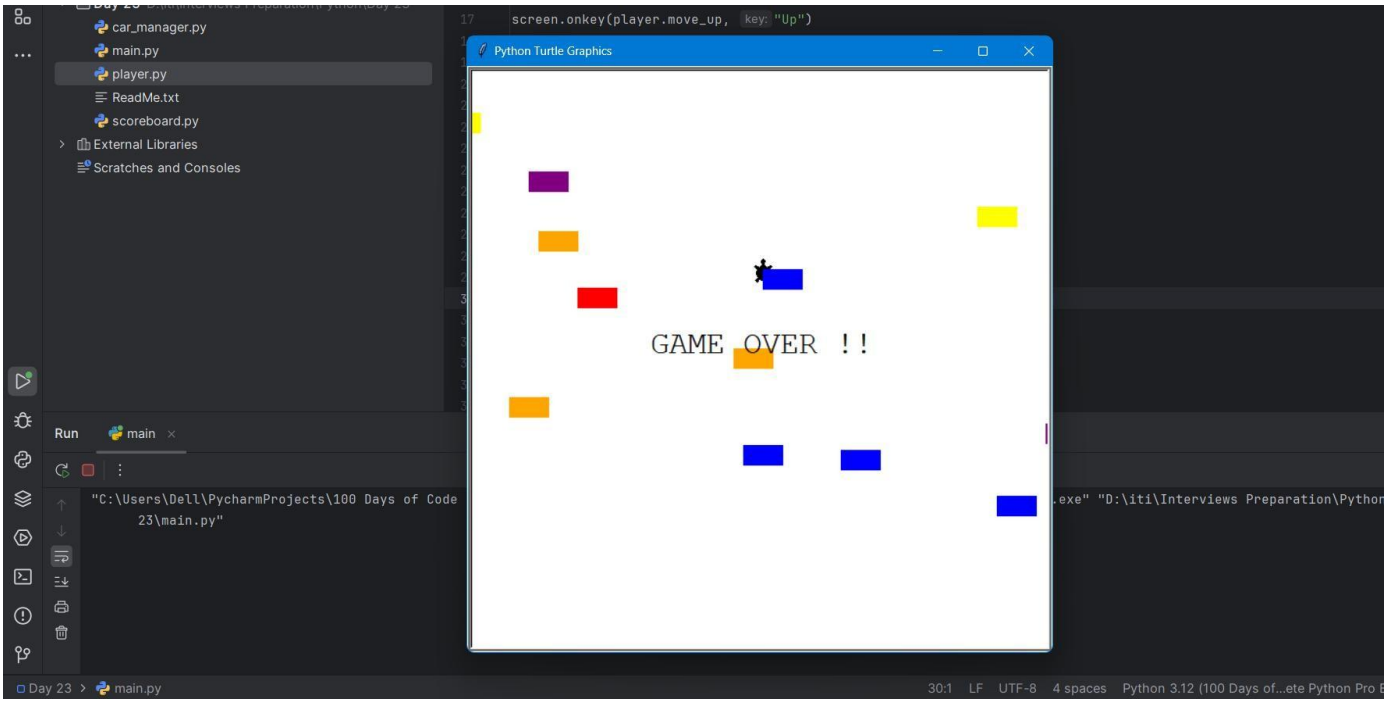
Structure

Player Class: Manages the turtle's movement and tracks whether the turtle has reached the finish line. It allows the player to move the turtle upward using keyboard inputs.

CarManager Class: Responsible for generating, moving, and removing cars from the screen. It manages the cars' speed and increases the difficulty by speeding up the cars after each successful level.

Scoreboard Class: Tracks the player's progress and displays the current level on the screen. It also handles the display of the "Game Over" message when the player collides with a car.


Python Turtle Graphics
Level : 2
Days of Code


car_manager.py
main.py
player.py
ReadMe.txt
scoreboard.py
External Libraries
Scratches and Consoles
screen.onkey(player.move_up, key: "Up")
Python Turtle Graphics
GAME OVER !!
Run
main
"C:\Users\Dell\PycharmProjects\100 Days of Code 23\main.py"
Day 23
main.py
30:1
LF
UTF-8
4 spaces
Python 3.12 (100 Days of...ete Python Pro

Day 24 : Automated Letter Generator :

The Automated Letter Generator is a Python project that simplifies the process of creating personalized letters for multiple recipients. By utilizing a template letter and a list of names, the program generates individual letters tailored for each person. This project demonstrates how to handle file input and output, string manipulation, and basic automation to streamline repetitive tasks.

Key Features

Template Letter: The program reads a template letter from a text file. This letter contains a placeholder for the recipient's name, which allows for easy personalization.

Dynamic Name Insertion: It reads a list of names from a separate text file and replaces the placeholder in the template letter with each name, creating a customized letter for every individual.

Output Management: The generated letters are saved in a designated output folder, with each file named according to the recipient. This organization makes it easy to manage and distribute the letters.

Through the development of this project, I gained the following insights and skills:

File Handling in Python: learned how to efficiently read from and write to text files, including managing file paths and ensuring proper file formatting.

String Manipulation: The project reinforced my understanding of string operations, including replacement of placeholders and trimming whitespace from user input.

Organizing Output: structuring output files in a clear and logical manner, making it easier to access and share generated letters.

Day 25 : U.S. States Game :

This project is a Python-based interactive game where users attempt to name all 50 U.S. states. The game uses a graphical interface powered by the Turtle module, along with data handling through pandas. The objective is for the user to correctly guess all the states by entering their names. The program provides visual feedback by displaying the name of each correctly guessed state on a U.S. map. If the user types "exit," the game ends, and a file is generated that lists the states that were not guessed.

Key Features

Graphical Interface:
The game displays a blank U.S. map using the Turtle module, creating an engaging and interactive environment for users.
Each correct guess is marked and written on the map by writing the state's name at its corresponding location.

User Input and Tracking:
The game continuously prompts users to input state names. Correct guesses are tracked, and users can see their progress in real time.
To end the game prematurely, the user can type "exit." At this point, the game saves the remaining unguessed states to a CSV file for future reference.

State Positioning:
Each state's coordinates (x, y) are preloaded from a CSV file containing state names and their respective coordinates on the map. When a state is guessed correctly, its name is drawn at the appropriate location.

State Validation:
The program ensures that state names are not duplicated by keeping track of previously guessed states.
It also handles invalid or incorrect guesses by informing the user without affecting the flow of the game.

Remaining States CSV:
If the player chooses to exit the game, the program generates a CSV file listing all the states that were not guessed, allowing the user to review the missed states.

Pandas for Data Manipulation: I utilized pandas to manage state data, perform checks, and handle data export, reinforcing my data manipulation skills in Python.

File Handling and Input: learned how to read data from a CSV file, work with structured data, and save outputs (in this case, the remaining unguessed states) back to a file.