

PROGRAMING LANGUAGES (CPE350) ASSIGMENT

Apple Nappy

64220064 - Zeynep GENEL

64210032 - Neva AYDIN

64210013 - Gül AKKOCA

64220060 - Kader Tuğba YARAMIŞ

64220017 - Berat NEVCANOĞLU

[**Introduction** 3](#_Toc189002436)

[**Project Overview** 3](#_Toc189002437)

[**Python’s History** 3](#_Toc189002438)

[**Comparison and Evaluation on Python Go and Kotlin Languages** 4](#_Toc189002439)

[**Syntax Differences** 4](#_Toc189002440)

[**Performance** 4](#_Toc189002441)

[**Community Support, Learning Resources, Documentation** 6](#_Toc189002442)

[**Applicability** 7](#_Toc189002443)

[**Strong and Weak Aspects of the Language** 9](#_Toc189002444)

[**Strong Aspects** 9](#_Toc189002445)

[**Weak Aspects** 10](#_Toc189002446)

[**Project Codes** 12](#_Toc189002447)

[1.Creating the Game Screen 12](#_Toc189002448)

[2.Game State Variables 12](#_Toc189002449)

[3.Pause Button Implementation 12](#_Toc189002450)

[4. Restarting the Game 13](#_Toc189002451)

[5. Game Over Handling 13](#_Toc189002452)

[6. Displaying the High Score 14](#_Toc189002453)

[7. CountDown Implementation 15](#_Toc189002454)

[8. Main Menu Implementation 16](#_Toc189002455)

[9. Game Flow Management 18](#_Toc189002456)

[10. Pause Functionality 18](#_Toc189002457)

[11. Quitting the Game 20](#_Toc189002458)

[12. Main Game Loop 20](#_Toc189002459)

[**Conclusion** 25](#_Toc189002460)

[References 26](#_Toc189002461)

# **Introduction**

In this report, we will provide an overview of the group project we completed using Python. The project involved designing and implementing a game with various features such as pausing, game-over handling, and score tracking. Our objective was to create an engaging and user-friendly game while maintaining efficient code practices and a structured development process.

# **Project Overview**

Our game features a dynamic interface where users can start, pause, and resume gameplay, with a high score tracking system to encourage player engagement. Throughout the development process, we focused on organizing the game logic efficiently and ensuring seamless interaction between different components.

# **Python’s History**

The [programming language](https://en.wikipedia.org/wiki/Programming_language) [Python](https://en.wikipedia.org/wiki/Python_(programming_language)) was conceived in the late 1980s, and its implementation was started in December 1989by [Guido van Rossum](https://en.wikipedia.org/wiki/Guido_van_Rossum) at [CWI](https://en.wikipedia.org/wiki/Centrum_Wiskunde_%26_Informatica) in [the Netherlands](https://en.wikipedia.org/wiki/The_Netherlands) as a successor to [ABC](https://en.wikipedia.org/wiki/ABC_(programming_language)) capable of [exception handling](https://en.wikipedia.org/wiki/Exception_handling) and interfacing with the [Amoeba operating system](https://en.wikipedia.org/wiki/Amoeba_(operating_system)). Van Rossum is Python's principal author, and his continuing central role in deciding the direction of Python is reflected in the title given to him by the Python community, [*Benevolent Dictator for Life* (BDFL)](https://en.wikipedia.org/wiki/Benevolent_dictator_for_life). (However, Van Rossum stepped down as leader on July 12, 2018.). Python was named after the [BBC TV](https://en.wikipedia.org/wiki/BBC_TV) show [*Monty Python's Flying Circus*](https://en.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus).

Python 2.0 was released on October 16, 2000, with many major new features, such as [list comprehensions](https://en.wikipedia.org/wiki/List_comprehension), [cycle-detecting](https://en.wikipedia.org/wiki/Cycle_detection) [garbage collector](https://en.wikipedia.org/wiki/Garbage_collection_(computer_science)) (in addition to [reference counting](https://en.wikipedia.org/wiki/Reference_counting)) and [reference counting](https://en.wikipedia.org/wiki/Reference_counting), for [memory management](https://en.wikipedia.org/wiki/Memory_management) and support for [Unicode](https://en.wikipedia.org/wiki/Unicode), along with a change to the development process itself, with a shift to a more transparent and community-backed process.

Python 3.0, a major, backwards-incompatible release, was released on December 3, 2008,after a long period of testing. Many of its major features have also been [backported](https://en.wikipedia.org/wiki/Backport) to the backwards-compatible, though now-unsupported, Python 2.6 and 2.7. Releases of Python 3 include the 2to3 utility, which automates the translation of Python 2 code to Python 3.

### **Key Features of the Game**

**Falling Objects (Fruits and Skulls)**

* + Apples and golden apples fall from the top.
  + Skulls act as obstacles that reduce lives when collected.
  + Objects spawn randomly or in groups.

**Player Controls (Cart Movement)**

* + The player controls a cart using the left and right keys.
  + The cart catches apples to gain points and must avoid skulls.

**Game Scoring System**

* + Collecting an apple increases the score by 1
  + Collecting a golden apple increases the score by 2
  + If a fruit falls off the screen, the player loses a life.

**Lives and Game Over Mechanism**

* + The player starts with 3 lives.
  + If a skull is collected or a fruit falls, a life is lost.
  + The game ends when lives reach 0, displaying a game-over screen with the high score.

**User Interface Elements**

* + **Score Display:** The current score is shown at the top left.
  + **High Score Tracking:** Displays the highest score achieved in the session.
  + **Lives Display:** Hearts indicate the number of remaining lives.
  + **Pause and Resume Buttons:** The player can pause/resume the game using a button.
  + **Game Menu:** Includes options for "Start Game," "Resume," and "Quit."

**Sound Effects and Background Music**

* + Various sounds for different actions (collecting fruits, losing a life, game over).
  + A countdown sound effect plays before the game starts.

**Animated Countdown at Game Start**

* + An animated countdown is displayed before the game begins.

**Implementation Details**

The game Apple Nappy is made using the PyGame library, which handles graphics, sounds, and player controls. It starts with a menu where the player can choose to start, resume, or quit. When the game begins, a countdown appears before the player takes control. The player moves a cart at the bottom of the screen using the left and right keys to catch falling objects. These objects include apples, golden apples, and skulls, which appear randomly or in small groups. Catching apples increases the score, while missing fruits or catching skulls reduces lives. The game keeps track of the score and high score, displaying them on the screen along with heart icons that show the number of lives left. If the player loses all lives, a game over screen appears with an option to restart. There is also a pause function that allows the player to stop the game and return to the menu. The way objects appear is random, making the game different each time it is played.

# **Comparison and Evaluation on Python Go and Kotlin Languages**

Programming languages like Python, Go, and Kotlin are powerful tools that offer distinct advantages for different needs and projects. Comparing these languages in terms of syntax, performance, community support, and applicability can simplify the process of making the right language choice.

## **Syntax Differences**

***Python***, being a high-level language, allows for increased efficiency with less code.

* For operations such as loops, string manipulation, and animations, Python offers a simpler and more concise syntax compared to Go.
* In Python, using libraries is easy (e.g., pygame). We can directly call the functions of the pygame library without the need to define an external function or structure.
* Thanks to its dynamic typing and minimal coding features, the coding process is straightforward.

***Go*** requires a more rigid syntax and more boilerplate code. Writing the same countdown in Go requires more structural code because:

* Go’s type system is stricter; you need to explicitly specify the types of variables.
* The strict type system and built-in tools ensure consistency in writing.

***Kotlin*** has a static type system, which increases code reliability by catching errors at compile time.

* The syntax is quite clean and compact, and it is particularly powerful with lambda expressions and higher-order functions.

## **Performance**

**Python**

* + Python is excellent for rapid prototyping, but its performance is limited as it is an interpreted language.
  + During game loops or animations, Python's low performance may become noticeable.
  + It is a slow language, and its **garbage collector** can create bottlenecks in high-performance real-time applications.
  + It is less preferred on mobile platforms due to optimization requirements.

**Go**

* + Go's compiled nature ensures high performance. However, its graphics and gaming libraries are less advanced compared to Python.

***Multithreading***

* + Go's goroutines feature is highly useful for parallelizing game loops with asynchronous operations.

***Garbage Collector***

* + Go’s garbage collector provides automatic memory management and is much better optimized for performance compared to Python.

**Kotlin**

* + **Advantages:**
    - Kotlin runs on the JVM, providing much better performance compared to Python.
    - You can achieve full hardware performance on Android during game development.
    - Kotlin is compatible with powerful game engines like LibGDX, thanks to its Java interoperability.
  + **Disadvantages:**
    - It is harder to develop prototypes as quickly as in Python.
    - The JVM garbage collector can sometimes cause latency issues.

## **Community Support, Learning Resources, Documentation**

**Python**

**Community Support:**

* + - Python, released in 1991, has a vast community.
    - Platforms like Stack Overflow, GitHub, and Reddit host numerous Python-related questions and answers.
    - The Python community includes users capable of addressing both beginner and advanced problems.
  + **Learning Resources:**

Python has abundant books, courses, blogs, and YouTube videos.

* + - There are extensive resources specifically for game development, data science, and artificial intelligence.
    - Libraries like Pygame have rich educational materials for game development.
  + **Documentation:**
    - Python’s documentation is comprehensive and easy to understand.
    - Library documentation includes examples, making it easier to learn and solve problems.
* **Go (Golang)**
  + **Community Support:**
    - Go, developed by Google in 2009, is a relatively new language.
    - While its community is not as large as Python’s, it is strong in backend development, microservices, and concurrency.
    - However, its support for niche areas like game development is limited compared to Python.
  + **Learning Resources:**
    - Go offers official documentation, books, and a few good online resources. However, it has a narrower range compared to Python.
    - Game development-specific resources using libraries like Ebiten and raylib-go are limited, making it slightly challenging for beginners.
  + **Documentation:**
    - Go’s official documentation is of high quality, with examples and clear explanations suitable for beginners and advanced users.
    - However, community-contributed examples are not as extensive as Python’s.
* **Kotlin**
  + **Community Support:**
    - Kotlin’s community is growing rapidly, especially in Android development.
    - However, it lacks the breadth of community support for game development compared to Python.
  + **Learning Resources:**
    - Kotlin has high-quality learning resources but fewer options specifically for game development.
    - Android-focused game development resources are more accessible.
  + **Documentation:**
    - Kotlin’s official documentation is excellent, clear, and supported by examples.
    - However, documentation for libraries like LibGDX may require more technical knowledge.

## **Applicability**

**Game Example: Apple Nappy** (A game where fruits fall, are caught, and have a life-loss and scoring system)

* **Python**
  + **Suitability:**
    - Python is very suitable for game development, especially for simple 2D games.
    - The Pygame library supports core game development aspects like game loops, animations, score management, and collision detection.
    - Prototypes can be created quickly, and animations are easy to implement.
  + **Limitations:**
    - Python struggles with performance for large or complex games.
    - Games built with Python often perform worse and are less suitable for large-scale projects compared to languages like C++ or Go.
* **Go (Golang)**
  + **Suitability:**
    - While Go is not specifically designed for game development, libraries like Ebiten or raylib-go can be used to create games.
    - Lightweight game loops and concurrency make animation and game mechanics efficient.
    - Its compiled nature provides better performance compared to Python.
  + **Limitations:**
    - Go's community and learning resources for game development are limited.
    - It lacks the rich ecosystem of visual effects or complex physics engines found in Python.
    - For beginners, the lack of comprehensive library support may be challenging.
* **Kotlin**
  + **Suitability:**
    - Kotlin is excellent for Android games. It integrates well with Java-based game engines like LibGDX.
    - Kotlin’s static type system and modern tools allow for reliable game loops and animations.
    - Games built with Kotlin perform much better on mobile devices compared to Python.
  + **Limitations:**
    - Kotlin does not allow rapid prototyping as quickly as Python.
    - Libraries like LibGDX require more technical knowledge to learn.

# **Strong and Weak Aspects of the Language**

## **Strong Aspects**

1. **Ease of Use and Readability**

Python’s syntax is simple, clean, and close to natural language, making it beginner-friendly and easy to read and maintain.

1. **Large Standard Library**

Python has a vast standard library that supports a wide range of tasks, from web development (e.g., Flask, Django) to data analysis (e.g., Pandas, NumPy) and machine learning (e.g., TensorFlow, PyTorch).

1. **Cross-Platform Compatibility**

Python is platform-independent, allowing the same code to run on Windows, macOS, and Linux without modification.

1. **Diverse Ecosystem and Libraries**

Python boasts extensive third-party libraries for almost every domain, such as data science, web development, AI, and automation.

1. **Community Support**

Python has a large and active community, making it easier to find resources, tutorials, and solutions to problems.

1. **Interpreted Language**

Being interpreted means no compilation step is needed, making debugging and testing faster.

1. **Versatility**

Python supports multiple programming paradigms: object-oriented, procedural, and functional programming.

1. **Ideal for Rapid Prototyping**

Python’s simplicity and quick development cycle make it ideal for prototyping and iterative development.

1. **Integration Capabilities**

Python can integrate easily with other languages like C, C++, and Java, and supports frameworks like Jython and CPython.

1. **Extensive Use in Data Science and AI**

Python dominates the fields of AI, machine learning, and data science due to libraries like Scikit-learn, Pandas, and Matplotlib.

## **Weak Aspects**

1. **Performance**

Python is slower than compiled languages like C++ or Java because it’s an interpreted language. This can be a bottleneck for high-performance applications.

1. **Global Interpreter Lock (GIL)**

Python's GIL can restrict true multi-threading, making it less efficient for CPU-bound tasks in multi-threaded environments.

1. **Weak in Mobile Development**

While Python is versatile, it is not a popular choice for mobile app development compared to languages like Kotlin or Swift.

1. **Memory Consumption**

Python uses more memory compared to lower-level languages, making it less suitable for memory-constrained systems.

1. **Runtime Errors**

Since Python is dynamically typed, errors (e.g., type errors) often surface only at runtime, which can lead to delayed debugging.

1. **Not Ideal for Low-Level Programming**

Python lacks direct access to hardware and low-level system calls, making it unsuitable for tasks like OS development or embedded systems.

1. **Dependency Management**

Managing dependencies in Python (e.g., conflicts between libraries) can be tricky, though tools like virtual environments help mitigate this.

1. **Scalability Concerns**

Python's interpreted nature and GIL make it less scalable for highly concurrent, large-scale systems compared to languages like Java or Go.

1. **Version Confusion**

Differences between Python 2 and Python 3 caused fragmentation in the past, though Python 2 has officially reached end-of-life.

1. **Error-Prone Dynamic Typing**

While dynamic typing makes Python flexible, it can lead to bugs that are hard to detect until runtime.

# **Project Codes**

## 1.Creating the Game Screen

The game screen is created with a resolution of 800x600 pixels.

The title of the game is set as "apple nappy".

# Screen dimensions

screenWidth = 800

screenHeight = 600

# Screen setup

gameDisplay = pygame.display.set\_mode((screenWidth, screenHeight))

pygame.display.set\_caption('Apple Nappy')

## 2.Game State Variables

To maintain the overall state of the game, we defined several key variables:

# Game state variables

is\_failed = False # Whether the game is over

is\_paused = False # Tracks whether the game is paused

max\_score = 0 # Player's high score

These variables allow us to track and control the flow of the game efficiently. The **is\_failed** variable determines if the game has ended, while **is\_paused** helps manage interruptions during gameplay. The **max\_score** variable records the highest achieved score.

## 3.Pause Button Implementation

To provide the player with control over gameplay, we implemented a pause button. The **draw\_pause\_button** function was created to manage the visual representation and functionality of the pause feature.

# Function to draw the pause button

def draw\_pause\_button():

button\_width, button\_height = 100, 40

button\_x = screenWidth - 150

button\_y = 60

draw\_button("Pause", button\_x, button\_y, button\_width, button\_height, pink, black, pause\_game)

We utilized the existing **draw\_button** function to avoid code redundancy and ensure maintainability. The pause button allows the user to access a menu with options such as "Quit," "Restart," "Resume," and "Go to Menu."

## 4. Restarting the Game

To allow players to restart the game after it ends, we implemented the **restart\_game** function. This function resets all the essential game variables to their initial states and ensures the game starts fresh. It also includes a call to the countdown function to reintroduce the player to the gameplay.

def restart\_game():

global is\_failed, score, circles, plate\_x, lives, gameover\_sound\_played

is\_failed = False

score = 0

lives = 3

circles = []

plate\_x = (screenWidth - plate\_width) // 2

gameover\_sound\_played = False

countdown()

## 5. Game Over Handling

The **game\_over** function was designed to manage the end-of-game scenario and update the high score if the player surpasses their previous record.

# Function to handle game over

def game\_over():

global max\_score, gameover\_sound\_played

# Update high score

if score > max\_score:

max\_score = score

# Display "Game Over" text

font = pygame.font.Font(main\_font, 55)

text = font.render("Game Over!", True, black)

text\_rect = text.get\_rect(center=(screenWidth // 2, screenHeight // 2 - 50))

gameDisplay.blit(text, text\_rect)

# Display high score

font = pygame.font.Font(main\_font, 40)

high\_score\_text = font.render(f"High Score: {max\_score}", True, black)

high\_score\_rect = high\_score\_text.get\_rect(center=(screenWidth // 2, screenHeight // 2))

gameDisplay.blit(high\_score\_text, high\_score\_rect)

We added the global variable **max\_score** to keep track of the highest score achieved in the game. Initially, we set it to zero to ensure a clean start. During gameplay, we check if the current score exceeds the recorded **max\_score**, and if so, we update it accordingly. At the end of the game, the high score is displayed to inform the player of their best performance.

To achieve this, we used the f**ont.render(...)** function from the **Pygame** library, which renders text into a **pygame.Surface** object. The **pygame.display.blit()** function is then used to draw this surface onto the main game window. This function plays a crucial role in visually presenting game elements by copying images, text, or shapes onto the screen.

Additionally, the **get\_rect()** function is used to generate a **pygame.Rect** object that defines the dimensions and position of the high score text. Essentially, this method creates a rectangle that precisely encloses the high score text and centers it on the game screen, ensuring a well-aligned and visually appealing layout.

## 6. Displaying the High Score

To provide players with feedback on their performance, we implemented the **show\_high\_score** function, which displays the high score below the current score.

# Function to show high score below the current score

def show\_high\_score():

font = pygame.font.Font(main\_font, 30)

text = font.render(f"High Score: {max\_score}", True, black)

gameDisplay.blit(text, (10, 50))

This function uses **pygame.font.Font** to define the text style and **gameDisplay.blit()** to place it on the screen.

## 7. CountDown Implementation

#function to countdown

def countdown():

# Play the countdown sound

countdown\_sound.play()

for i in range(3, 0, -1): # Counts down from 3 to 1

for size in range(50, 150, 5): # Increases the font size (from 50 to 150)

gameDisplay.blit(background\_image, (0, 0)) # Draws the background

countdown\_font = pygame.font.Font(cd\_font, size)

text = countdown\_font.render(str(i), True, black) # Renders the countdown number

text\_rect = text.get\_rect(center=(screenWidth // 2, screenHeight // 2)) # Centers the text

gameDisplay.blit(text, text\_rect) # Draws the text on the screen

pygame.display.update() # Updates the display

pygame.time.delay(10) # Creates animation with a small delay

pygame.time.delay(500) # Keeps the number on the screen for a short duration

Our goal in this code is to add a countdown animation before the start of the game. Once the countdown ends, it displays a "GO!" message on the screen.

As each number is displayed on the screen, the font size increases.The range(50, 150, 5) loop increases the font size by 5 units from 50 to 150. This creates a growing animation effect

# "Go!" animation

for size in range(50, 150, 5): # Increases the font size for "Go!"

gameDisplay.blit(background\_image, (0, 0)) # Draws the background

countdown\_font = pygame.font.Font(cd\_font, size)

text = countdown\_font.render("Go!", True, black) # Renders the "Go!" text

text\_rect = text.get\_rect(center=(screenWidth // 2, screenHeight // 2)) # Centers the text

gameDisplay.blit(text, text\_rect) # Draws the text on the screen

pygame.display.update() # Updates the display

pygame.time.delay(10) # Creates animation with a small delay

pygame.time.delay(500) # Keeps "Go!" on the screen for a short duration

pygame.time.delay(10): A 10-millisecond delay is added each time the text size increases.

pygame.time.delay(500): Each number remains on the screen for a short duration (0.5 seconds).

After the countdown is complete, the screen displays the word “Go!”

The word “Go!” is also displayed with a growing animation effect.

The "Go!" text stays on the screen for half a second (0.5 seconds).

**Key Points of Our Code**

1. **Animation:** We create visual dynamism by gradually increasing the text size, making the countdown more engaging.
2. **Delays**: We add small delays at each step to ensure the animation appears smooth and visually appealing.
3. **Redrawing:** We redraw the background and text in every iteration to clear previous visuals and maintain a clean display.
4. **Flexibility:** We can easily adjust the countdown duration, text size, or delay times to suit the game's needs.

## 8. Main Menu Implementation

The **show\_menu** function was created to provide players with an intuitive interface at the start of the game. The menu allows users to start a new game, resume a paused game, or quit.

# Function to display the main menu

def show\_menu():

global menu\_running

is\_paused = False

menu\_running = True

while menu\_running:

gameDisplay.blit(background\_image, (0, 0)) # Display background

# Display game title

font = pygame.font.Font(title\_font, 67)

title\_surface = font.render("Apple Nappy", True, black)

title\_rect = title\_surface.get\_rect(center=(screenWidth // 2, screenHeight // 4))

gameDisplay.blit(title\_surface, title\_rect)

# Draw menu buttons

button\_width, button\_height = 200, 60

button\_x = screenWidth // 2 - button\_width // 2

start\_button\_y = screenHeight // 2 - 40

resume\_button\_y = screenHeight // 2 + 40

quit\_button\_y = screenHeight // 2 + 120

draw\_button("Start Game", button\_x, start\_button\_y, button\_width, button\_height, pink, black, start\_game)

# Show "Resume" button only if the game is paused

if is\_paused:

draw\_button("Resume", button\_x, resume\_button\_y, button\_width, button\_height, pink, black, resume\_game)

draw\_button("Quit", button\_x, quit\_button\_y, button\_width, button\_height, pink, black, quit\_game)

for event in pygame.event.get():

if event.type == pygame.QUIT: # Exit game if the close button is pressed

pygame.quit()

sys.exit()

pygame.display.update()

clock.tick(FPS)

First, the **is\_paused** variable is set to False, and to keep the loop running, the **menu\_running** variable is set to True. Since the code is inside a while loop, it continuously displays the background, game title, and menu buttons such as "Resume" and "Quit."

After rendering the menu, the code checks for user interactions using the **pygame.event.get()** function. This function retrieves a list of all events that have occurred since the last time it was called. In **Pygame**, events represent user interactions such as mouse clicks and key presses, as well as system events like window closing.

The function returns a list of **pygame.event** objects, with each object representing a single event that has occurred. For example, **pygame.MOUSEBUTTONDOWN** indicates that a mouse button was pressed.

If the function detects a quit event, the game will exit using the **sys.exit()** command, ensuring a clean termination of the program.

## 9. Game Flow Management

To ensure proper game flow, we implemented the **start\_game** function, which initializes essential variables and prepares the game environment.

# Function to start the game

def start\_game():

countdown()

global running, menu\_running, max\_score, is\_paused

max\_score = 0 # Reset high score

running = True # Enter game loop

menu\_running = False # Exit menu loop

is\_paused = False # Ensure game isn't paused

This function resets critical values to guarantee that the game starts fresh each time. I added the values like **is\_paused** , **menu\_runngin** , **max\_score** to be ensure that these functions and values are work correct .

## 10. Pause Functionality

To handle interruptions, the **pause\_game** function was developed to freeze the game and present options to the player.

# Function to pause the game

def pause\_game():

global is\_paused

is\_paused = True

while is\_paused:

gameDisplay.blit(background\_image, (0, 0)) # Draw background

# Display "Paused" text

font = pygame.font.Font(main\_font, 70)

pause\_text = font.render("Paused", True, black)

pause\_text\_rect = pause\_text.get\_rect(center=(screenWidth // 2, screenHeight // 2 - 100))

gameDisplay.blit(pause\_text, pause\_text\_rect)

# Draw pause menu buttons

button\_width, button\_height = 200, 60

button\_x = screenWidth // 2 - button\_width // 2

resume\_button\_y = screenHeight // 2

menu\_button\_y = screenHeight // 2 + 80

draw\_button("Resume", button\_x, resume\_button\_y, button\_width, button\_height, pink, black, resume\_game)

draw\_button("Go to Menu", button\_x, menu\_button\_y, button\_width, button\_height, pink, black, show\_menu)

for event in pygame.event.get():

if event.type == pygame.QUIT: # Exit game if the close button is pressed

pygame.quit()

sys.exit()

pygame.display.update()

clock.tick(FPS)

Here is my **is\_paused** function. The first step is to update the **is\_paused** variable to True, indicating that the game is paused. The function then enters a while loop, which continues running as long as the **is\_paused** condition remains True.

Within the loop, the background, pause text, and buttons such as "Resume" are displayed on the screen.

To handle user interactions, the **pygame.event.get()** function is used to retrieve a list of events that have occurred. The function iterates through these events, and if a quit operation is detected, the game exits. The pause menu allows the player to either resume or return to the main menu, ensuring flexibility during gameplay.

## 11. Quitting the Game

A simple quit function was implemented to safely close the game and terminate the program execution.

# Function to quit the game

def quit\_game():

pygame.quit()

sys.exit()

## 12. Main Game Loop

The core of the game lies in the main game loop, which continuously updates the screen, checks for events, and manages game states.

# Display the menu at the start

show\_menu()

# Main game loop

running = True

while running:

gameDisplay.blit(background\_image, (0, 0)) # Draw background

circles.append((x, y, "white", is\_fruit)) # Meyveyi listeye ekle

for event in pygame.event.get():

if event.type == pygame.QUIT: # Exit the game if the close button is pressed

running = False

elif is\_failed: # If game is over, display game over screen

game\_over()

if not is\_failed:

draw\_pause\_button()

# Draw game elements

show\_high\_score()

pygame.display.update()

clock.tick(FPS)

# Quit pygame

pygame.quit()

sys.exit()

The loop efficiently handles game rendering and event processing while maintaining a consistent frame rate.

Within the loop, it checks if the **is\_failed** variable is True. If so, the **game\_over()** function is called to display the game over screen. Otherwise, the **draw\_pause\_button()** function is executed, which displays a pause button on the screen. This button allows the player to pause the game and access options such as "Quit," "Restart," "Resume," and "Go to Menu."

Additionally, the **show\_high\_score()** function is called to display the highest score achieved in the game. Finally, **pygame.display.update()** ensures that all graphical changes are rendered on the screen, and **clock.tick(FPS)** maintains a steady frame rate.

**Add a new fruit (or object) to the screen randomly**

# Ensure there are at most 5 objects on the screen

if len(circles) == 1 and random.randint(1, 180) == 1:

is\_fruit = random.choice(["apple", "grape", "skull"]) # Randomly choose an object type

margin = 200

x = random.randint(margin + circle\_radius, screenWidth - margin - circle\_radius) # Random x position within margin

y = -circle\_radius # Start above the screen

circles.append((x, y, "white", is\_fruit)) # Add the object to the list

The main purpose of this part of the code is to add a new fruit when there is only one object on the screen, and we do this randomly.

**Random Chance:** We checks with random.randint(1, 180) == 1 to decide if a new object should spawn (introducing randomness)

**Object Type:** A random object type is selected (apple, grape, or skull).

**Positioning:** The object is given a random x position within the screen's horizontal bounds, leaving a margin to avoid the edges.

**Start Position:** The object starts just above the screen (y=-circle\_radius) ready to fall down.

**Adding to List:** The object is appended to the circles list, so it can be tracked and rendered during the game.

# Groups

if len(circles) < 2 and random.randint(1, 180) == 1:

group\_x = random.randint(100, screenWidth - 100) # Center of the group

initial\_y = -circle\_radius # Starting height of the first fruit

for i in range(random.randint(1, 2)): # Drop 1-2 fruits at the same time

while True: # Loop until a suitable position is found

x = random.randint(max(0, group\_x - 50), min(screenWidth, group\_x + 50)) # Position within the group

y = initial\_y - i \* (circle\_radius + 13) # Add spacing between fruits

is\_fruit = random.choice(["apple", "grape", "skull"]) # Randomly select an object

In this part of our code, we planned for some of the falling fruits to drop in groups (randomly), allowing the player to occasionally earn double points while playing.

**Random Chance:** If len(circles) < 2 and random.randint(1, 180) == 1, a group of fruits is created. This randomly triggers the addition of new objects.

**Group Center Position:** group\_x determines the central position where the fruits will group together, which will appear in a specific area on the screen.

**Initial Height:** initial\_y sets the starting height from which the fruits will fall, starting from the top of the screen.

**Adding Fruits:** Using for i in range(random.randint(1, 2)), either 1 or 2 fruits are added at the same time, ensuring multiple fruits can fall within the same group.

**Position Calculation:** The x and y positions of the fruits are randomly set within the group area. The x is chosen near the group center, while y places the fruits with a gap between them.

**Object Type:** is\_fruit randomly selects the type of object to fall, such as an apple, grape, or skull.

Check overlap

# Check for overlap

overlap = False

for existing\_x, existing\_y, \_, existing\_type in circles:

if abs(x - existing\_x) < circle\_radius \* 2 and abs(y - existing\_y) < circle\_radius \* 2:

if (is\_fruit == "skull" and existing\_type in ["apple", "grape"]) or \

(is\_fruit in ["apple", "grape"] and existing\_type == "skull"):

overlap = True

break # If overlap is found, break the loop

if not overlap:

break # Exit the loop if no overlap is found and add the fruit

circles.append((x, y, "white", is\_fruit)) # Add the fruit to the list

for event in pygame.event.get():

if event.type == pygame.QUIT: # Exit the game if the close button is pressed

running = False

Our aim in this code is to optimize the falling of fruits because objects can fall on top of each other, for example, apples falling on top of each other. In this code, we tried to optimize and reduce this situation.

Checking for Events:

for event in pygame.event.get():

if event.type == pygame.QUIT: # Exit game if the close button is pressed

pygame.quit()

sys.exit()

In the above, if the player closes the game window, the game exits.

To Move the Plate:

The plate is moved left or right using the arrow keys. The plate's movement is restricted to stay within the screen boundaries.

keys = pygame.key.get\_pressed()

if not is\_failed and not is\_paused: # Update game state if not paused or game over

# Move plate left or right

if (keys[pygame.K\_LEFT] or keys[pygame.K\_a]) and plate\_x > 0:

plate\_x -= plate\_speed

if (keys[pygame.K\_RIGHT] or keys[pygame.K\_d]) and plate\_x < screenWidth - plate\_width:

plate\_x += plate\_speed

Collision Detection:

* If the plate catches a fruit, the score increases by 1.
* If the plate catches a skull, the game ends.
* If a fruit falls to the bottom of the screen, the game ends.

# Check for collisions with the plate

if plate\_y <= y <= plate\_y + plate\_height and plate\_x <= x <= plate\_x + plate\_width:

if is\_fruit == "apple":

score += 1 # Increase score for apple

collect\_sound.play()

elif is\_fruit == "grape":

score += 2 # Increase score for grape

collect\_sound.play()

else:

lives -= 1 # Lose a life for skull

skull\_sound.play()

if lives == 0:

is\_failed = True

circles.pop(i)

break

elif y > screenHeight: # Remove objects that fall off the screen

if is\_fruit in ["apple", "grape"]:

lives -= 1

if lives == 0:

is\_failed = True

circles.pop(i)

Break

To update the screen:

* The screen should be updated at each iteration of the game loop.
* The FPS limit ensures consistent and smooth gameplay.

pygame.display.update()

clock.tick(FPS)

# **Conclusion**

Python continues to solidify its position as one of the most widely used and versatile programming languages. Its ease of use, extensive library support, and adaptability to emerging fields like artificial intelligence, machine learning, and data science make it indispensable for modern development. As technology evolves, Python's strong community and cross-platform compatibility ensure its relevance in solving complex problems and driving innovation.

However, Python is not without limitations. Challenges like performance constraints, dependency management, and limited mobile application support mean it may not always be the best choice for every scenario. Despite these drawbacks, Python's strengths—such as rapid prototyping, integration capabilities, and its dominance in data-driven domains—outweigh its weaknesses. As demand for simplicity, automation, and AI-driven applications grows, Python will likely remain a cornerstone of programming education, research, and industry application for years to come.

# References

1. <https://en.wikipedia.org/wiki/History_of_Python> (web site)