

Build a Tetris Game in Python

This tutorial guides you through building an **object-oriented Tetris game** using **Python and Pygame**, with correct Tetromino logic, real-time controls, clean geometry (no grid padding), and a restartable game loop.

Prerequisites

You should:

- Understand basic Python syntax and structures (loops, classes, functions)
- Be familiar with 2D coordinate systems
- Have Python 3.7+ installed

Install Pygame

To get started, install the pygame library:

```
pip install pygame
```

This provides tools for handling graphics, input, timing, and drawing rectangles for Tetris blocks.

Game Constants and Shape Definitions

We begin by importing required modules and defining global constants for the screen, game board, colors, and Tetromino shapes:

```
import pygame
import random
import sys
pygame.init()
# Screen dimensions
WIDTH, HEIGHT = 300, 600
COLS, ROWS = 10, 20
BLOCK_SIZE = WIDTH // COLS

# Timing
FPS = 60
FALL_DELAY = 500 # ms between automatic drops

# Colors
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
COLORS = [
    (0, 255, 255), # I
    (0, 0, 255),   # J
    (255, 165, 0), # L
    (255, 255, 0), # O
    (0, 255, 0),   # S
    (128, 0, 128), # T
    (255, 0, 0),   # Z
]

# Canonical Tetromino shape definitions (no padding, clean geometry)
```

```

SHAPES = [
    [[1, 1, 1, 1]],           # I
    [[1, 0, 0], [1, 1, 1]],  # J
    [[0, 0, 1], [1, 1, 1]],  # L
    [[1, 1], [1, 1]],        # O
    [[0, 1, 1], [1, 1, 0]],  # S
    [[0, 1, 0], [1, 1, 1]],  # T
    [[1, 1, 0], [0, 1, 1]]   # Z
]

```

Explaining the Shape Format

Each shape is defined using 1s and 0s in a 2D list. This format is simple to manipulate, rotate, and map to screen coordinates.

```
SHAPES = [
```

The coordinates of each 1 in a shape map directly to block positions on the board using the Tetromino's current .x and .y offsets.

Each Tetromino is a 2D List:

- Each **inner list is a row** of the shape.
- Each 1 represents a block in that row. A 0 means empty space.

This format allows us to easily:

- **Render** the block on a grid
- **Rotate** the block by transposing and reversing rows
- **Check for collisions** using simple 2D grid math

For example:

```
[[1, 0, 0],
 [1, 1, 1]]
```

This is the "J" piece:

- Top row has one block (leftmost)
- Bottom row has three blocks

This will look like this



Screen Dimensions

```
WIDTH, HEIGHT = 300, 600
```

- **WIDTH:** Total screen width in pixels.
- **HEIGHT:** Total screen height in pixels.
- These are chosen to fit a 10x20 grid comfortably.

Grid Configuration

```
COLS, ROWS = 10, 20
```

```
BLOCK_SIZE = WIDTH // COLS
```

- **COLS:** Number of columns on the board (standard Tetris = 10).
- **ROWS:** Number of vertical cells (standard = 20).
- **BLOCK_SIZE:** Each Tetromino block will be this many pixels wide and tall.

This ensures that the blocks are square and the entire playfield fits into the window.

Timing

```
FPS = 60
```

```
FALL_DELAY = 500
```

- **FPS:** How many frames per second to update the game screen. Affects animation smoothness.
- **FALL_DELAY:** Time in milliseconds before the current Tetromino automatically drops by 1 cell.

You can reduce `FALL_DELAY` dynamically to increase difficulty as the player progresses.

Colors

```
COLORS = [ ... ]
```

Each Tetromino gets a unique RGB color so the player can distinguish them easily. These are used to color the blocks during rendering.

Create the Tetromino Class

The Tetromino class represents a single falling Tetris shape (such as I, J, L, O, S, T, or Z). It encapsulates all the properties and behaviors required to manage a piece during gameplay.

Each Tetromino is treated as an object with:

- **Shape geometry:** Defined as a 2D list (`self.shape`) where each 1 represents a block and 0 is empty space.
- **Color:** The RGB color used when drawing this shape on screen.
- **Grid position:** The shape's top-left anchor on the game grid, controlled by `self.x` and `self.y`.
- **Rotation logic:** A method to rotate the shape in-place, without needing predefined rotation states.

This object-oriented design keeps the behavior of each piece self-contained and modular, allowing the game engine to work with all Tetrominoes generically.

How It Works

```
class Tetromino:
    def __init__(self, shape, color):
```

```

self.shape = shape
self.color = color
self.x = COLS // 2 - len(shape[0]) // 2 # center hz1
self.y = 0 # spawn at top

```

shape is a 2D list like `[[1, 1, 1, 1]]` or `[[1, 0, 0], [1, 1, 1]]`. Each 1 represents an active block in the piece.

color is the color is selected from the predefined `COLORS` list and corresponds to the shape type.

self.x is the horizontal position on the grid where the Tetromino starts. It is centered by subtracting half the width of the shape from the midpoint of the board (`COLS // 2`).

self.y is the vertical position always begins at 0 so the piece enters from the top row.

Rotation Logic

```

def rotate(self):
    self.shape = [list(row) for row in zip(*self.shape[::-1])]

```

This is a **90-degree clockwise rotation** using a common Python idiom:

self.shape[::-1] reverses the rows (top to bottom becomes bottom to top).

zip(*...) transposes the matrix (columns become rows).

list(row) ensures each new row is mutable.

Example:

Before rotation:

```

[[1, 0, 0],
 [1, 1, 1]]

```

After rotation:

```

[[1, 1],
 [1, 0],
 [1, 0]]

```

Occupied Grid Cells

```

def get_cells(self):
    return [(self.x + j, self.y + i)
            for i, row in enumerate(self.shape)
            for j, val in enumerate(row) if val]

```

This method calculates the **absolute (x, y)** positions on the board where the Tetromino's blocks currently are.

- `i` = row index within the shape matrix (vertical offset)
- `j` = column index within the shape matrix (horizontal offset)
- `self.x + j` translates local shape column to board column
- `self.y + i` translates local shape row to board row

- Only positions where val is 1 are returned (i.e., where blocks exist)

Build the Game Class

The Game class is the central controller of the entire Tetris engine. It orchestrates the game state, input handling, collision detection, rendering, and the rules of play.

What This Class Manages

- The board state — a 2D grid that keeps track of locked blocks
- The currently falling Tetromino
- Spawning and rotating new Tetrominoes
- Moving pieces left, right, or down
- Collision detection and locking when a piece lands
- Clearing full lines
- Detecting game over conditions
- Handling user input via the keyboard
- Rendering all elements to the screen each frame
- Providing a way to restart after game over

Data Structures

```
self.grid = [[BLACK for _ in range(COLS)] for _ in range(ROWS)]
```

The board is represented as a 2D list (list of rows).

Each cell holds a color:

BLACK means empty

Any other RGB color means that cell is occupied by a locked Tetromino block

This makes rendering and logic simple — if `grid[y][x] != BLACK`, it's occupied

Core Methods (with Concepts)

`spawn_tetromino()`

- Creates a new random Tetromino using a shape/color pair.
- Places it at the top of the board (`y = 0`), centered.
- If the spawn position is invalid (i.e. overlaps locked blocks), the game is over.

`valid_position(tetromino, dx=0, dy=0)`

- Checks whether moving the given Tetromino by (`dx`, `dy`) would result in a valid state.
- Valid = does not go out of bounds and does not collide with locked cells on the grid.

lock_tetromino()

- Called when a Tetromino can no longer move down.
- Adds all of its occupied cells to the grid (converts falling block into static blocks).
- Triggers line-clearing logic.

clear_lines()

- Scans the grid for full rows (rows with no BLACK blocks).
- Removes full rows and inserts empty ones at the top.
- This is where the game gets its core challenge mechanic and scoring potential.

drop_one()

- Moves the current Tetromino down by one row.
- If it cannot move, it locks it in place.

hard_drop()

- Instantly drops the current Tetromino all the way to the lowest valid position and locks it.
- Triggered by pressing the SPACE key.
- Improves game speed and responsiveness.

rotate()

- Attempts to rotate the current Tetromino.
- If the rotated shape causes an invalid position, the rotation is reverted.

draw_board()

- Draws the current state of the grid, block by block.
- Uses the color stored in each grid cell.
- Only draws blocks that are not BLACK.

draw_current()

- Draws the current falling Tetromino using its `get_cells()` coordinates.

Main Game Loop (run())

This is the heart of the game — where everything is updated and rendered frame by frame.

Inside `run()`:

1. Clock Tick

```
self.clock.tick(FPS)
```

Regulates the speed of the game to 60 frames per second.

2. Handle Events

```
for event in pygame.event.get():
    ...
```

Processes keyboard input (left, right, rotate, hard drop, restart).

3. Auto-Fall Timer

```
if pygame.time.get_ticks() - self.drop_time > FALL_DELAY:
    self.drop_one()
```

Every FALL_DELAY ms, the Tetromino automatically drops one row.

4. Render Scene

- Clear screen
- Draw locked blocks (draw_board())
- Draw current piece (draw_current())
- Show game over screen if needed

5. Display Frame

```
pygame.display.flip()
```

Refreshes the screen with the updated content.

Game Over & Restart

- If a Tetromino spawns in an invalid position (i.e. overlapping blocks at the top), the game sets `self.running = False` and shows a **game over screen**.
- Pressing the R key resets the game using the `reset()` method.

Run the Game

The `if __name__ == "__main__":` block creates a Game object and starts the game loop:

```
if __name__ == "__main__":
    Game().run()
```

Optional Extensions

- Add a **scoring system**: Track lines cleared
- Add **preview next piece**
- Add a **hold** feature (swap current with held Tetromino)
- Add **levels and speed increase**
- Add **sound effects or music**
- Show a **ghost piece** (shadow where it will land)