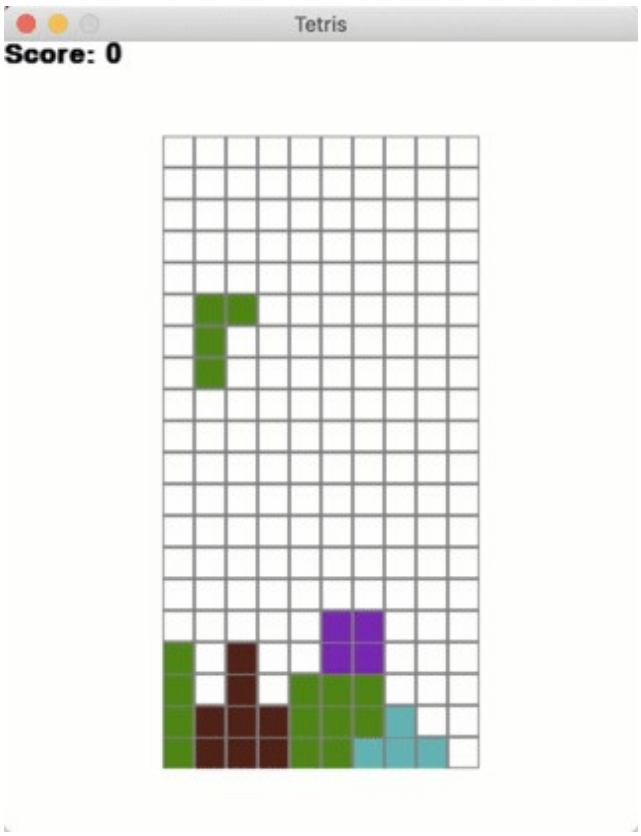
How to write Tetris in Python

Timur Bakibayev Follow
May 18, 2020 · 5 min read ★

Step by step guide to writing Tetris in Python with PyGame

In this tutorial, we will write a simple Tetris using the PyGame library in Python. The algorithms inside are pretty simple but can be a little challenging for the beginners. We will not concentrate on PyGame mechanics too much, but rather focus on the game logic. If you are too lazy to read all the stuff, you may simply copy and paste the code in the end.



Tetris Game

Prerequisites

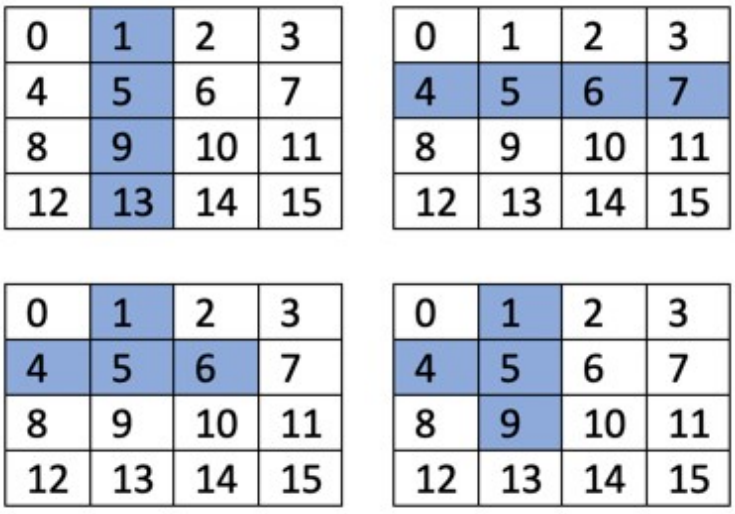
1. Python3. This may be downloaded from the official website.
2. PyGame. Go to your command prompt or terminal, depending on the OS you are using, and type `pip install pygame` or `pip3 install pygame`.
3. Basic knowledge of Python. See my other articles for that if needed.

You may experience issues with installing PyGame or Python itself, but this is out of scope here. Please refer to StackOverflow for that :)

I personally experienced a problem on Mac with having anything displayed on the screen, and installing some specific version of PyGame solved the problem: `pip install pygame==2.0.0.dev4`.

The Figure Class

We start with the Figures class. Our goal is to store the figure types together with the rotations. We could, of course, rotate them using matrix rotation, but that can make it too complex.



The main idea of figures representation

So, we simply have a list of lists of figures like that:

```
class Figure:
    figures = [
        [[1, 5, 9, 13], [4, 5, 6, 7]],
        [[1, 2, 5, 9], [0, 4, 5, 6], [1, 5, 9, 8], [4, 5, 6, 10]],
        [[1, 2, 6, 10], [5, 6, 7, 9], [2, 6, 10, 11], [3, 5, 6,
7]],
        [[1, 4, 5, 6], [1, 4, 5, 9], [4, 5, 6, 9], [1, 5, 6, 9]],
        [[1, 2, 5, 6]],
    ]
```

Where the main list contains figure types, and the inner lists contain their rotations. The numbers in each figure represent the positions in a 4x4 matrix where the figure is solid. For instance, the figure [1,5,9,13] represents a line. To better understand that, please refer to the picture above.

As an exercise try to add some missing figures here, namely the “z” figures.

The `__init__` function would be as follows:

```
class Figure:
    ...
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.type = random.randint(0, len(self.figures) - 1)
        self.color = random.randint(1, len(colors) - 1)
        self.rotation = 0
```

where we randomly pick a type and a color.

And we need to quickly be able to rotate and get the current rotation of a figure, for this we have these two simple methods:

```
class Figure:
    ...
    def image(self):
        return self.figures[self.type][self.rotation]

    def rotate(self):
        self.rotation = (self.rotation + 1) %
```

```
    len(self.figures[self.type])
```

The Tetris Class

We first initialize the Game with some variables:

```
class Tetris:
    level = 2
    score = 0
    state = "start"
    field = []
    height = 0
    width = 0
    x = 100
    y = 60
    zoom = 20
    figure = None
```

where the state tells us if we are still playing a game or not. The `field` is the field of the game that contains zeros where it is empty, and the colors where there are figures (except the one that is still flying down).

We initialize the game with the following simple method:

```
class Tetris:
    ...
    def __init__(self, height, width):
        self.height = height
        self.width = width
        for i in range(height):
            new_line = []
            for j in range(width):
                new_line.append(0)
            self.field.append(new_line)
```

That creates a field with the size `height x width`.

Creating a new figure and position it at coordinates (3,0) is simple:

```
class Tetris:
    ...
    def new_figure(self):
        self.figure = Figure(3, 0)
```

The more interesting function is to check if the currently flying figure intersecting with something fixed on the field. This may happen when the figure is moving left, right, down, or rotating.

```
class Tetris:
    ...
    def intersects(self):
        intersection = False
        for i in range(4):
            for j in range(4):
                if i * 4 + j in self.figure.image():
                    if i + self.figure.y > self.height - 1 or \
                            j + self.figure.x > self.width - 1 or
\
                            j + self.figure.x < 0 or \
                            self.field[i + self.figure.y][j +
self.figure.x] > 0:
                        intersection = True
        return intersection
```

It is pretty simple: we go and check each cell in the 4x4 matrix of the current Figure, whether it is out of game bounds and whether it is touching some busy game field. We check if `self.field[..][..] > 0`, because there may be any color. And if there is a zero, that means that the field is empty, so there is no problem.

Having this function, we can now check if we are allowed to move or rotate the Figure. If it moves down and intersects, then this means we have reached the bottom, so we need to “freeze” the figure on our field:

```
class Tetris:
    ...
    def freeze(self):
        for i in range(4):
            for j in range(4):
                if i * 4 + j in self.figure.image():
```

```
                    self.field[i + self.figure.y][j +
self.figure.x] = self.figure.color
        self.break_lines()
        self.new_figure()
        if self.intersects():
            game.state = "gameover"
```

After freezing, we have to check if there are some full horizontal lines that should be destroyed. Then we create a new Figure, and if it already intersects, then game over :)

Checking the full lines is relatively simple and straightforward, but pay attention to the fact that destroying a line goes from the bottom to the top:

```
class Tetris:
    ...
    def break_lines(self):
        lines = 0
        for i in range(1, self.height):
            zeros = 0
            for j in range(self.width):
                if self.field[i][j] == 0:
                    zeros += 1
            if zeros == 0:
                lines += 1
                for i1 in range(i, 1, -1):
                    for j in range(self.width):
                        self.field[i1][j] = self.field[i1 - 1][j]
        self.score += lines ** 2
```

Now, we are missing the moving methods:

```
class Tetris:
    ...
    def go_space(self):
        while not self.intersects():
            self.figure.y += 1
        self.figure.y -= 1
        self.freeze()

    def go_down(self):
        self.figure.y += 1
        if self.intersects():
            self.figure.y -= 1
            self.freeze()

    def go_side(self, dx):
```

```
        old_x = self.figure.x
        self.figure.x += dx
        if self.intersects():
            self.figure.x = old_x

    def rotate(self):
        old_rotation = self.figure.rotation
        self.figure.rotate()
        if self.intersects():
            self.figure.rotation = old_rotation
```

As you can see, the `go_space` method practically duplicates the `go_down` method, but it goes down until it reaches the bottom or some fixed figure.

And in every method, we remember the last position, change the coordinates, and check if there is an intersection. If there is, we return to the previous state.

PyGame and the Complete Code

We are almost done!

There is some simple logic left, which is the game loop and the PyGame stuff. So, let's see at the complete code now:

Run and enjoy the game! :)

And don’t forget to share it with your friends!