How I used PIL to dynamically generate tournament brackets

I used the Pillow library (PIL) to draw a combination of lines, shapes, and text, to illustrate the tournament bracket and store it in an Image object. The contents of the image are then saved to a PNG file and rendered on the screen by the tournament-detail template.

How does it work?

The width and length of each “entry” (horizontal line representing a position in the bracket) is calculated based on the size of the screen and the number of entries.

A list of `nodes` are constructed which contain the x and y value of each entry. Then the nodes are sent to a method which draws a horizontal line for each node, provided the width of the entries. This method also constructs a new list of nodes, to be sent to a method that draws vertical lines.

```
from PIL import Image, ImageDraw, ImageFont
```

```
class Node:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def draw_horizontal_lines(self, draw, nodes, x_length):
        nodes_size = len(nodes)
        flag = False
        extra_future_horizontal_nodes = []
        future_vertical_nodes = []
        for i, node in enumerate(nodes):
            draw.line([(node.x, node.y), (node.x + x_length, node.y)], fill="black")

            if (i % 2 == 0):
                if (i < nodes_size - 1):
                    future_vertical_nodes.append(Node(node.x + x_length, node.y))
                elif (nodes_size > 1):
                    extra_future_horizontal_nodes.append(Node(node.x + x_length, node.y))
                    draw.line([(node.x + x_length, node.y), (node.x + x_length * 2, node.y)],
fill="black")

        backup_dy = 0
        if (nodes_size > 3):
            if (nodes[nodes_size - 1].y - nodes[nodes_size - 2].y != nodes[1].y - nodes[0].y
and nodes_size % 2 == 0):
                flag = True
                backup_dy = nodes[nodes_size - 1].y - nodes[nodes_size - 2].y

        if (nodes_size > 1):
            y_length = nodes[1].y - nodes[0].y
            self.draw_vertical_lines(draw, future_vertical_nodes, y_length,
extra_future_horizontal_nodes, flag, backup_dy)
```

The `draw_vertical_lines` method connects the right edges of each of the entries. Each vertical line represents a matchup between two players. This method also generates a new list of nodes to send to `draw_horizontal_lines`, in which each will represent the winner of the matchups of the current round, and an entry for the next round.

```
    def draw_vertical_lines(self, draw, nodes, y_length, extra_horizontal_nodes, flag,
backup_dy):
        future_horizontal_nodes = []
        for i, node in enumerate(nodes):
            dy = y_length
            if (i == len(nodes) - 1 and flag):
                dy = backup_dy

            y = node.y + (dy // 2)
            future_horizontal_nodes.append(Node(node.x, y))
            draw.line([(node.x, node.y), (node.x, node.y + dy)], fill="black")

        future_horizontal_nodes.extend(extra_horizontal_nodes)
        self.draw_horizontal_lines(draw, future_horizontal_nodes, self.entry_width)
```

`draw_horizontal_lines` and `draw_vertical_lines` keep **recursively** calling each other with a smaller and smaller amount of nodes, until there is a single winning entry and an entire bracket is drawn onto an image. Each of the names of the players are then written on the leftmost entries.

The image is then saved to a PNG file, where the path is dependent on the primary key of the tournament. The tournament-detail template then renders the file and displays it on the screen to the user.

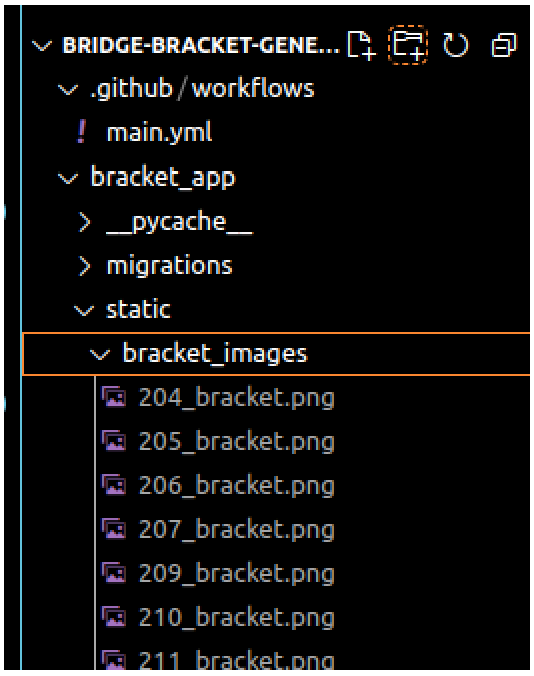
View function:

```
def tournament_detail_view(request, pk):
    tournament = Tournament.objects.get(pk=pk)

    jsonDec = json.decoder.JSONDecoder()
    players = jsonDec.decode(tournament.players)

    context = {
        'tournament': tournament,
        'player_list': players,
        'path': "bracket_images/" + str(pk) + "_bracket.png",
        'size': len(players)
    }

    return render(request, 'bracket_app/tournament_detail.html', context)
```

tournament-detail template:

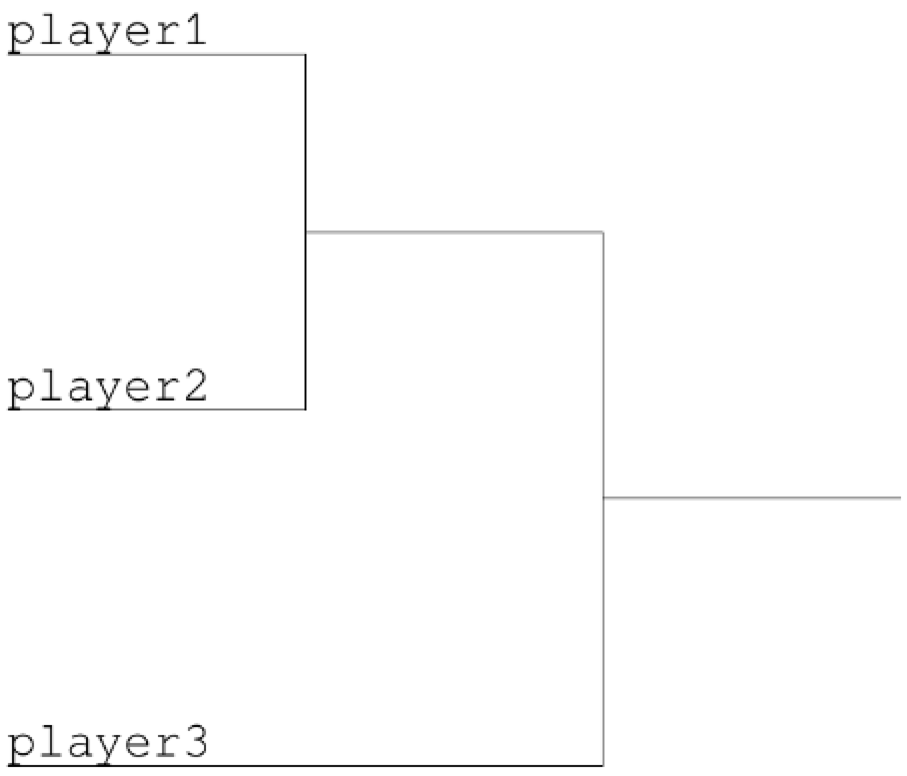
```
{% extends 'bracket_app/base_template.html' %}

{% load static %}

{% block content %}
<h1>{{ tournament.title }}</h1>

<p><strong>Completed:</strong> {{ tournament.completed }}</p>
<p><strong>Organizer:</strong> {{ tournament.creator }}</p>
<p><strong>Number of Players:</strong> {{ size }}</p>

<img src="{% static path %}" alt="Tournament Bracket">

{% endblock %}
```

Where the images are stored:



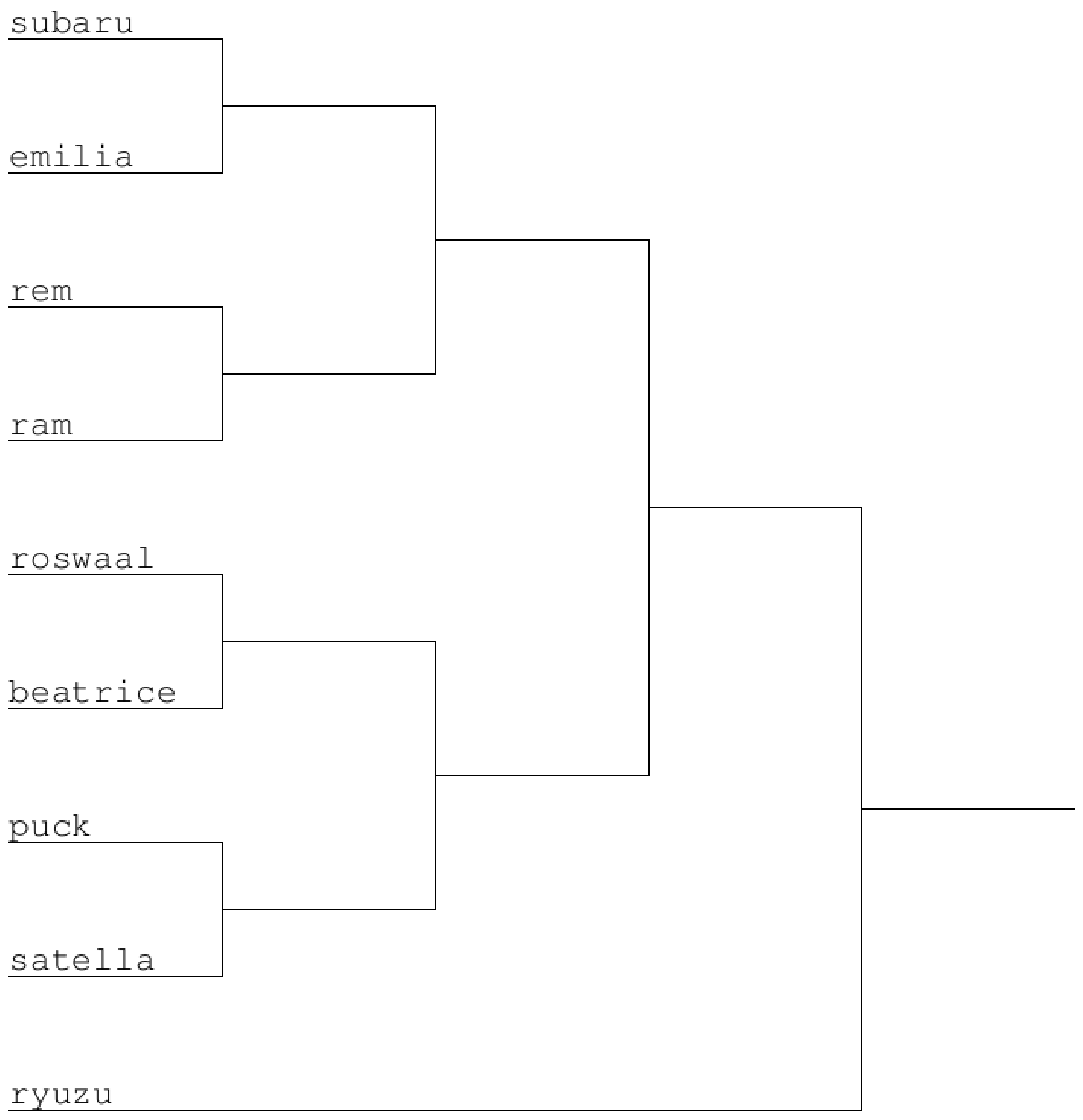
```
BRIDGE-BRACKET-GENE...
  .github / workflows
    main.yml
  bracket_app
    __pycache__
    migrations
    static
      bracket_images
        204_bracket.png
        205_bracket.png
        206_bracket.png
        207_bracket.png
        209_bracket.png
        210_bracket.png
        211_bracket.png
```

Example of a bracket:



Things to keep in mind

There are some issues with my solution. For example, I didn't account for what should happen if there are an uneven number of players for multiple rounds. Here's what happens when there's 9 players:



Notice how **ryuzu** has 3 bye rounds in a row. That's not how it's supposed to work. What it should do is give ryuzu a bye round, and then give another player such as **satella** a bye round. The bye rounds should be spread out amongst players, otherwise it's unfair. I think to implement this, you would need to add another attribute to the `Node` class like so, to keep track of whether it exists in a bye round.

```python
class Node:
    def __init__(self, x, y, bye=False):
        self.x = x
        self.y = y
        self.bye = bye
```