Odd Semester (2023)

BINUS UNIVERSITY

BINUS INTERNATIONAL

Algorithm and Programming

Final Project Report

Student Information:

Surname: Basuki **Given Name:** Adrian Nugroho Basuki **Student ID:** 2702298210

Course Code : COMP6047001 **Course Name :** Algorithm and Programming

Class : L1AC **Lecturer** : Jude Joseph Lamug Martinez, MCS

Type of Assignment: Final Project Report

Submission Pattern

Due Date : 12 January 2024 **Submission Date** : 12 January 2024

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

Binus International seriously regards all forms of plagiarism, cheating, and collusion as academic offenses which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept, and consent to Binus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance.

Signature of Student:

[illegible]

Adrian Nugroho Basuki

Table Of Contents

Introduction

I. Background

For this final project, I decided to create a Discord bot named EmpyBot. The name of this bot originated from the phrase multi-purpose, reflecting the bot's nature, which could be abbreviated into MP, thus resulting with the name Empy.

As Discord has evolved from a niche gamer-centric platform to a mainstream hub for diverse online communities, users face new challenges in maintaining an organized, entertaining, and engaging space. EmpyBot aims to bridge these gaps by offering a comprehensive solution. Drawing inspiration from Discord's ascent, EmpyBot is designed to align with the platform's user-centric ethos, addressing common pain points such as message management and spam. By incorporating interactive elements, EmpyBot taps into Discord's social nature, catering to the platform's diverse and growing user base. In addition, the integration of live information further solidifies EmpyBot as a versatile companion, aligning with Discord's commitment to providing users with a feature-rich and enjoyable communication experience. This project not only delves into the technical aspects of bot development but also underscores the symbiotic relationship between evolving platforms like Discord and the demand for innovative solutions.

Project Specifications

I. Description

This Discord bot was built using a few Python libraries and APIs including Nextcord, Asyncio, Random module, and aiohttp framework. These imported resources allowed the code to access the bot through its given token, enabling it to respond to user commands within the server's text channels. EmpyBot offers users a range of features with various functions that may assist users in managing their servers, or for entertainment purposes. Its large features include a poll creator, weather information display, and a tic tac toe game which has two different modes: single player and one versus one. Furthermore, the bot also provides additional smaller, but useful, features such as a command that allows users to delete a specified number of messages in a text channel and a system that detects if a user is sending too many messages in a short amount of time (spamming). EmpyBot utilizes the slash commands feature from Discord to be more descriptive and organized with the commands, so that users could easily understand what the feature does, as well as the required inputs. However, I still implemented commands that are run by using the bot's assigned prefix ("-") for some of it's features, as I thought it would be more practical to use the prefix instead of the slash commands.

II. Bot Requirements

a. Nextcord API

Nextcord is a Python library and API designed as an enhancement to the Discord.py library. It serves as a powerful tool that helps developers create discord bots and applications with ease. Nextcord provides a seamless interaction with the Discord API, offering improved performance, enhanced functionality, and better support for Discord's evolving features. Developers can harness Nextcord to build feature-rich bots, implement real-time updates, manage user interactions, and access a wide array of Discord functionalities.

In this final project, Nextcord was integrated for efficient interaction management. By creating an instance of `commands.Bot` with a specified command prefix and enabling all the intents, the bot gains access to information and events provided by Nextcord. This allows the Discord bot to handle commands, events and even errors.

b. Asyncio library

Asyncio library is an integral part of asynchronous programming in Python that increases the efficiency and responsiveness of Discord bots. It allows tasks to run independently without waiting for each other to complete, therefore it can handle concurrent tasks and maintain responsiveness during operations that might otherwise cause delays.

I used Asyncio's sleep function to create an asynchronous time delay without disrupting other functions. This function was applied in the poll creation feature to wait a specified amount of time before showing the results of the poll.

c. Random module

The random module in Python is a standard library module that provides functions for generating random numbers and performing various randomization tasks. It serves the purpose of introducing unpredictability and variability to the Discord bot's functionality.

This module was implemented in the creation of EmpyBot's Tic Tae Toe feature. It was used to decide the player who gets the first turn by generating a random integer of either 1 or 2 using the randint function. If it generates 1, then player one gets the first turn, and vice versa.

Other than that, it also played a role in the Tic Tac Toe game's feature where it allows users to play against the bot. To create the bot's moves based on the difficulty that the players selected, a chance of the bot making mistakes or blunders needed to be introduced to the code. This is where the random module came in handy. It allowed me to generate values from 0.0 to 1.0, using the random() function, and then checking if it's less than the specified blunder probability value for the chosen difficulty to decide whether the bot should follow the given algorithm or make a random move. This is where the randint function is used again. It generates an integer from one to nine to determine what the bot's next random move would be.

d. AIOHTTP Framework

The AIOHTTP framework is a Python library specifically designed for handling asynchronous HTTP requests and responses. It is built on top of the Asyncio module, making it well-suited for asynchronous programming, which makes it perfect for the creation of Discord bots. Its design allows for concurrent handling of multiple requests without blocking the execution of other tasks, making it suitable for responsive Discord bots.

In this project, the AIOHTTP framework is used for making asynchronous HTTP requests, specifically to retrieve real-time weather information from an external API. The aiohttp.ClientSession is utilized to create an asynchronous HTTP client session, allowing the bot to send HTTP requests asynchronously without blocking the execution of other tasks.

e. Typing Module

The typing module in Python provides a way to specify and check the type of variables. In this project, I implemented the optional class from this module in the poll creation feature, so that certain fields don’t have to be filled for the command to work. Optional is a generic type hint that indicates a type that can be either the specified type or None. If the user inputs nothing, then the value for the field that was left empty would be None.

f. Extras

Other than the Python libraries and APIs mentioned above, there are also other requirements that the bot needs in order to function.

First, to run the bot itself, Nextcord's run method needs the bot's token as the parameter. The bot's token can be found in the Discord Developer Portal site that shows the applications (bots) that you've created, as well as the settings and properties for each bot.

Aside from that, there are two other requirements to run the weather information feature of the bot, which are an API key and an API endpoint to retrieve current weather information. The API key is obtained by creating an account on https://www.weatherapi.com/. The website would then display your API key on the dashboard. Next, the API endpoint URL could be taken from the weather API documentation page. In my case, the URL uses current.json method to retrieve current weather information for a specified location in JSON format.

III. Features

a. Muting and unmuting members

These features are two different commands from EmpyBot that allow server administrators to mute and unmute members. These two commands use discord's slash command feature.

The /mute command takes in a member's name and a reason as input and could only be used by an administrator. Once the command is sent, the specified member will receive a new role that prevents them from entering messages in every text channel in the server, as well as a message through direct messages that notifies them with the reason of the mute. If the server does not have a 'Muted' role, the bot would create the role if it were given the proper permission to do so. This command could only work if the member doesn't have a 'Muted' role already.

The /unmute command, on the other hand, only takes a member's name as input and it will unmute the specified member if they have a 'Muted' role. If they do not have the role, then the bot would send a message that tells the users that the member is not muted. If the 'Muted' role does not exist in that server, then logically there wouldn't be anyone who are currently muted by the bot, as the bot creates the role first before muting anyone.

b. Spam detection

The spam detection feature is the only feature on this bot that is not a command. It is an event that constantly monitors the messages that users input in different text channels of the server. The message count of each user is stored inside a dictionary inside the code that resets every 3 seconds. If a person is starting to spam, the bot will send a warning message to tell the user to avoid spamming. Then, if they continue to spam, the user will automatically be muted. Of course, the administrator can easily unmute the user by using the /unmute command or by removing the 'Muted' role, but if they don't then the user will be muted for 5 minutes long. After 5 minutes, their 'Muted' role would be removed, and the bot would notify the user in the channel where the user spammed.

c. Clearing Messages

The clear command is a feature that allows users to clear a specified number of messages (maximum of 100 messages) in a Discord server's text channels. This command uses the bot's prefix to activate ("-clear"), and it could only be used by users who have the 'Administrator' permission in a server.

d. Poll Creation

The /createpoll slash command is a feature that allows users to create custom polls in a specified channel.

Upon writing the command, Discord will show a few input fields for the channel the user wishes to send this poll to, the poll duration in minutes, a question, 2 required options that answers the question, along with several optional fields, such as: 3 other options and a custom thumbnail.

After sending the command, the bot would reply to the user's message with an embed that consists of the elements that were inputted. The bot would also provide message reactions to the embed for the users to react to according to the options they choose.

After the specified amount of time, the poll would then close. The bot will edit the message by replacing an embed with a new one that displays the results of the poll with the percentage of votes as well as a percentage bar.

e. Tic Tac Toe

The tic tac toe game is a feature that has two different modes: player versus player mode and single player mode where a user plays against the bot.

The player versus player mode uses the slash command /ttt to activate. Upon writing the command, Discord will give two input fields in the message field that accepts users as input. After sending the command, the bot would reply with an embed, showing the empty tic tac toe board on the main part of the embed and whose turn it is in the footer. If it's their turn, the player gets to use the "-place" command followed by an integer from 1-9 to place a tile. Once their turn is done, it would be the other player's turn. This process of switching turns would repeat until either the board is full (a draw), or one player wins (three marks in one row). Once the game is over, the bot would then either notify the winner (if it is the case where there is a winner) or send a message telling the players that "It's a tie!". If a player wants to end the match early, they could use the "-end" command while their game is ongoing.

On the other hand, the single player mode uses the slash command /tttsp to activate. This mode basically works the same way as the player versus player mode, but the other player is a bot. Upon writing the command, Discord will give as single input field in the message field that gives the users difficulty options for the bot. The easy difficulty makes the bot more beginner friendly, as it is easier to play against, while the hard difficulty would make the bot much more difficult to defeat, and the medium mode would make the bot be somewhere in the middle between hard and easy.

The in-game commands "-place" and "-end" could only be used by players that are currently playing the game so that other users aren't able to disrupt their game. It could also only be used if a game has been started, otherwise the bot would reply with a message, telling the users that a game has not been started.

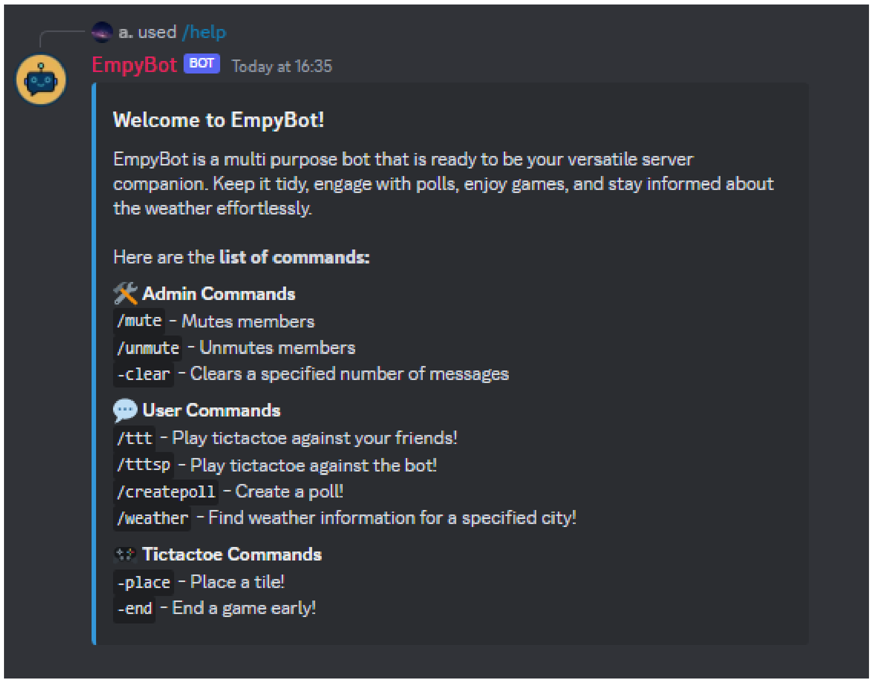
f. Weather Display

The /weather slash command is a feature that shows the local time and weather conditions, such as: temperature, humidity, and wind speed, of a specified city in the form of an embed.

Upon writing the command, Discord will give a single input field in the message field that takes in names of cities (strings) as input. Once the command message is sent, the bot would reply with an embed containing the aforementioned information. In addition, the color of the embed would change depending on the temperature of the region. The color of the embed would be orange for hotter temperatures and blue for colder temperatures.

IV. Screenshots

a. List of commands



a. used /help

EmpyBot BOT Today at 16:35

Welcome to EmpyBot!

EmpyBot is a multi purpose bot that is ready to be your versatile server companion. Keep it tidy, engage with polls, enjoy games, and stay informed about the weather effortlessly.

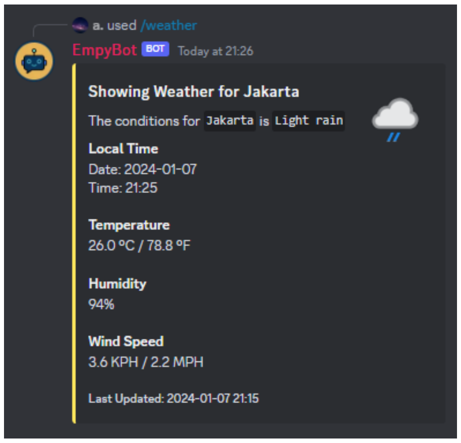
Here are the **list of commands:**

🛠️ **Admin Commands**
`/mute` - Mutes members
`/unmute` - Unmutes members
`-clear` - Clears a specified number of messages

💬 **User Commands**
`/ttt` - Play tictactoe against your friends!
`/tttsp` - Play tictactoe against the bot!
`/createpoll` - Create a poll!
`/weather` - Find weather information for a specified city!

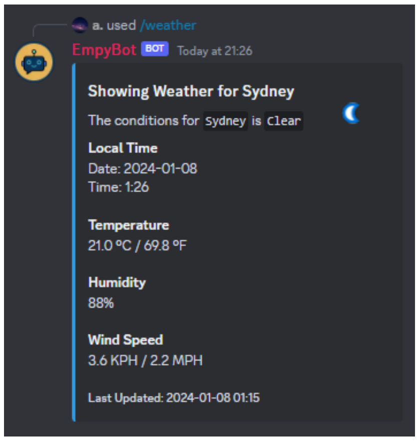
🎮 **Tictactoe Commands**
`-place` - Place a tile!
`-end` - End a game early!

b. Weather bot



a. used /weather
EmpyBot BOT Today at 21:26

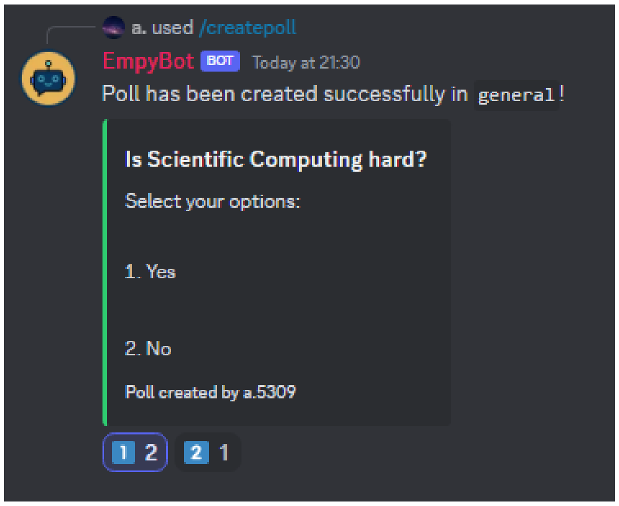
Showing Weather for Jakarta
The conditions for `Jakarta` is `Light rain`
Local Time
Date: 2024-01-07
Time: 21:25
Temperature
26.0 ºC / 78.8 ºF
Humidity
94%
Wind Speed
3.6 KPH / 2.2 MPH
Last Updated: 2024-01-07 21:15



a. used /weather
EmpyBot BOT Today at 21:26

Showing Weather for Sydney
The conditions for `Sydney` is `Clear`
Local Time
Date: 2024-01-08
Time: 1:26
Temperature
21.0 ºC / 69.8 ºF
Humidity
88%
Wind Speed
3.6 KPH / 2.2 MPH
Last Updated: 2024-01-08 01:15

c. Poll Creation



a. used /createpoll
EmpyBot BOT Today at 21:30
Poll has been created successfully in `general`!

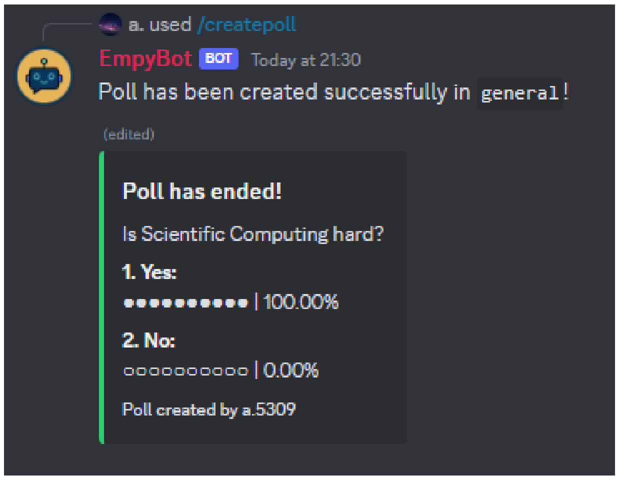
Is Scientific Computing hard?
Select your options:

1. Yes

2. No

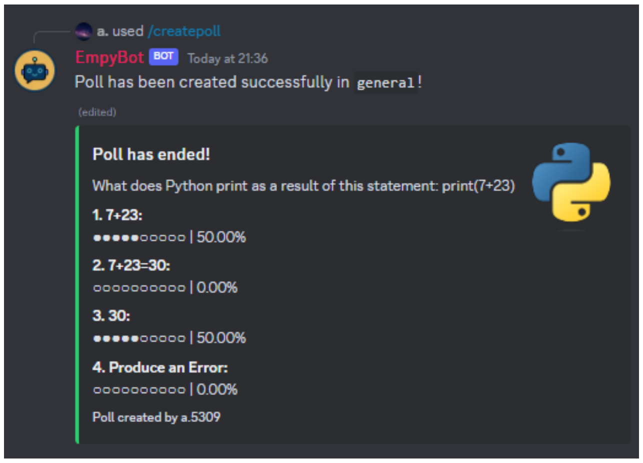
Poll created by a.5309

1 2 2 1



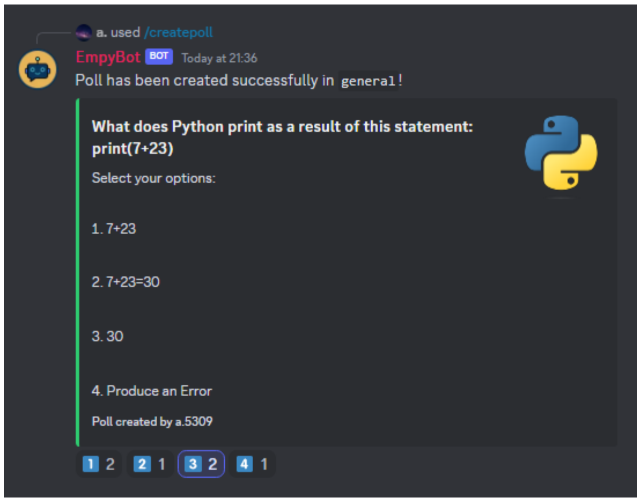
a. used /createpoll
EmpyBot BOT Today at 21:30
Poll has been created successfully in `general`!
(edited)

Poll has ended!
Is Scientific Computing hard?
1. Yes:
●●●●●●●●●● | 100.00%
2. No:
○○○○○○○○○○ | 0.00%
Poll created by a.5309



a. used /createpoll
EmpyBot BOT Today at 21:36
Poll has been created successfully in `general`!
(edited)

Poll has ended!
What does Python print as a result of this statement: print(7+23)
1. 7+23:
●●●●●○○○○○ | 50.00%
2. 7+23=30:
○○○○○○○○○○ | 0.00%
3. 30:
●●●●●○○○○○ | 50.00%
4. Produce an Error:
○○○○○○○○○○ | 0.00%
Poll created by a.5309

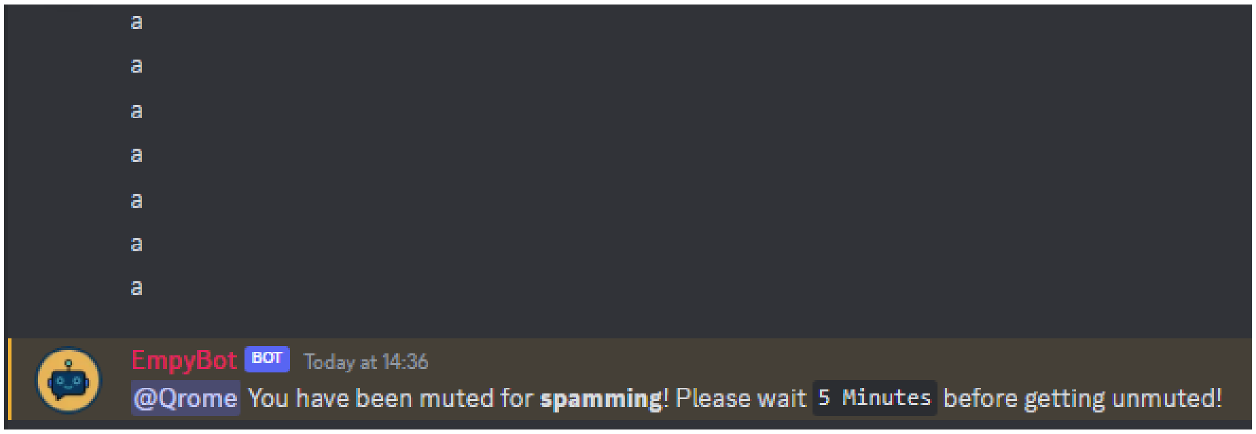


a. used /createpoll
EmpyBot BOT Today at 21:36
Poll has been created successfully in `general`!

What does Python print as a result of this statement: print(7+23)
Select your options:
1. 7+23
2. 7+23=30
3. 30
4. Produce an Error
Poll created by a.5309

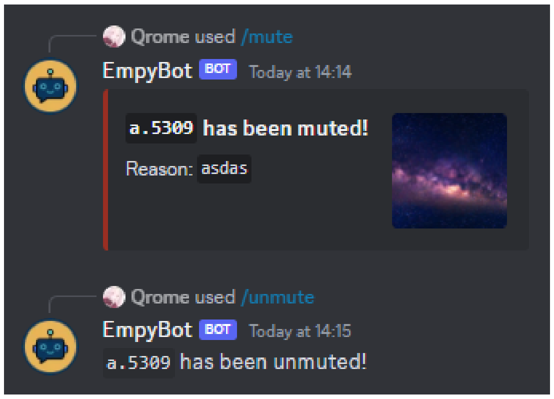
1️⃣ 2 2️⃣ 1 3️⃣ 2 4️⃣ 1

d. Muting, Unmuting and Spam Detection



a
a
a
a
a
a
a

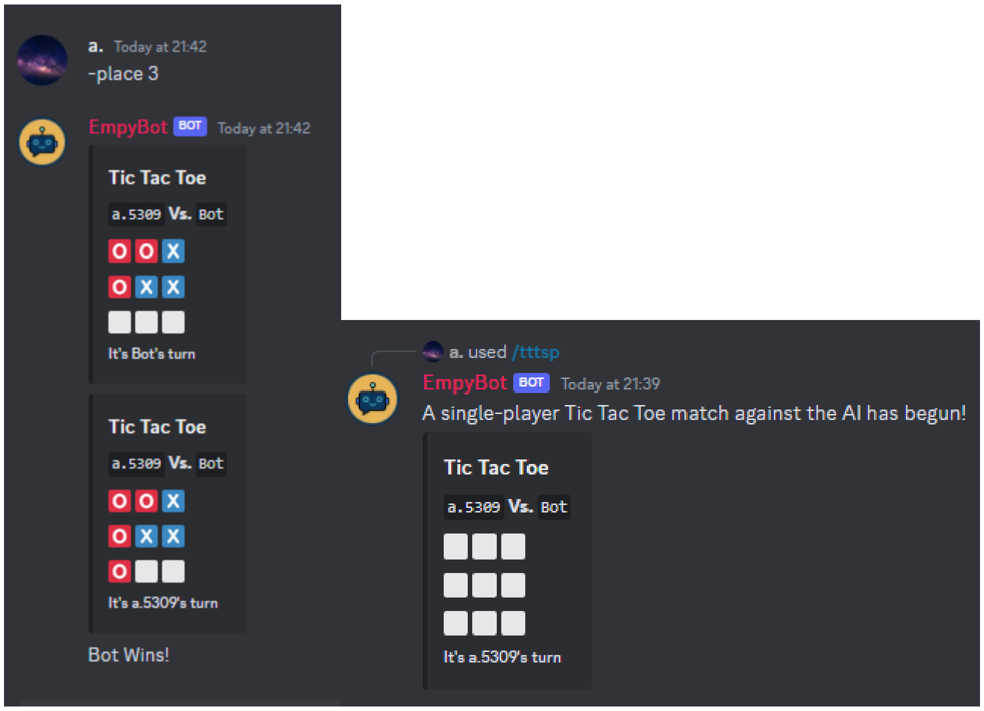
EmpyBot BOT Today at 14:36
@Qrome You have been muted for **spamming**! Please wait `5 Minutes` before getting unmuted!


Qrome used /mute
EmpyBot BOT Today at 14:14
a.5309 has been muted!
Reason: asdas

Qrome used /unmute
EmpyBot BOT Today at 14:15
a.5309 has been unmuted!

e. Tic tac toe

i. Single player


a. Today at 21:42
-place 3

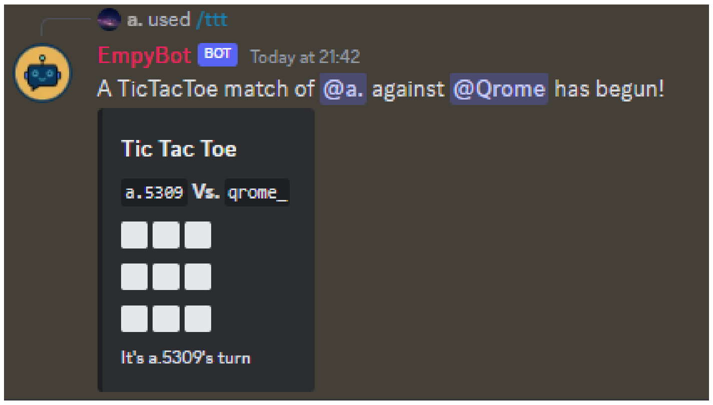
EmpyBot BOT Today at 21:42
Tic Tac Toe
a.5309 Vs. Bot
O O X
O X X
⬜ ⬜ ⬜
It's Bot's turn

Tic Tac Toe
a.5309 Vs. Bot
O O X
O X X
O ⬜ ⬜
It's a.5309's turn

Bot Wins!

a. used /tttsp
EmpyBot BOT Today at 21:39
A single-player Tic Tac Toe match against the AI has begun!
Tic Tac Toe
a.5309 Vs. Bot
⬜ ⬜ ⬜
⬜ ⬜ ⬜
⬜ ⬜ ⬜
It's a.5309's turn

ii. Player vs. player



a. used /ttt

EmpyBot BOT Today at 21:42

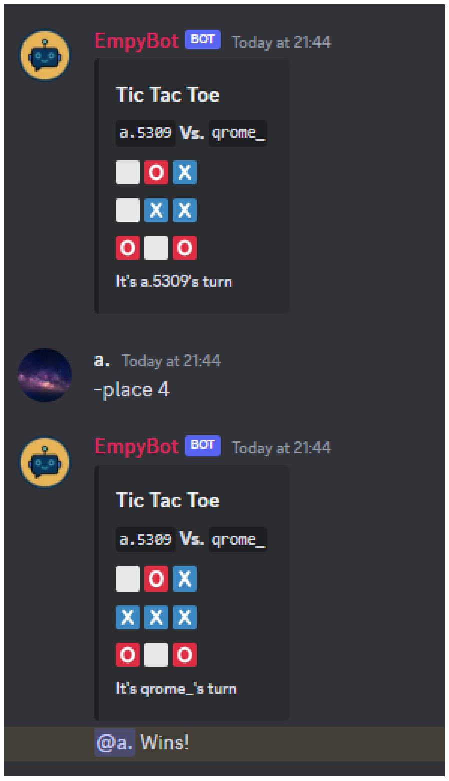
A TicTacToe match of @a. against @Qrome has begun!

Tic Tac Toe

a.5309 Vs. qrome_

⬜⬜⬜
⬜⬜⬜
⬜⬜⬜

It's a.5309's turn



EmpyBot BOT Today at 21:44

Tic Tac Toe

a.5309 Vs. qrome_

⬜ O X
⬜ X X
O ⬜ O

It's a.5309's turn

a. Today at 21:44

-place 4

EmpyBot BOT Today at 21:44

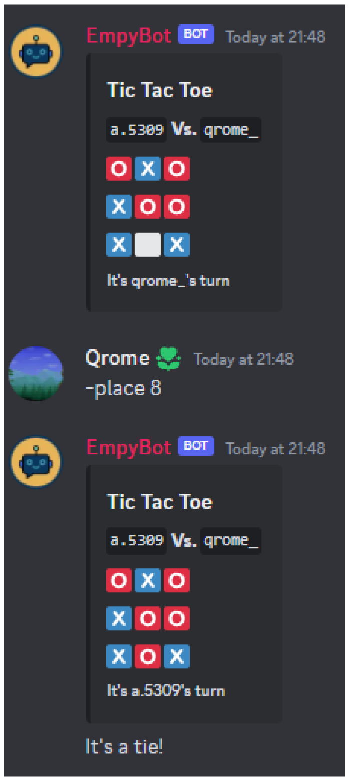
Tic Tac Toe

a.5309 Vs. qrome_

⬜ O X
X X X
O ⬜ O

It's qrome_'s turn

@a. Wins!



EmpyBot BOT Today at 21:48

Tic Tac Toe

a.5309 Vs. qrome_

O X O
X O O
X ⬜ X

It's qrome_'s turn

Qrome Today at 21:48

-place 8

EmpyBot BOT Today at 21:48

Tic Tac Toe

a.5309 Vs. qrome_

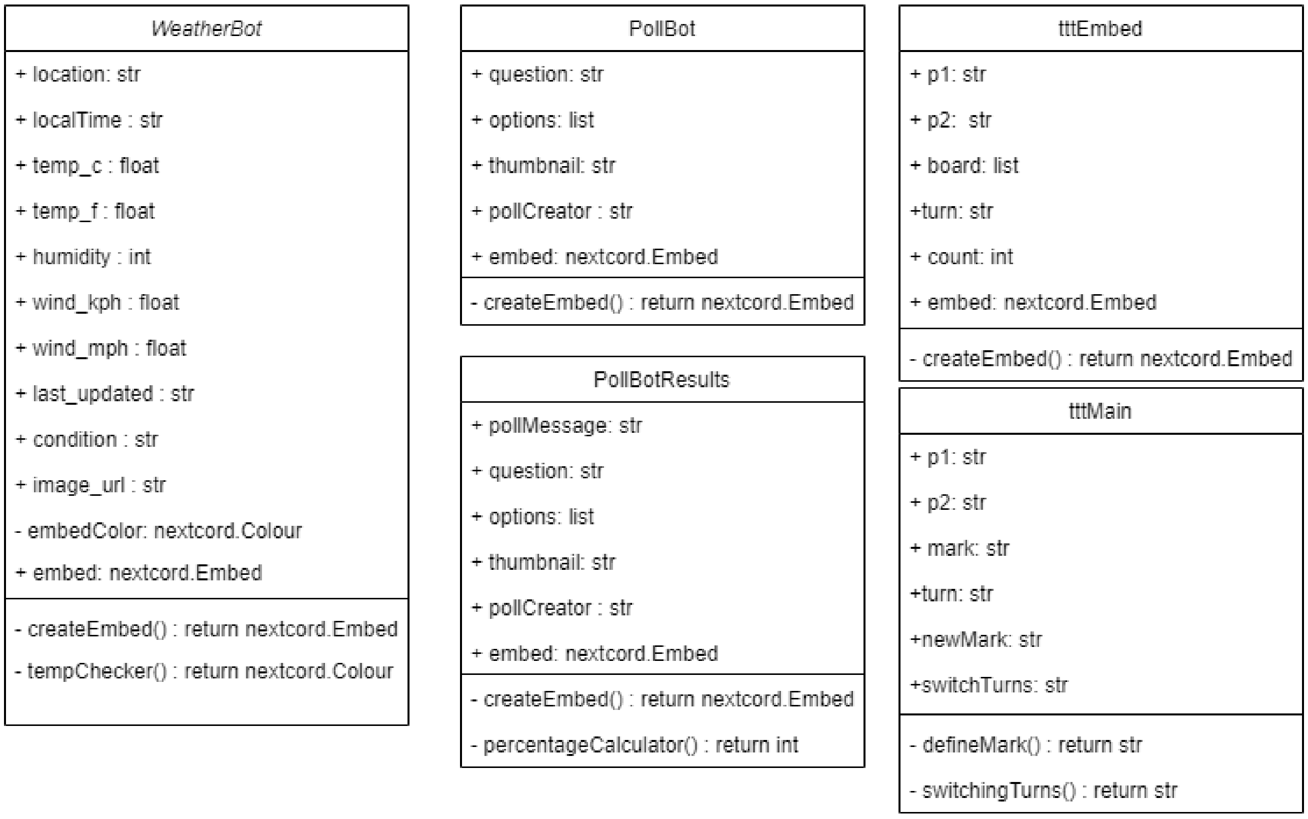
O X O
X O O
X O X

It's a.5309's turn

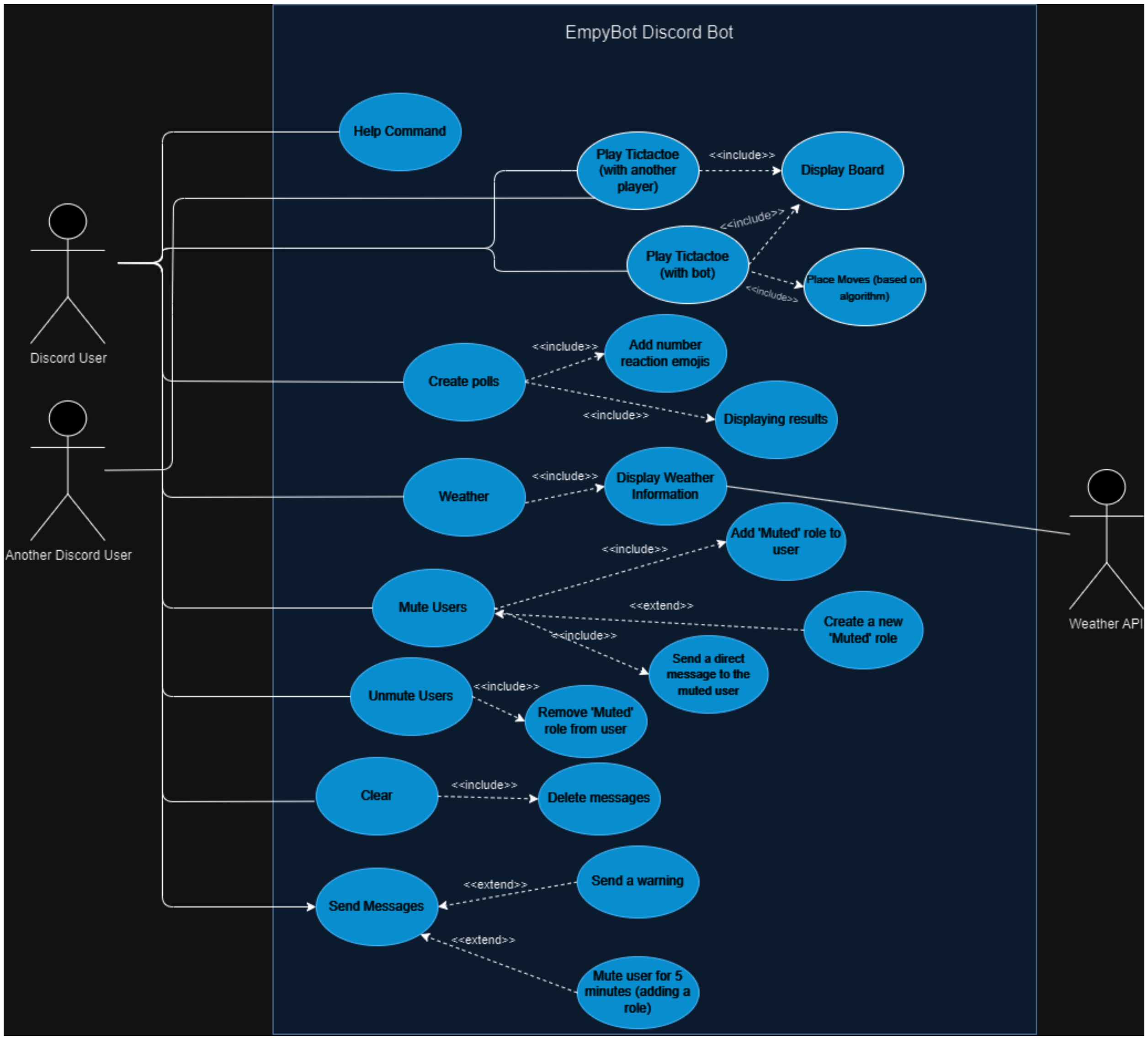
It's a tie!

Diagrams

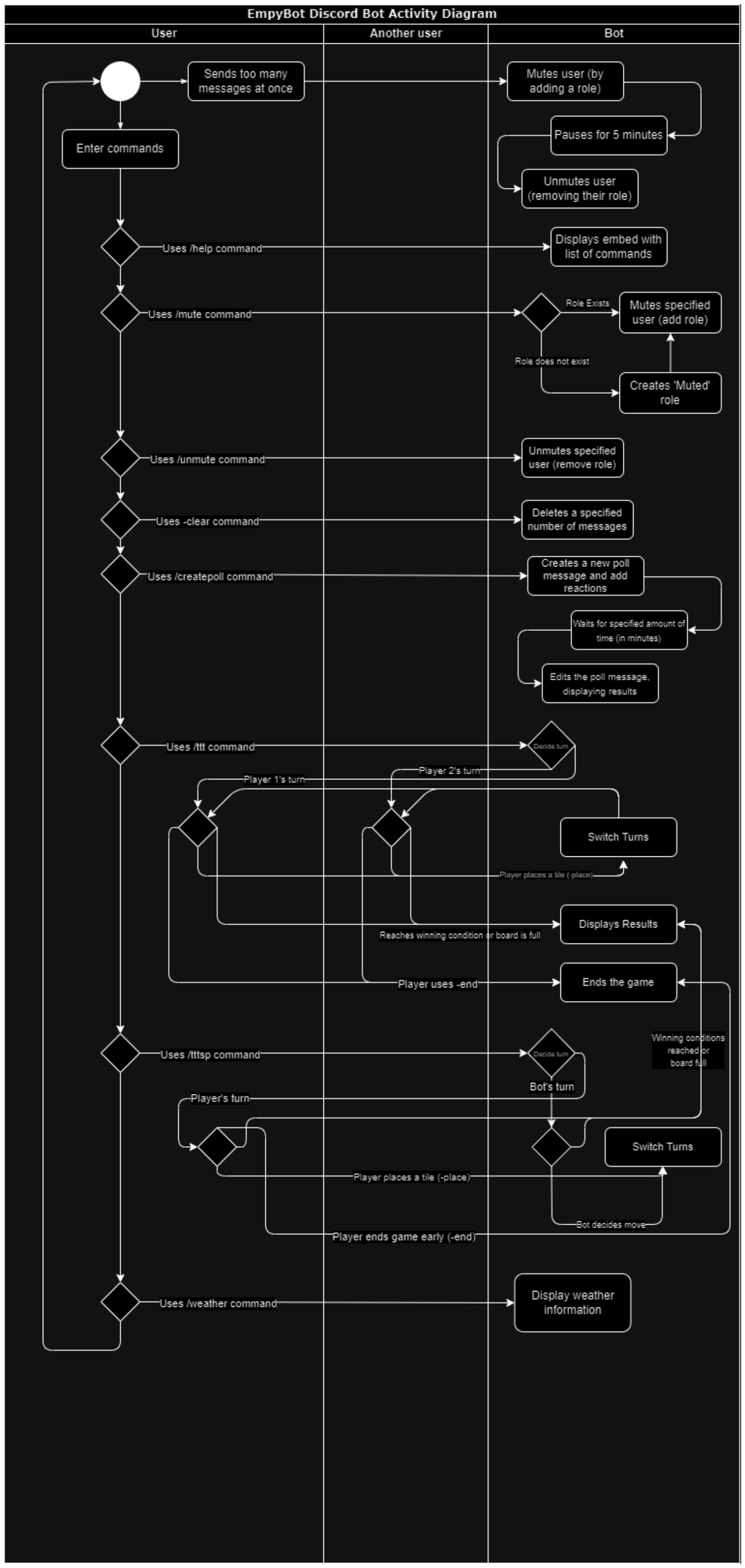
I. Class Diagram



II. Use Case Diagram



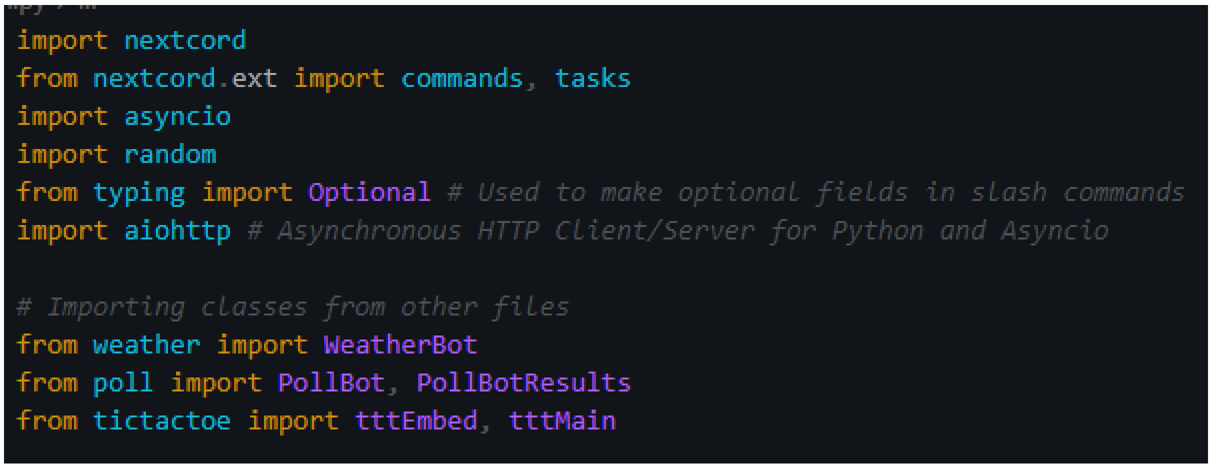
III. Activity Diagram


EmpyBot Discord Bot Activity Diagram
User
Another user
Bot
Sends too many messages at once
Mutes user (by adding a role)
Pauses for 5 minutes
Unmutes user (removing their role)
Enter commands
Uses /help command
Displays embed with list of commands
Uses /mute command
Role Exists
Mutes specified user (add role)
Role does not exist
Creates 'Muted' role
Uses /unmute command
Unmutes specified user (remove role)
Uses -clear command
Deletes a specified number of messages
Uses /createpoll command
Creates a new poll message and add reactions
Waits for specified amount of time (in minutes)
Edits the poll message, displaying results
Uses /ttt command
Decide turn
Player 2's turn
Player 1's turn
Switch Turns
Player places a tile (-place)
Displays Results
Reaches winning condition or board is full
Ends the game
Player uses -end
Winning conditions reached or board full
Uses /tttsp command
Decide turn
Bot's turn
Player's turn
Switch Turns
Player places a tile (-place)
Bot decides move
Player ends game early (-end)
Uses /weather command
Display weather information

Program Code

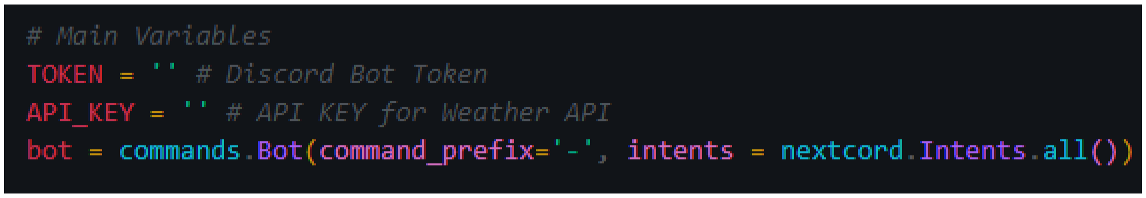
I. main.py

This file is the main file of this project that runs the bot. It contains most of the code that this program runs on including the commands, events, and other functions.



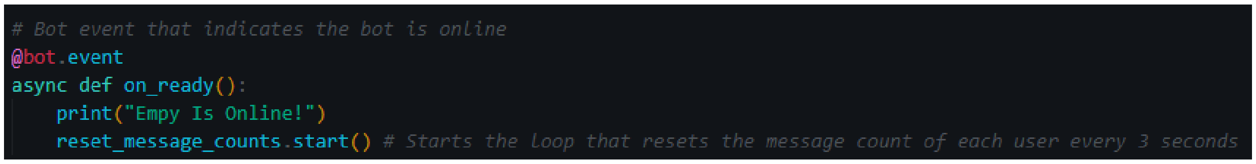
```python
import nextcord
from nextcord.ext import commands, tasks
import asyncio
import random
from typing import Optional # Used to make optional fields in slash commands
import aiohttp # Asynchronous HTTP Client/Server for Python and Asyncio

# Importing classes from other files
from weather import WeatherBot
from poll import PollBot, PollBotResults
from tictactoe import tttEmbed, tttMain
```

At the top of the file, I imported the required libraries, APIs, and modules, along with the classes from my other created files to be used within this program.



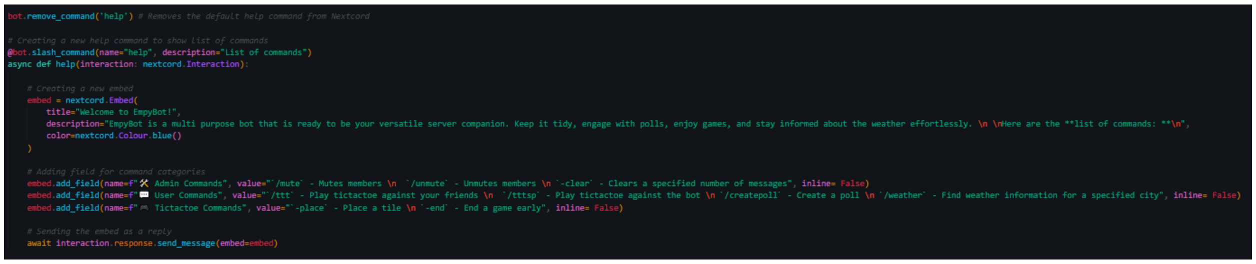
```python
# Main Variables
TOKEN = '' # Discord Bot Token
API_KEY = '' # API KEY for Weather API
bot = commands.Bot(command_prefix='-', intents = nextcord.Intents.all())
```

Next, I defined the main variables that includes the Discord bot token (which was taken from the Discord developer portal for my app), the weather API key from the API website, and created a new Discord bot instance by using the commands.Bot class from the nextcord library.



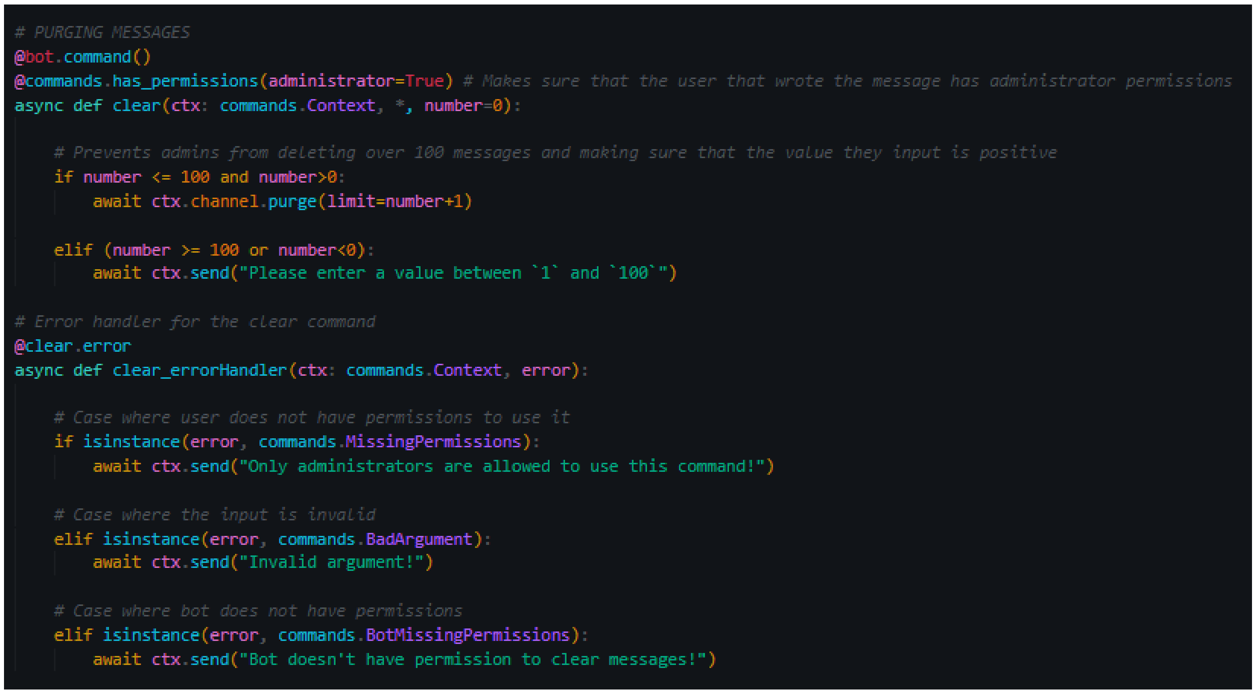
```python
# Bot event that indicates the bot is online
@bot.event
async def on_ready():
    print("Empy Is Online!")
    reset_message_counts.start() # Starts the loop that resets the message count of each user every 3 seconds
```

This @bot.event decorator runs the following asynchronous function when the Discord bot successfully connects to the Discord server. Inside the on_ready() function, the message "Empy Is Online!" is printed to the console, showing that the bot is now ready to be used. Additionally, there is a call to reset_message_counts.start(). This is to run a @tasks.loop decorator with a function named reset_message_counts that has been defined elsewhere in the code. This loop is designed for a spam detection mechanism, storing message counts for users in a dictionary and emptying the dictionary every 3 seconds.



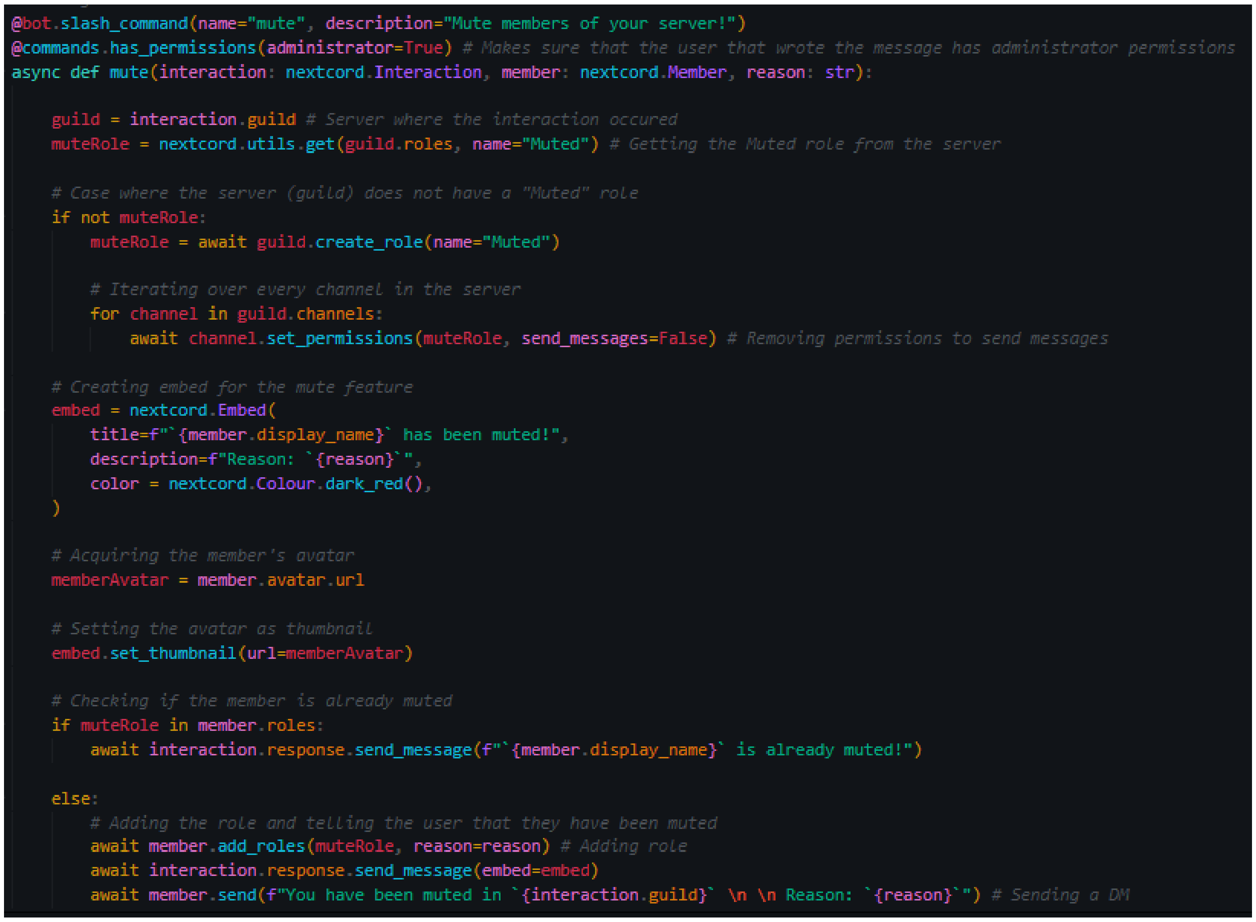
```python
bot.remove_command('help') # Removes the default help command from Nextcord

# Creating a new help command to show list of commands
@bot.slash_command(name="help", description="List of commands")
async def help(interaction: nextcord.Interaction):

    # Creating a new embed
    embed = nextcord.Embed(
        title="Welcome to EmpyBot!",
        description="EmpyBot is a multi purpose bot that is ready to be your versatile server companion. Keep it tidy, engage with polls, enjoy games, and stay informed about the weather effortlessly. \n \nHere are the **list of commands: **\n",
        color=nextcord.Colour.blue()
    )

    # Adding field for command categories
    embed.add_field(name=f"🛠 Admin Commands", value="`/mute` - Mutes members \n  `/unmute` - Unmutes members \n `-clear` - Clears a specified number of messages", inline= False)
    embed.add_field(name=f"💬 User Commands", value="`/ttt` - Play tictactoe against your friends \n  `/tttsp` - Play tictactoe against the bot \n `/createpoll` - Create a poll \n `/weather` - Find weather information for a specified city", inline= False)
    embed.add_field(name=f"🎮 Tictactoe Commands", value="`-place` - Place a tile \n `-end` - End a game early", inline= False)

    # Sending the embed as a reply
    await interaction.response.send_message(embed=embed)
```

Following the @bot.event decorator, I removed the default help command provided by Nextcord, because it only showed commands that uses the prefix and had an unsatisfactory design. Thus, I replaced it with a more modern slash command that displays a custom embed I created which shows the list of all commands.

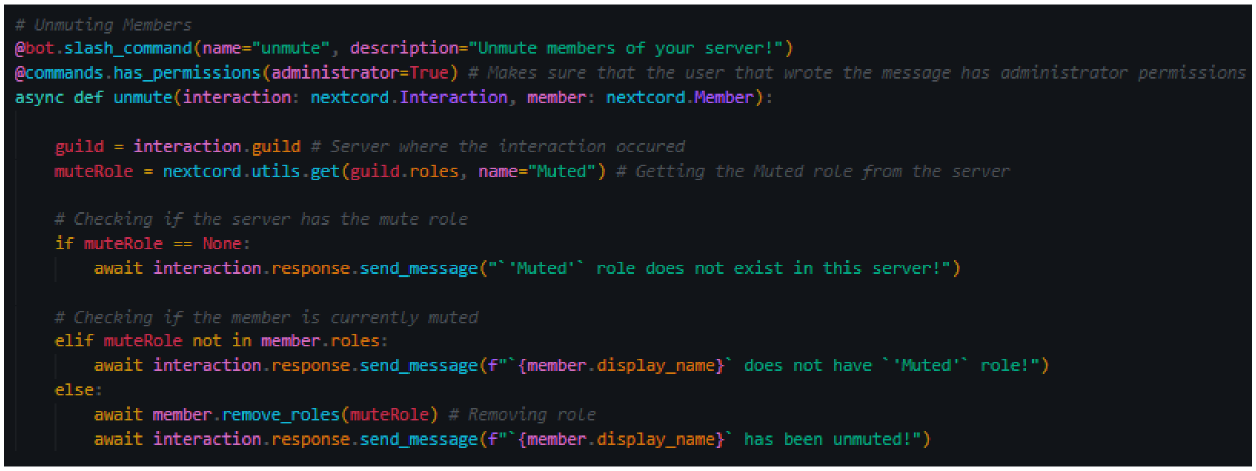


```python
# PURGING MESSAGES
@bot.command()
@commands.has_permissions(administrator=True) # Makes sure that the user that wrote the message has administrator permissions
async def clear(ctx: commands.Context, *, number=0):

    # Prevents admins from deleting over 100 messages and making sure that the value they input is positive
    if number <= 100 and number>0:
        await ctx.channel.purge(limit=number+1)

    elif (number >= 100 or number<0):
        await ctx.send("Please enter a value between `1` and `100`")

# Error handler for the clear command
@clear.error
async def clear_errorHandler(ctx: commands.Context, error):

    # Case where user does not have permissions to use it
    if isinstance(error, commands.MissingPermissions):
        await ctx.send("Only administrators are allowed to use this command!")

    # Case where the input is invalid
    elif isinstance(error, commands.BadArgument):
        await ctx.send("Invalid argument!")

    # Case where bot does not have permissions
    elif isinstance(error, commands.BotMissingPermissions):
        await ctx.send("Bot doesn't have permission to clear messages!")
```

This section is the start of the main section of the code. It is also the start of the server moderation features group in the code, which mainly use the @commands.has_permissions(administrator=True) decorator to make sure that only administrators could use the commands. This feature is a command that allows users to use the "-clear" command, followed by an integer between 1 and 100 (inclusive), to clear a specified number of messages in a channel. It takes in the context parameter (containing information about message, server, and channel where command took place) and a number with default value of 0. The asterisk signifies that all arguments following it must be keyword arguments. It checks if the input number follows the conditions. If it does, it will use the purge method to clear messages, otherwise it would send a message to the channel. This command also has an error handler which checks the type of errors and gives responses according to the type of error.



```
@bot.slash_command(name="mute", description="Mute members of your server!")
@commands.has_permissions(administrator=True) # Makes sure that the user that wrote the message has administrator permissions
async def mute(interaction: nextcord.Interaction, member: nextcord.Member, reason: str):

    guild = interaction.guild # Server where the interaction occured
    muteRole = nextcord.utils.get(guild.roles, name="Muted") # Getting the Muted role from the server

    # Case where the server (guild) does not have a "Muted" role
    if not muteRole:
        muteRole = await guild.create_role(name="Muted")

        # Iterating over every channel in the server
        for channel in guild.channels:
            await channel.set_permissions(muteRole, send_messages=False) # Removing permissions to send messages

    # Creating embed for the mute feature
    embed = nextcord.Embed(
        title=f"`{member.display_name}` has been muted!",
        description=f"Reason: `{reason}`",
        color = nextcord.Colour.dark_red(),
    )

    # Acquiring the member's avatar
    memberAvatar = member.avatar.url

    # Setting the avatar as thumbnail
    embed.set_thumbnail(url=memberAvatar)

    # Checking if the member is already muted
    if muteRole in member.roles:
        await interaction.response.send_message(f"`{member.display_name}` is already muted!")

    else:
        # Adding the role and telling the user that they have been muted
        await member.add_roles(muteRole, reason=reason) # Adding role
        await interaction.response.send_message(embed=embed)
        await member.send(f"You have been muted in `{interaction.guild}` \n \n Reason: `{reason}`") # Sending a DM
```

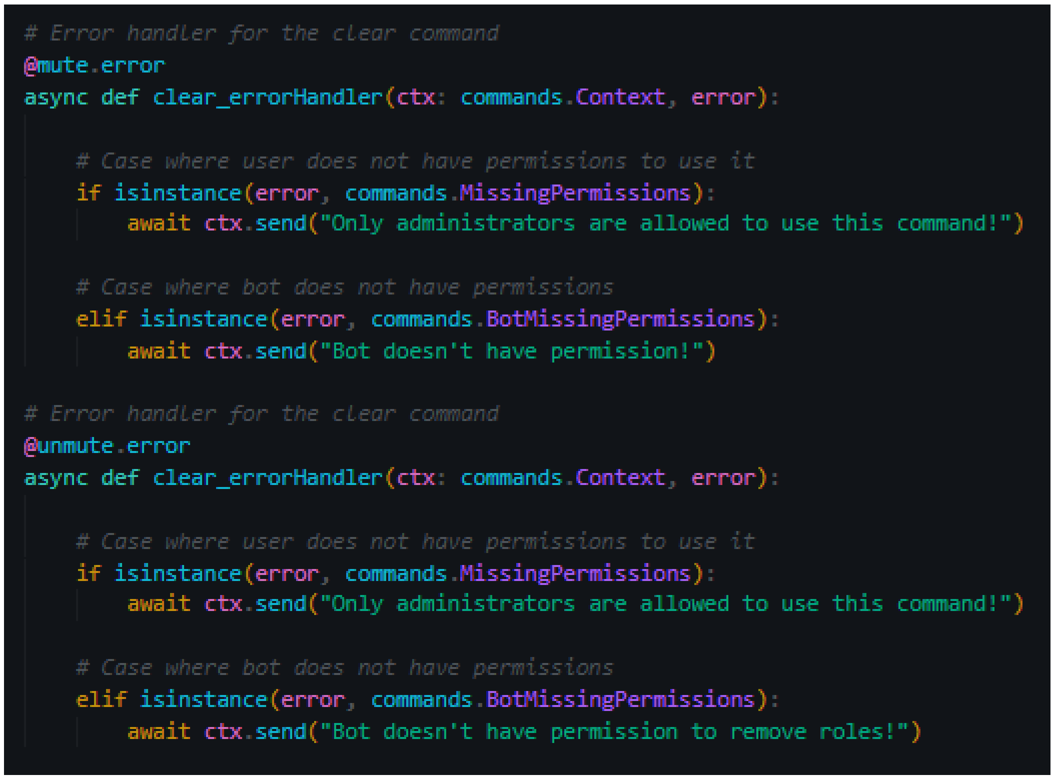


```
# Unmuting Members
@bot.slash_command(name="unmute", description="Unmute members of your server!")
@commands.has_permissions(administrator=True) # Makes sure that the user that wrote the message has administrator permissions
async def unmute(interaction: nextcord.Interaction, member: nextcord.Member):

    guild = interaction.guild # Server where the interaction occured
    muteRole = nextcord.utils.get(guild.roles, name="Muted") # Getting the Muted role from the server

    # Checking if the server has the mute role
    if muteRole == None:
        await interaction.response.send_message("`'Muted'` role does not exist in this server!")

    # Checking if the member is currently muted
    elif muteRole not in member.roles:
        await interaction.response.send_message(f"`{member.display_name}` does not have `'Muted'` role!")
    else:
        await member.remove_roles(muteRole) # Removing role
        await interaction.response.send_message(f"`{member.display_name}` has been unmuted!")
```

These are the mute and unmute slash commands that allow server administrators to mute and unmute server members.

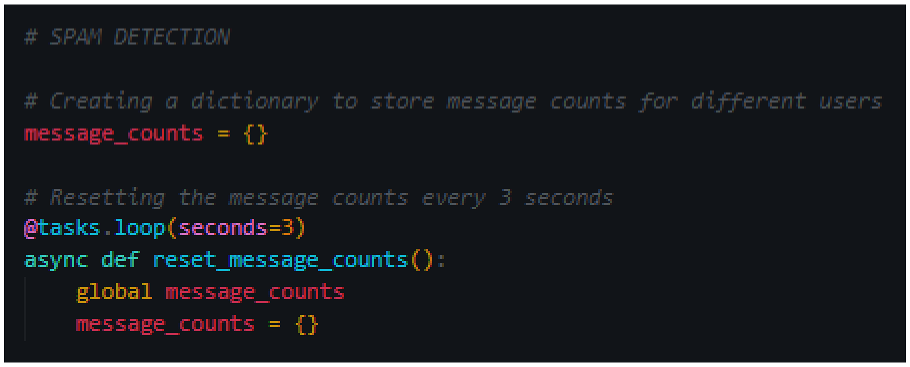
The mute function first creates a guild object to access the attributes of the server and get a role called “Muted” from the server. It then checks whether the role exists on the server or not. If it doesn’t it would create a role that sets the permission to send messages to every channel to false. Next, it would create an embed that tags the muted member and shows the reason, using the member’s avatar as the thumbnail. It then uses another set of if-else statements to check whether the role has been previously added to the user (check if user is muted). If the user is not muted, the bot would add the role to the specified user, muting them.

The unmute function is quite like the mute function; however, it is shorter as it only removes the role. After creating object and getting the ‘Muted’ role, it runs a set of if-else statements to check whether the mute role exists and check if the member is even muted in the first place. It would then give responses according to the conditions.



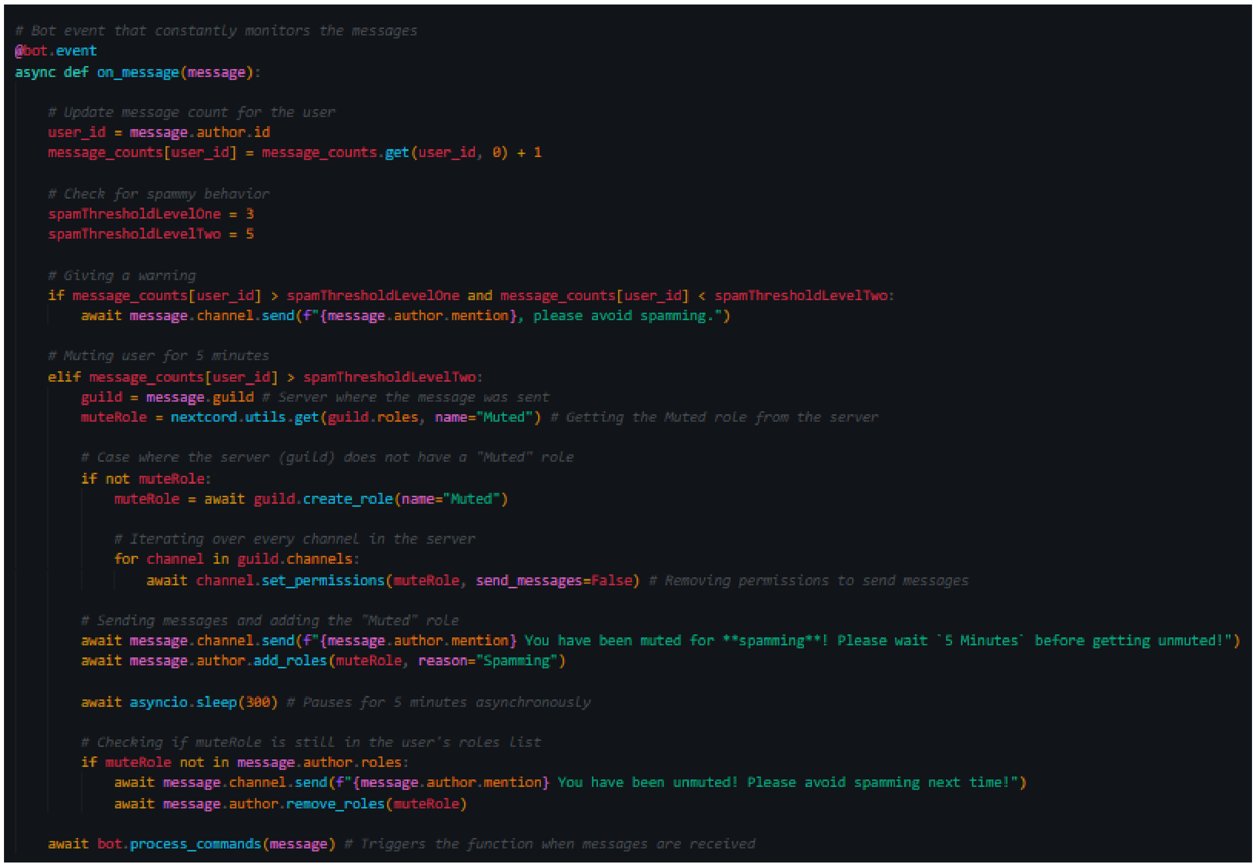
```python
# Error handler for the clear command
@mute.error
async def clear_errorHandler(ctx: commands.Context, error):

    # Case where user does not have permissions to use it
    if isinstance(error, commands.MissingPermissions):
        await ctx.send("Only administrators are allowed to use this command!")

    # Case where bot does not have permissions
    elif isinstance(error, commands.BotMissingPermissions):
        await ctx.send("Bot doesn't have permission!")

# Error handler for the clear command
@unmute.error
async def clear_errorHandler(ctx: commands.Context, error):

    # Case where user does not have permissions to use it
    if isinstance(error, commands.MissingPermissions):
        await ctx.send("Only administrators are allowed to use this command!")

    # Case where bot does not have permissions
    elif isinstance(error, commands.BotMissingPermissions):
        await ctx.send("Bot doesn't have permission to remove roles!")
```

These are the error handlers for the mute and unmute commands, which are mainly related to missing permissions.



```python
# SPAM DETECTION

# Creating a dictionary to store message counts for different users
message_counts = {}

# Resetting the message counts every 3 seconds
@tasks.loop(seconds=3)
async def reset_message_counts():
    global message_counts
    message_counts = {}
```

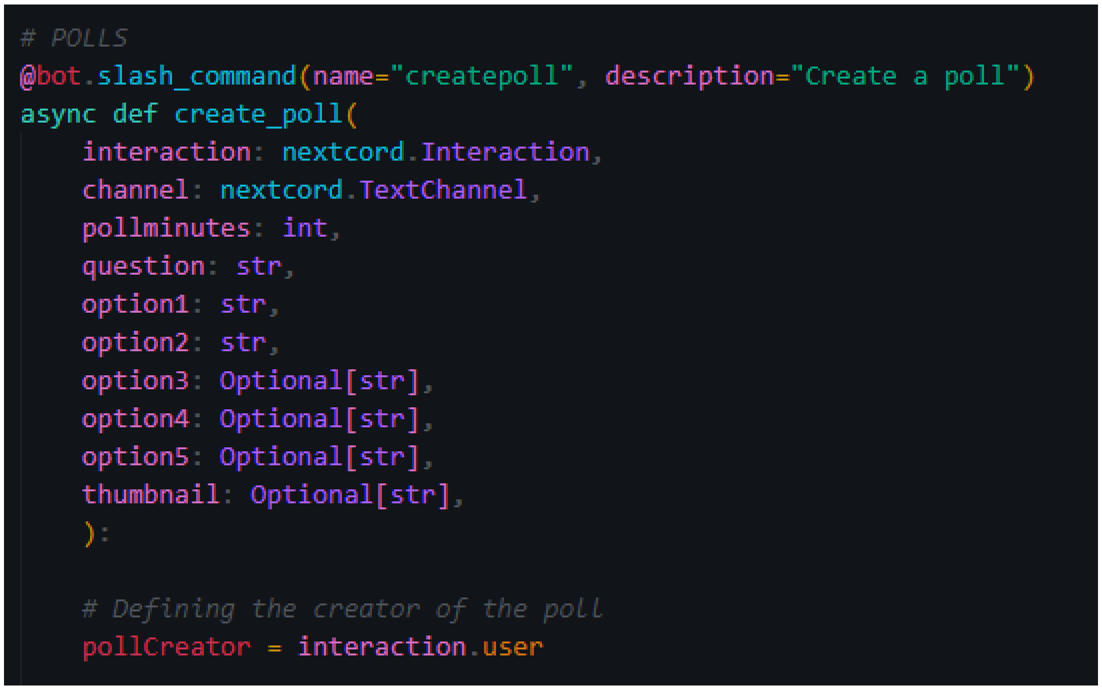
This is the @tasks.loop decorator that was mentioned earlier.



```python
# Bot event that constantly monitors the messages
@bot.event
async def on_message(message):

    # Update message count for the user
    user_id = message.author.id
    message_counts[user_id] = message_counts.get(user_id, 0) + 1

    # Check for spammy behavior
    spamThresholdLevelOne = 3
    spamThresholdLevelTwo = 5

    # Giving a warning
    if message_counts[user_id] > spamThresholdLevelOne and message_counts[user_id] < spamThresholdLevelTwo:
        await message.channel.send(f"{message.author.mention}, please avoid spamming.")

    # Muting user for 5 minutes
    elif message_counts[user_id] > spamThresholdLevelTwo:
        guild = message.guild # Server where the message was sent
        muteRole = nextcord.utils.get(guild.roles, name="Muted") # Getting the Muted role from the server

        # Case where the server (guild) does not have a "Muted" role
        if not muteRole:
            muteRole = await guild.create_role(name="Muted")

            # Iterating over every channel in the server
            for channel in guild.channels:
                await channel.set_permissions(muteRole, send_messages=False) # Removing permissions to send messages

        # Sending messages and adding the "Muted" role
        await message.channel.send(f"{message.author.mention} You have been muted for **spamming**! Please wait `5 Minutes` before getting unmuted!")
        await message.author.add_roles(muteRole, reason="Spamming")

        await asyncio.sleep(300) # Pauses for 5 minutes asynchronously

        # Checking if muteRole is still in the user's roles list
        if muteRole not in message.author.roles:
            await message.channel.send(f"{message.author.mention} You have been unmuted! Please avoid spamming next time!")
            await message.author.remove_roles(muteRole)

    await bot.process_commands(message) # Triggers the function when messages are received
```

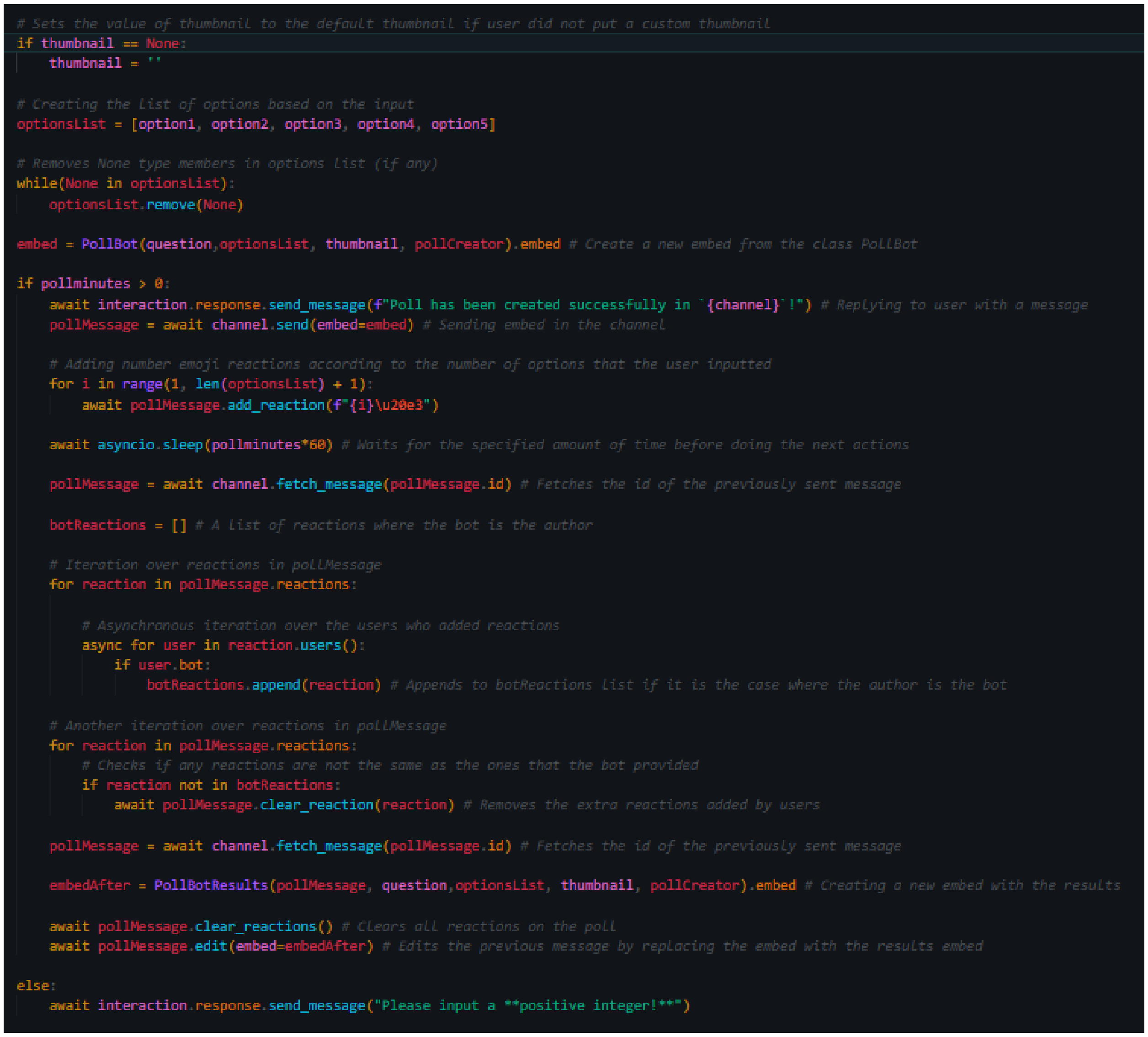
This is the spam detection function which uses a @bot.event decorator with an asynchronous function that takes in messages as input. It first takes the message author's user ID and retrieves the current message count for the specified user ID from the message_counts dictionary. If the user ID is not already a key in the dictionary, it returns 0 as the default value. Then, the values for the number of messages to be detected as spam are set.

A set of if-else statements is created to check if the user surpassed the first and second threshold. If they only surpassed the first threshold, the bot only sends a message in the channel to warn them. But if they surpassed the second threshold, the bot would try to mute them using the same method as the mute slash command (adding the role). After that, I used ‘await asyncio.sleep(300)’ to cause the coroutine in which it is placed to pause or "sleep" for 5 minutes before continuing with the next set of instructions. After 5 minutes, the bot would check if the role is still there in case an administrator has removed it, and if it’s still there, the bot would remove it.



```
# POLLS
@bot.slash_command(name="createpoll", description="Create a poll")
async def create_poll(
    interaction: nextcord.Interaction,
    channel: nextcord.TextChannel,
    pollminutes: int,
    question: str,
    option1: str,
    option2: str,
    option3: Optional[str],
    option4: Optional[str],
    option5: Optional[str],
    thumbnail: Optional[str],
    ):

    # Defining the creator of the poll
    pollCreator = interaction.user
```

After that, we have the code for creating custom polls. It takes in a bunch of required inputs and some optional inputs. The interaction parameter represents an object of the nextcord.Interaction class, which represents an interaction with a Discord bot. Interactions occur when a user interacts with a bot's commands, buttons, or other interactive elements. The Interaction class encapsulates information about the interaction, allowing the bot to respond accordingly. The channel input gives the user a list of the available channels in the server for them to send the poll to. The pollminutes input allows users to input the duration of the poll in minutes (integers). The question and options are for the main part of the poll, containing the question users want to ask and the options they provide. Lastly, the thumbnail input is an optional input field that allows users to set custom thumbnails for the embed. The first line of the function defines the pollCreator variable to be the user that interacted with the command.



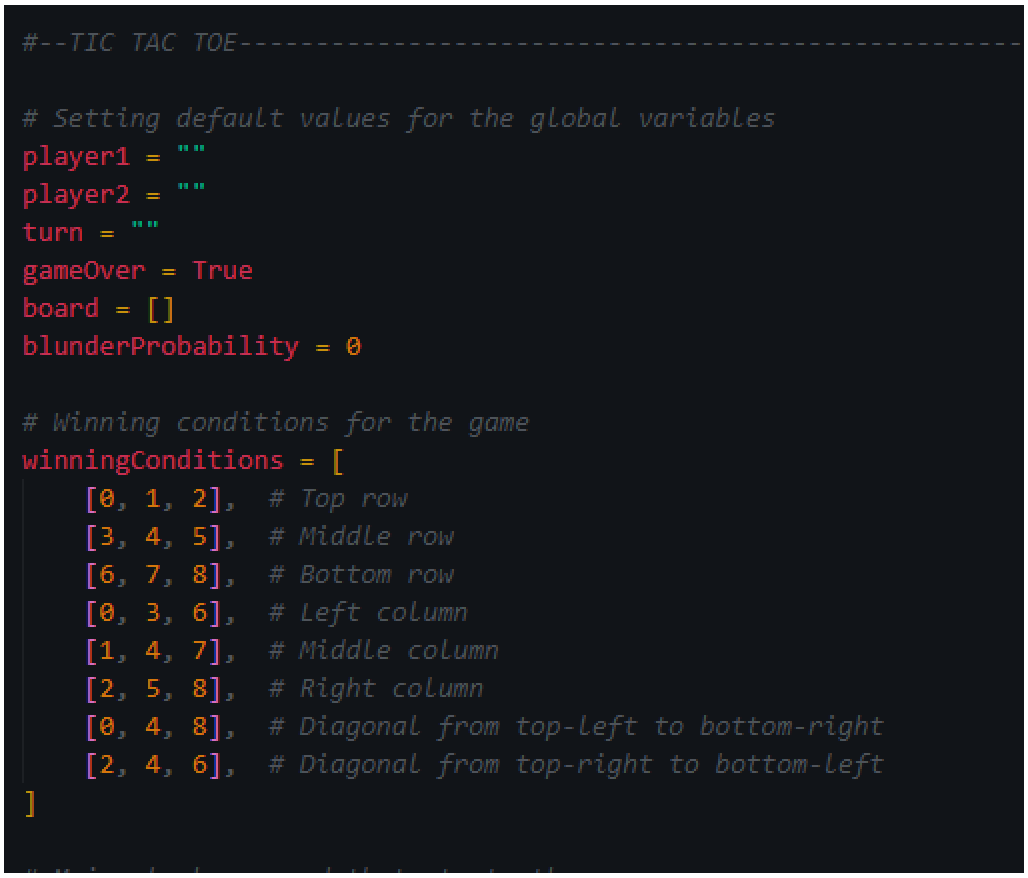
```python
# Sets the value of thumbnail to the default thumbnail if user did not put a custom thumbnail
if thumbnail == None:
    thumbnail = ''

# Creating the list of options based on the input
optionsList = [option1, option2, option3, option4, option5]

# Removes None type members in options list (if any)
while(None in optionsList):
    optionsList.remove(None)

embed = PollBot(question,optionsList, thumbnail, pollCreator).embed # Create a new embed from the class PollBot

if pollminutes > 0:
    await interaction.response.send_message(f"Poll has been created successfully in `{channel}`!") # Replying to user with a message
    pollMessage = await channel.send(embed=embed) # Sending embed in the channel

    # Adding number emoji reactions according to the number of options that the user inputted
    for i in range(1, len(optionsList) + 1):
        await pollMessage.add_reaction(f"{i}\u20e3")

    await asyncio.sleep(pollminutes*60) # Waits for the specified amount of time before doing the next actions

    pollMessage = await channel.fetch_message(pollMessage.id) # Fetches the id of the previously sent message

    botReactions = [] # A list of reactions where the bot is the author

    # Iteration over reactions in pollMessage
    for reaction in pollMessage.reactions:

        # Asynchronous iteration over the users who added reactions
        async for user in reaction.users():
            if user.bot:
                botReactions.append(reaction) # Appends to botReactions list if it is the case where the author is the bot

    # Another iteration over reactions in pollMessage
    for reaction in pollMessage.reactions:
        # Checks if any reactions are not the same as the ones that the bot provided
        if reaction not in botReactions:
            await pollMessage.clear_reaction(reaction) # Removes the extra reactions added by users

    pollMessage = await channel.fetch_message(pollMessage.id) # Fetches the id of the previously sent message

    embedAfter = PollBotResults(pollMessage, question,optionsList, thumbnail, pollCreator).embed # Creating a new embed with the results

    await pollMessage.clear_reactions() # Clears all reactions on the poll
    await pollMessage.edit(embed=embedAfter) # Edits the previous message by replacing the embed with the results embed

else:
    await interaction.response.send_message("Please input a **positive integer!**")
```

Next, it checks if the user inputted a URL for a custom thumbnail and sets the thumbnail URL to empty (no thumbnail) if they didn't fill out the field. A list of the options is then created. A while loop is initiated after that to remove an element of None type while it is within the list (from unfilled optional fields).

Then, I used an instance of the Pollbot class (embed attribute), from another file, to generate an embed for the poll, using the questions, list of options, thumbnail, and poll creator as parameters.

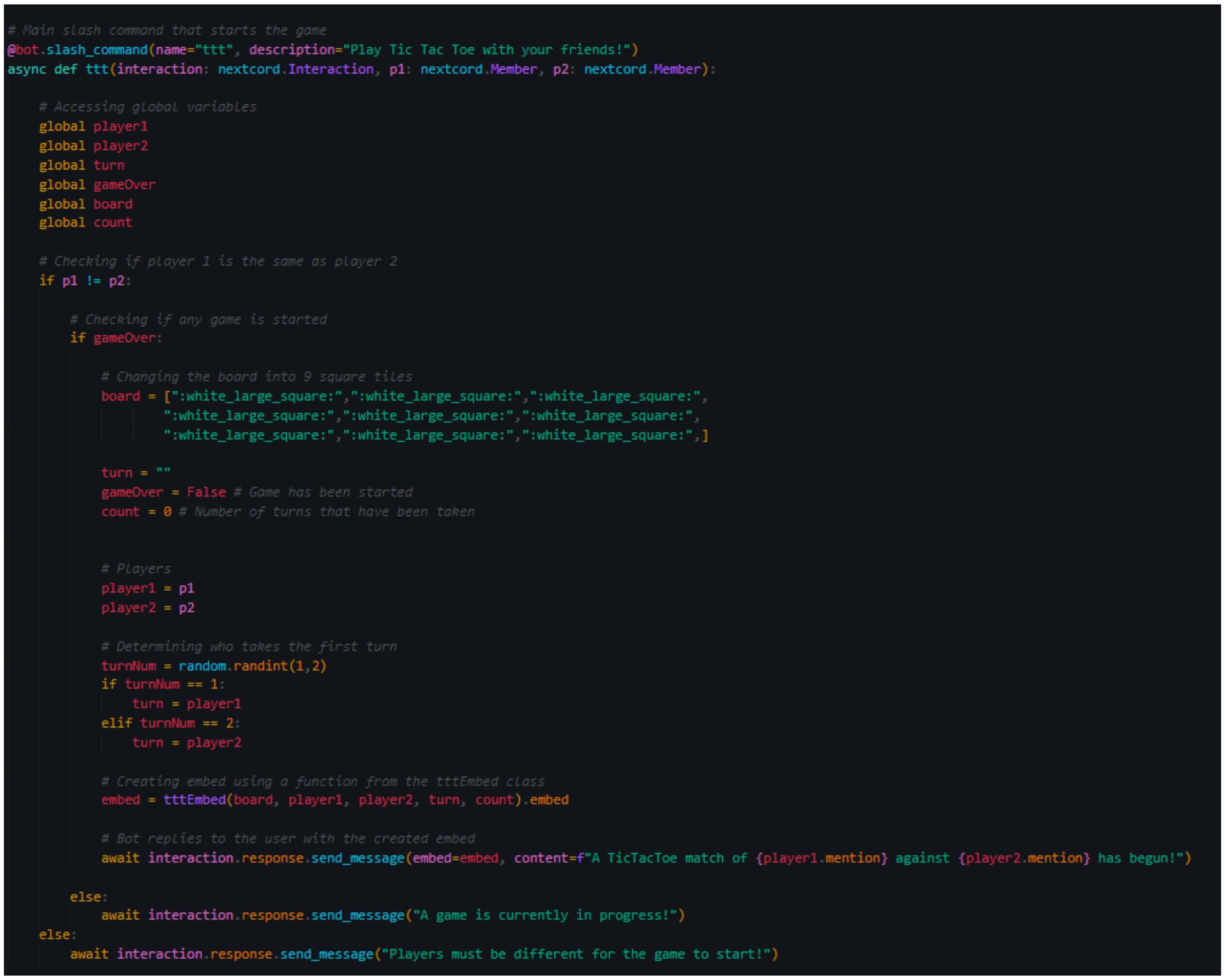
A set of if-else statements are then initiated to check whether the pollminutes input is a positive integer. If it is, the program would continue normally and send a reply to tell the user that the poll has been successfully created. The previously created embed is then sent to the specified channel and the bot would add number emoji reactions based on the number of options so that users could react to it. I used the asyncio.sleep again to wait for the specified number of minutes before showing results.

After that, the program would fetch the poll message's ID and create a new list for the reactions added by the bot. A for loop is then initiated to add the reactions that the bot added to the list. Another for loop is then initiated to iterate over the poll message's reactions list to check if any reactions are not within the list of bot reactions to remove any extra reactions added by the users. The program would then retrieve the poll message ID again after removing the extra reactions and a new embed would be created using the embed attribute of the PollBotResults class, which takes in the poll message id, question, list of options, thumbnail and poll creator as parameters. The bot would then clear every reaction and display the result of the poll, ending the poll.



```python
#--TIC TAC TOE-----------------------------------------------

# Setting default values for the global variables
player1 = ""
player2 = ""
turn = ""
gameOver = True
board = []
blunderProbability = 0

# Winning conditions for the game
winningConditions = [
    [0, 1, 2],  # Top row
    [3, 4, 5],  # Middle row
    [6, 7, 8],  # Bottom row
    [0, 3, 6],  # Left column
    [1, 4, 7],  # Middle column
    [2, 5, 8],  # Right column
    [0, 4, 8],  # Diagonal from top-left to bottom-right
    [2, 4, 6],  # Diagonal from top-right to bottom-left
]
```

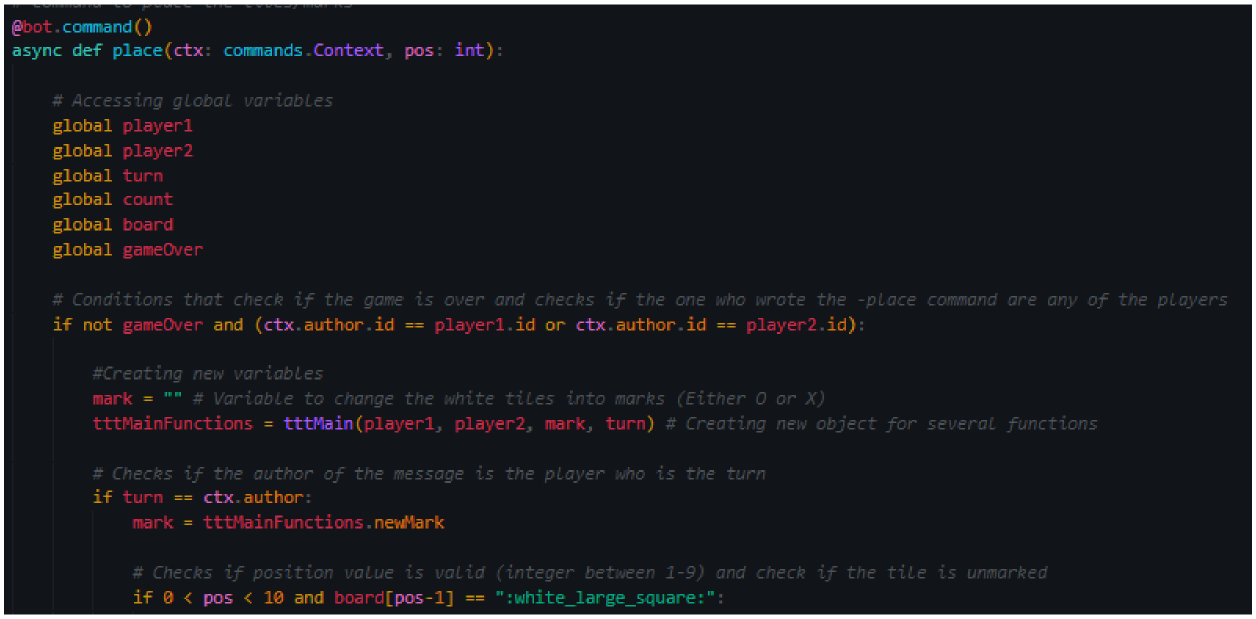
Now we are moving on to the games section of the code. To start the tic-tac-toe section, I defined a few variables for the games. Player 1 and player 2 are the variables which contain the players of the game, the turn variable is the player who is currently taking the turn, the gameOver variable is a boolean that tells whether game is started or not, board is the main board of the tic-tac-toe, blunderProbability is the probability in which the bot is going to make blunders in single player mode, and the winning conditions are positions of marks which give a winning condition.



```python
# Main slash command that starts the game
@bot.slash_command(name="ttt", description="Play Tic Tac Toe with your friends!")
async def ttt(interaction: nextcord.Interaction, p1: nextcord.Member, p2: nextcord.Member):

    # Accessing global variables
    global player1
    global player2
    global turn
    global gameOver
    global board
    global count

    # Checking if player 1 is the same as player 2
    if p1 != p2:

        # Checking if any game is started
        if gameOver:

            # Changing the board into 9 square tiles
            board = [":white_large_square:",":white_large_square:",":white_large_square:",
                    ":white_large_square:",":white_large_square:",":white_large_square:",
                    ":white_large_square:",":white_large_square:",":white_large_square:",]

            turn = ""
            gameOver = False # Game has been started
            count = 0 # Number of turns that have been taken


            # Players
            player1 = p1
            player2 = p2

            # Determining who takes the first turn
            turnNum = random.randint(1,2)
            if turnNum == 1:
                turn = player1
            elif turnNum == 2:
                turn = player2

            # Creating embed using a function from the tttEmbed class
            embed = tttEmbed(board, player1, player2, turn, count).embed

            # Bot replies to the user with the created embed
            await interaction.response.send_message(embed=embed, content=f"A TicTacToe match of {player1.mention} against {player2.mention} has begun!")

        else:
            await interaction.response.send_message("A game is currently in progress!")
    else:
        await interaction.response.send_message("Players must be different for the game to start!")
```

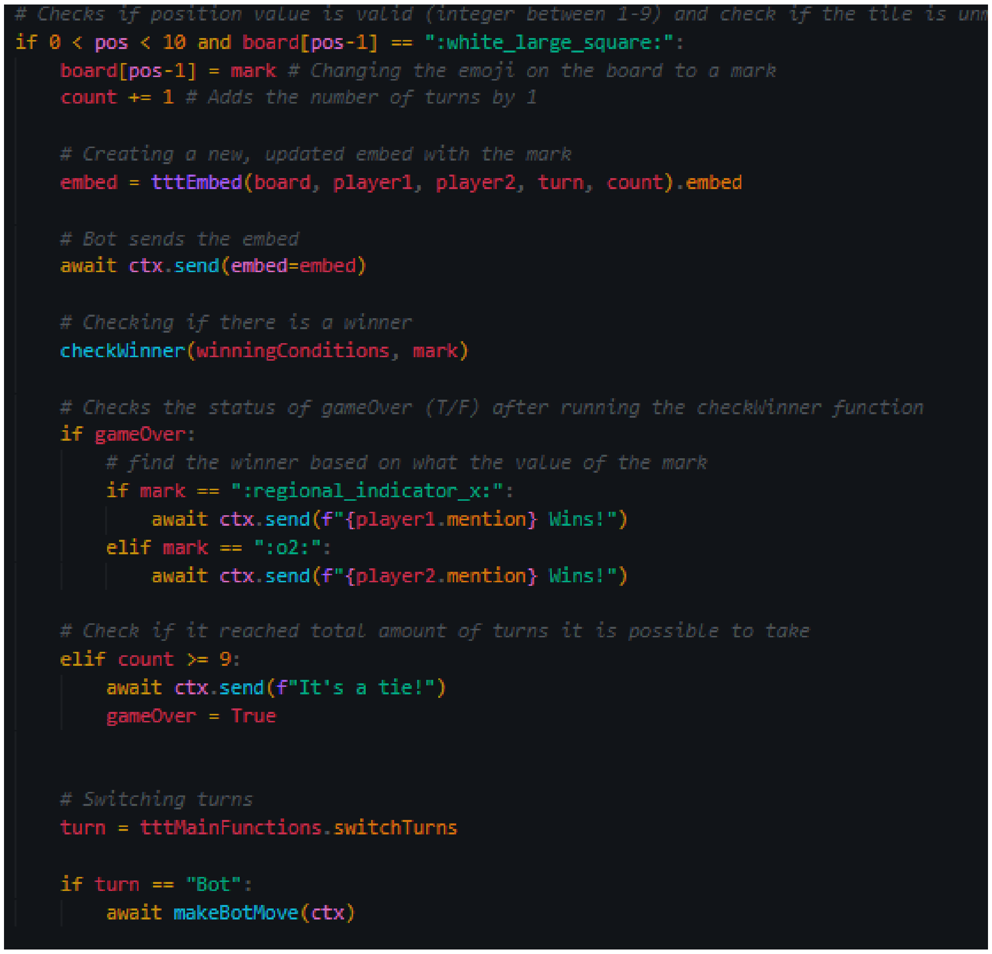
This is the code that starts the game for the player versus player mode. It takes in two players as the parameters.

First, it accesses the global variables that were defined previously and uses if-else statements to make sure player 1 and player 2 are not the same user, and make sure that a game is not ongoing. When the proper conditions are met, the game would start, changing the bot into 9 white tiles and setting gameOver to False. The player to get the first turn is decided by using the random module's randint method to generate a number of either 1 or 2 to be used in an if-else statement to decide the first player. Then, an embed is created by using the embed attribute of the tttEmbed class I created in another file. To end the initialization of the game, a message is sent, containing the embed and a text message that pings the players.



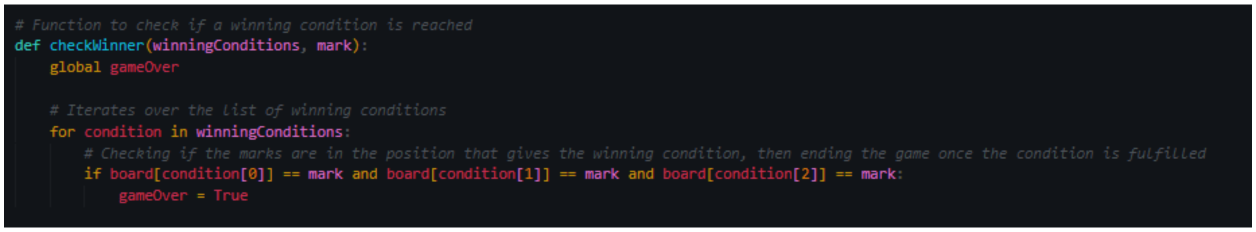
```
@bot.command()
async def place(ctx: commands.Context, pos: int):

    # Accessing global variables
    global player1
    global player2
    global turn
    global count
    global board
    global gameOver

    # Conditions that check if the game is over and checks if the one who wrote the -place command are any of the players
    if not gameOver and (ctx.author.id == player1.id or ctx.author.id == player2.id):

        #Creating new variables
        mark = "" # Variable to change the white tiles into marks (Either O or X)
        tttMainFunctions = tttMain(player1, player2, mark, turn) # Creating new object for several functions

        # Checks if the author of the message is the player who is the turn
        if turn == ctx.author:
            mark = tttMainFunctions.newMark

            # Checks if position value is valid (integer between 1-9) and check if the tile is unmarked
            if 0 < pos < 10 and board[pos-1] == ":white_large_square:":
```

The next command for this game is the place command which uses the bot's prefix to activate and takes position (an integer) as input. This command is used for the player to place their mark tiles on their turn. The code first accesses the global variables and uses a bunch of if-else statements to check for several conditions. The conditions to be met are: the game is not over, the one who wrote the command is a player, it must be the user's turn for the command to work, the position inputted must be between 0 and 10 (1-9) and the position on the board must be a large white tile (unmarked). Additionally, two new variables are introduced within this function: mark and tttMainFunctions which is an object created from the tttMain class. The mark will then be changed by using the newMark attribute of the tttMain class based on who is currently taking the turn.



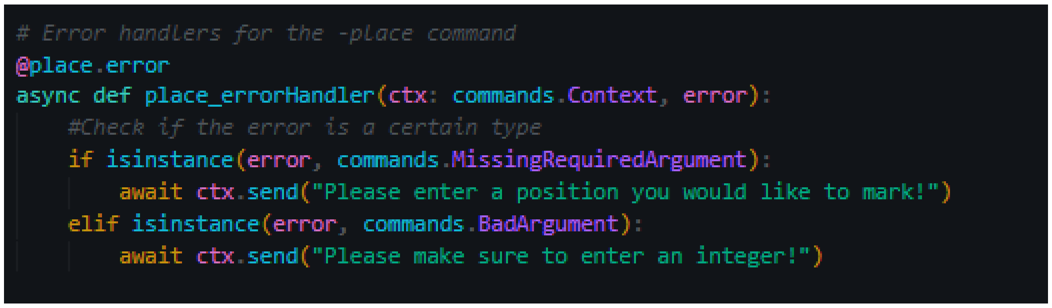
```
# Checks if position value is valid (integer between 1-9) and check if the tile is unm
if 0 < pos < 10 and board[pos-1] == ":white_large_square:":
    board[pos-1] = mark # Changing the emoji on the board to a mark
    count += 1 # Adds the number of turns by 1

    # Creating a new, updated embed with the mark
    embed = tttEmbed(board, player1, player2, turn, count).embed

    # Bot sends the embed
    await ctx.send(embed=embed)

    # Checking if there is a winner
    checkWinner(winningConditions, mark)

    # Checks the status of gameOver (T/F) after running the checkWinner function
    if gameOver:
        # find the winner based on what the value of the mark
        if mark == ":regional_indicator_x:":
            await ctx.send(f"{player1.mention} Wins!")
        elif mark == ":o2:":
            await ctx.send(f"{player2.mention} Wins!")

    # Check if it reached total amount of turns it is possible to take
    elif count >= 9:
        await ctx.send(f"It's a tie!")
        gameOver = True


    # Switching turns
    turn = tttMainFunctions.switchTurns

    if turn == "Bot":
        await makeBotMove(ctx)
```

This is the code that is executed once the conditions are fulfilled. The bot will first replace the specified position of the board with a mark of the player that is taking the current turn and add the count variable by 1 to keep track of how many turns have been taken. Next, a new embed with the updated board and turn would be created using the same class and attribute as the one in the initializing function. After sending the embed, the code proceeds to check if there is any winner using the checkWinner function (shown in the screenshot below). If it is a case where there is a winner, the program would check the current mark and send a message to announce the winner. If the count reaches 9 (highest possible number of turns taken in tic-tac-toe), the game will end in a draw. The last portion of this part of the program shows the process of switching turns using the switchTurns method of the tttMain class. It then checks if the current player taking the turn is “Bot”, which is the bot itself. If the other player is the bot (case where user plays single player mode), the program would continue with the makeBotMove function which would be explain later on in this report.



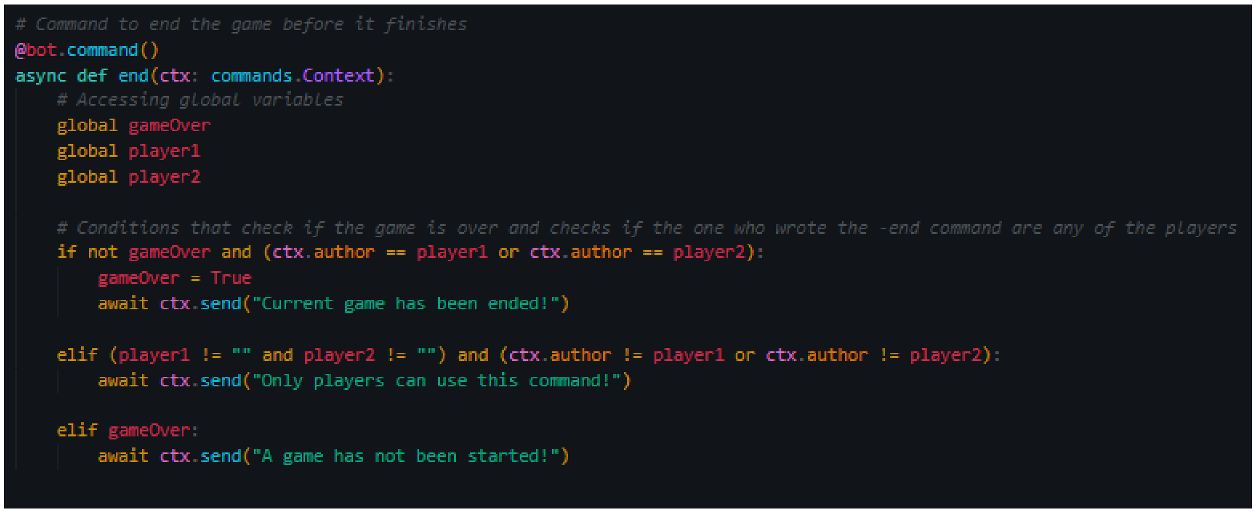
```python
# Function to check if a winning condition is reached
def checkWinner(winningConditions, mark):
    global gameOver

    # Iterates over the list of winning conditions
    for condition in winningConditions:
        # Checking if the marks are in the position that gives the winning condition, then ending the game once the condition is fulfilled
        if board[condition[0]] == mark and board[condition[1]] == mark and board[condition[2]] == mark:
            gameOver = True
```

This function checks if there is any winner in the game. It iterates over each condition of the winningConditions list and compares it with the current state of the board. If there is any line of the same marks formed, matching the winning condition, the game will end.



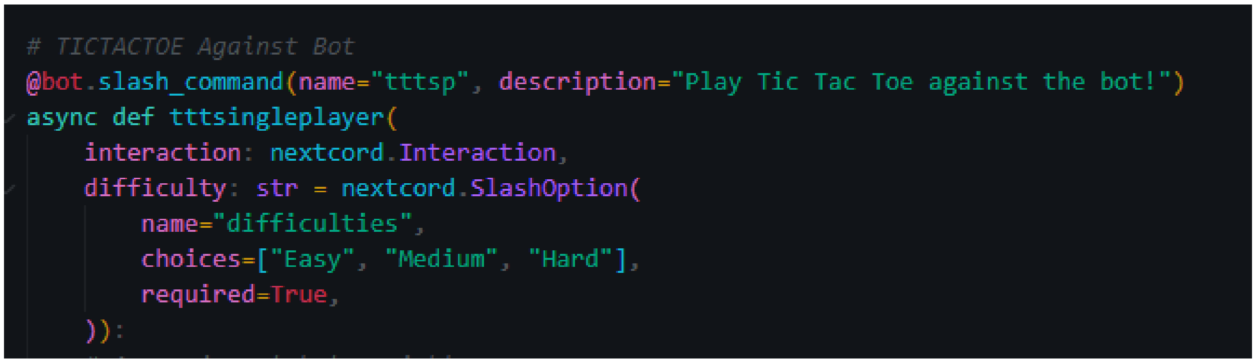
```python
# Error handlers for the -place command
@place.error
async def place_errorHandler(ctx: commands.Context, error):
    #Check if the error is a certain type
    if isinstance(error, commands.MissingRequiredArgument):
        await ctx.send("Please enter a position you would like to mark!")
    elif isinstance(error, commands.BadArgument):
        await ctx.send("Please make sure to enter an integer!")
```

The place command also has its own error handler for cases where it’s missing an argument or receiving an invalid argument.



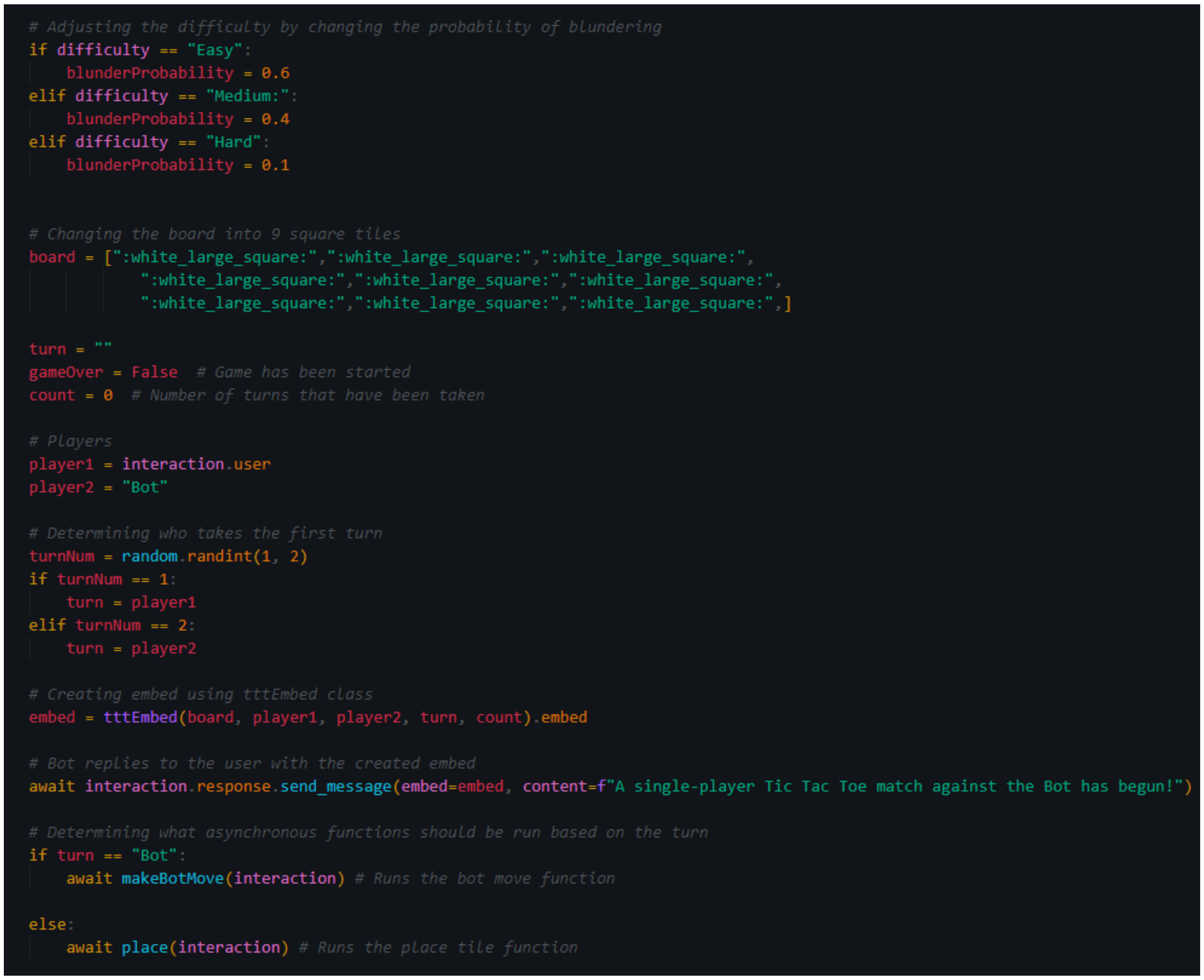
```python
# Command to end the game before it finishes
@bot.command()
async def end(ctx: commands.Context):
    # Accessing global variables
    global gameOver
    global player1
    global player2

    # Conditions that check if the game is over and checks if the one who wrote the -end command are any of the players
    if not gameOver and (ctx.author == player1 or ctx.author == player2):
        gameOver = True
        await ctx.send("Current game has been ended!")

    elif (player1 != "" and player2 != "") and (ctx.author != player1 or ctx.author != player2):
        await ctx.send("Only players can use this command!")

    elif gameOver:
        await ctx.send("A game has not been started!")
```

If a player wants to end their match early, there is a command to do just that. This asynchronous function uses an if-else statement to check for a few conditions. In order to end the game, there should currently be an ongoing game and the user that wrote the command must be either one of the two players playing the current game.



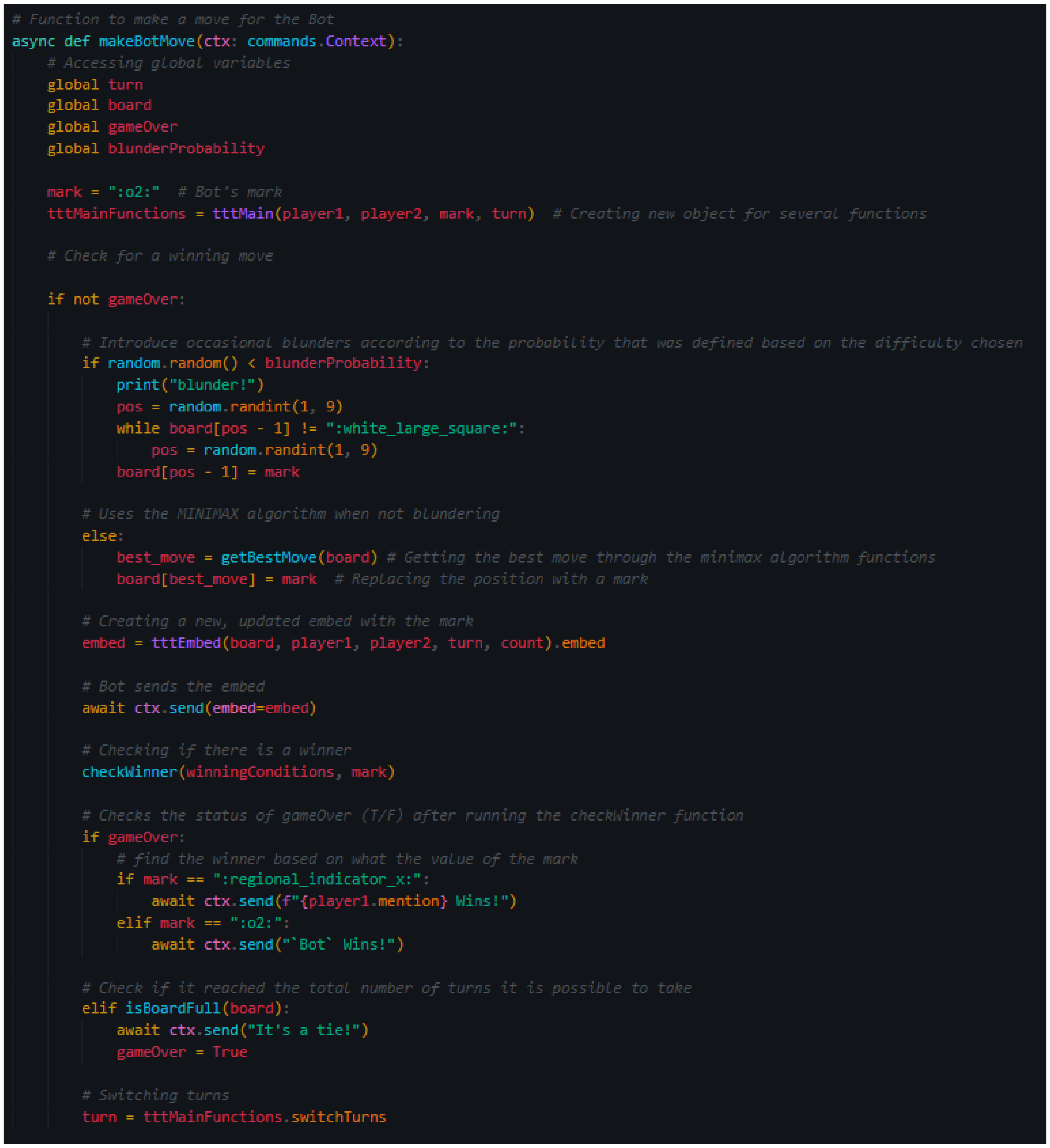
```python
# TICTACTOE Against Bot
@bot.slash_command(name="tttsp", description="Play Tic Tac Toe against the bot!")
async def tttsingleplayer(
    interaction: nextcord.Interaction,
    difficulty: str = nextcord.SlashOption(
        name="difficulties",
        choices=["Easy", "Medium", "Hard"],
        required=True,
    )):
```

The next slash command is the single player mode of tic-tac-toe, where the player gets to play against the bot in varying difficulties. It only takes the three difficulties as input which would be shown as a list in the discord slash input field.



```python
# Adjusting the difficulty by changing the probability of blundering
if difficulty == "Easy":
    blunderProbability = 0.6
elif difficulty == "Medium:":
    blunderProbability = 0.4
elif difficulty == "Hard":
    blunderProbability = 0.1


# Changing the board into 9 square tiles
board = [":white_large_square:",":white_large_square:",":white_large_square:",
            ":white_large_square:",":white_large_square:",":white_large_square:",
            ":white_large_square:",":white_large_square:",":white_large_square:",]

turn = ""
gameOver = False  # Game has been started
count = 0  # Number of turns that have been taken

# Players
player1 = interaction.user
player2 = "Bot"

# Determining who takes the first turn
turnNum = random.randint(1, 2)
if turnNum == 1:
    turn = player1
elif turnNum == 2:
    turn = player2

# Creating embed using tttEmbed class
embed = tttEmbed(board, player1, player2, turn, count).embed

# Bot replies to the user with the created embed
await interaction.response.send_message(embed=embed, content=f"A single-player Tic Tac Toe match against the Bot has begun!")

# Determining what asynchronous functions should be run based on the turn
if turn == "Bot":
    await makeBotMove(interaction) # Runs the bot move function

else:
    await place(interaction) # Runs the place tile function
```

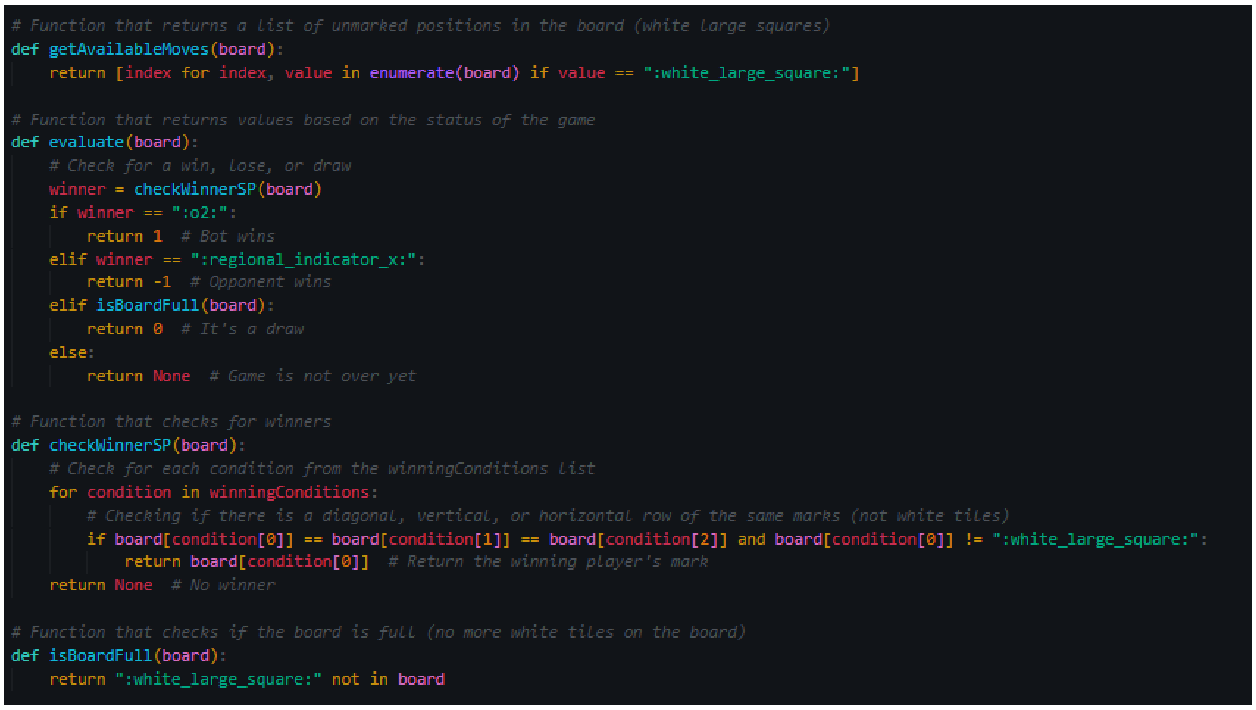
After accessing the global variables and checking if a game has been started, the game will be initialized using quite a similar method with the player versus player mode of tic-tac-toe. The difference between the single player mode and the player versus player mode is that in the single player mode, the program will first check the difficulty that the user has selected. The easy difficulty gives a high probability of the bot making blunders and the hard difficulty gives a low probability of the bot making blunders. Other than that, at the end of the code, the program decides what function to run according to who is taking the current turn.



```python
# Function to make a move for the Bot
async def makeBotMove(ctx: commands.Context):
    # Accessing global variables
    global turn
    global board
    global gameOver
    global blunderProbability

    mark = ":o2:"  # Bot's mark
    tttMainFunctions = tttMain(player1, player2, mark, turn)  # Creating new object for several functions

    # Check for a winning move

    if not gameOver:

        # Introduce occasional blunders according to the probability that was defined based on the difficulty chosen
        if random.random() < blunderProbability:
            print("blunder!")
            pos = random.randint(1, 9)
            while board[pos - 1] != ":white_large_square:":
                pos = random.randint(1, 9)
            board[pos - 1] = mark

        # Uses the MINIMAX algorithm when not blundering
        else:
            best_move = getBestMove(board) # Getting the best move through the minimax algorithm functions
            board[best_move] = mark  # Replacing the position with a mark

        # Creating a new, updated embed with the mark
        embed = tttEmbed(board, player1, player2, turn, count).embed

        # Bot sends the embed
        await ctx.send(embed=embed)

        # Checking if there is a winner
        checkWinner(winningConditions, mark)

        # Checks the status of gameOver (T/F) after running the checkWinner function
        if gameOver:
            # find the winner based on what the value of the mark
            if mark == ":regional_indicator_x:":
                await ctx.send(f"{player1.mention} Wins!")
            elif mark == ":o2:":
                await ctx.send("`Bot` Wins!")

        # Check if it reached the total number of turns it is possible to take
        elif isBoardFull(board):
            await ctx.send("It's a tie!")
            gameOver = True

        # Switching turns
        turn = tttMainFunctions.switchTurns
```

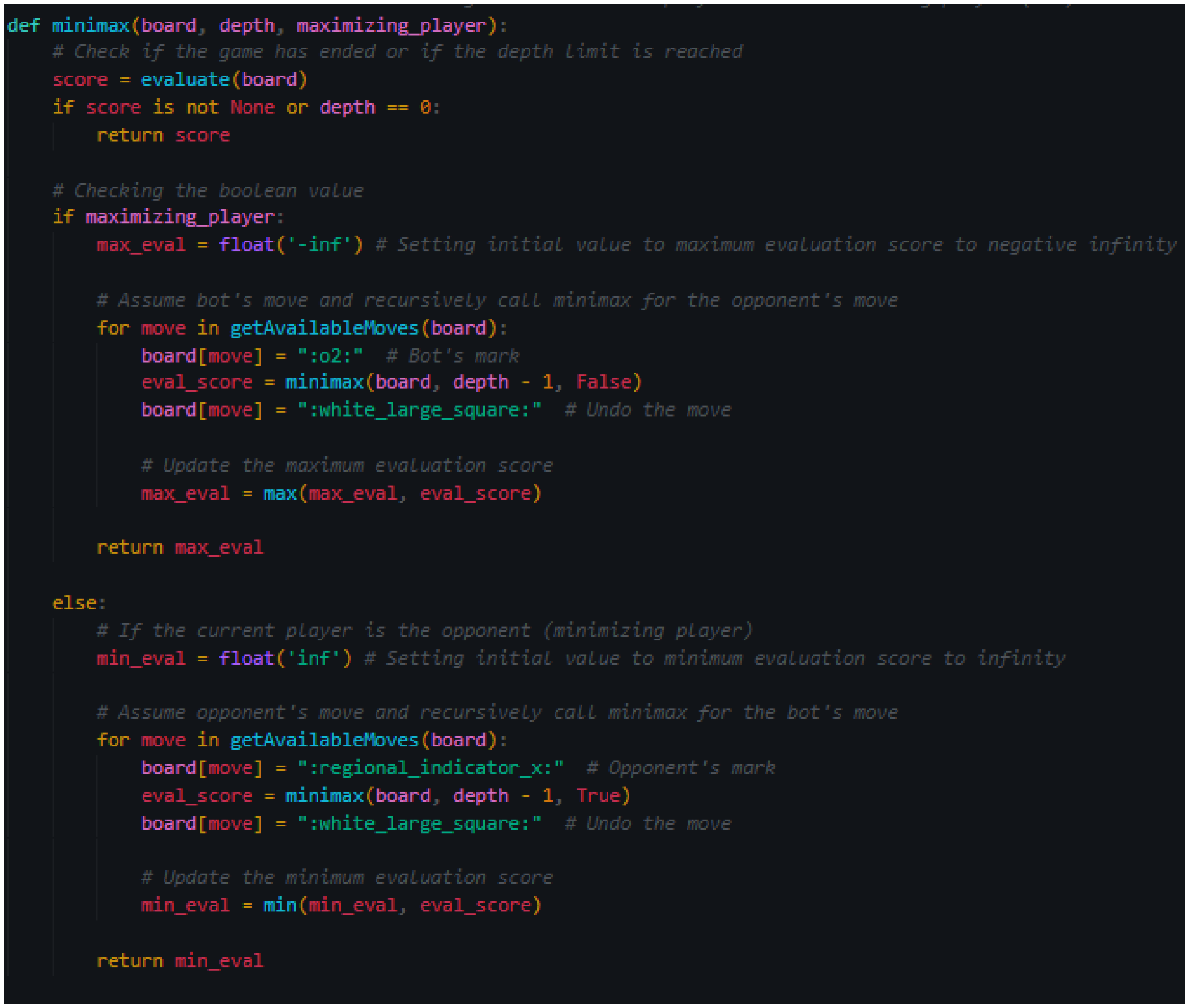
If it is the bot's turn, the program would run this asynchronous function which determines the bot's moves. The function starts off by accessing the global variables and defining the mark for the bot. It also creates an object using the tttMain function, just like the player versus player mode. If the game is still ongoing, I created an if-else statement that checks if the blunderProbability value, defined earlier based on the selected difficulty, is greater than a random value (from 0.0 to 1.0) generated by the random module's random function. If the randomly generated value is less than the blunderProbability value, it would use the random module's randint function to generate a random value between 1 and 9 (inclusive) to make the bot's next move, making it seem like the bot has made a blunder. While the position of the board is already marked by either the bot or the player, the program would keep generating random values until the position of the board is a white tile (unmarked). If the random value is greater than the probability, the bot would use the minimax algorithm (to be explained in the next section of the report) through the getBestMove function to make the most optimal move for it to win. The rest of the code works the same way as the place command. It sends an embed of the current state of the board, checks if there is a winner, then runs an if-else statement to check if either the game is over, or board is full.

The next section of the single player tic-tac-toe feature consists of a bunch of functions that represent the minimax algorithm. The Minimax algorithm is a decision-making algorithm commonly used in two-player turn-based games, such as chess or tic-tac-toe. The primary goal of the Minimax algorithm is to determine the optimal move for a player, considering the possible moves of the opponent. It works by recursively evaluating the possible outcomes of each move, assuming that both players play optimally. The algorithm assigns a score to each possible move, maximizing the score for the current player (the bot) and minimizing the score for the opponent (the player). This process is repeated until a terminal state, or a specified depth is reached. Minimax is widely used in artificial intelligence for game-playing agents to make strategic decisions.



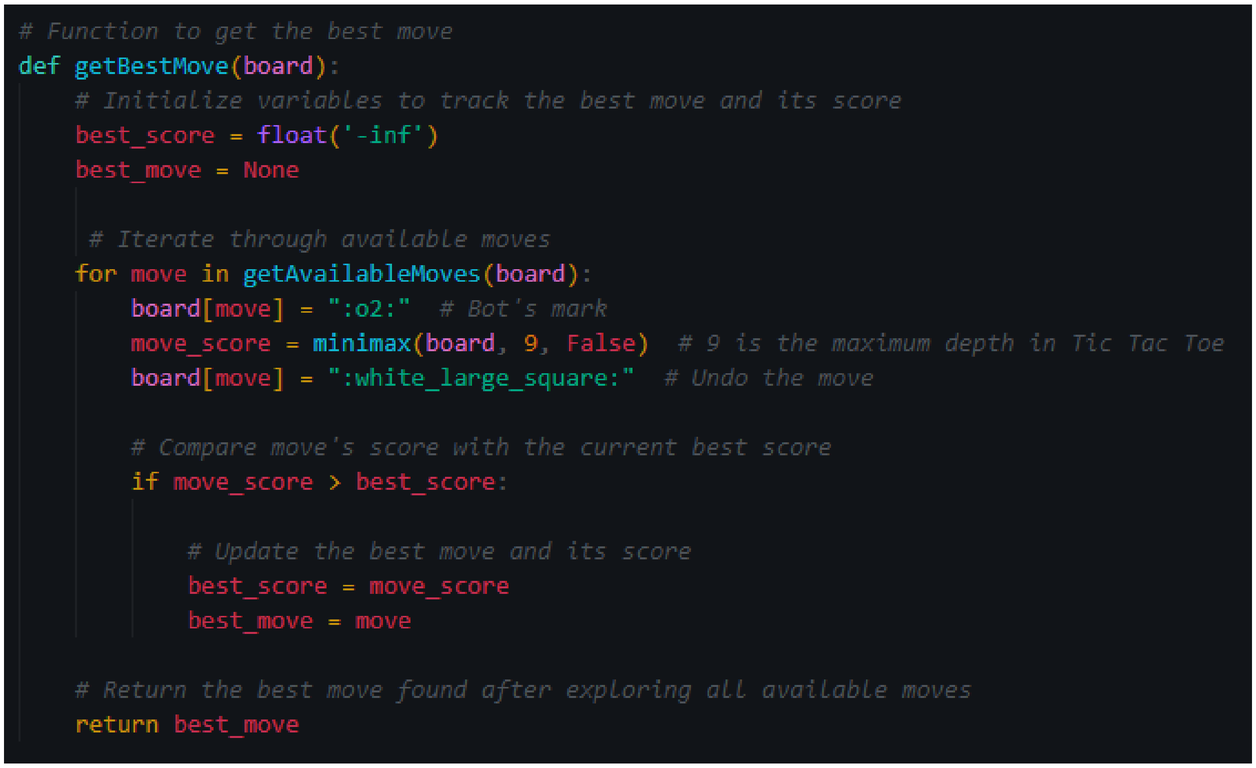
```python
# Function that returns a list of unmarked positions in the board (white large squares)
def getAvailableMoves(board):
    return [index for index, value in enumerate(board) if value == ":white_large_square:"]

# Function that returns values based on the status of the game
def evaluate(board):
    # Check for a win, lose, or draw
    winner = checkWinnerSP(board)
    if winner == ":o2:":
        return 1  # Bot wins
    elif winner == ":regional_indicator_x:":
        return -1  # Opponent wins
    elif isBoardFull(board):
        return 0  # It's a draw
    else:
        return None  # Game is not over yet

# Function that checks for winners
def checkWinnerSP(board):
    # Check for each condition from the winningConditions list
    for condition in winningConditions:
        # Checking if there is a diagonal, vertical, or horizontal row of the same marks (not white tiles)
        if board[condition[0]] == board[condition[1]] == board[condition[2]] and board[condition[0]] != ":white_large_square:":
            return board[condition[0]]  # Return the winning player's mark
    return None  # No winner

# Function that checks if the board is full (no more white tiles on the board)
def isBoardFull(board):
    return ":white_large_square:" not in board
```

These are the functions that are going to be used in the main part of the minimax algorithm. The first function checks the board and returns the list of unmarked positions (white tiles). Then, the evaluate function uses the checkWinnerSP function (checks if there is a winner) and returns the scores according to the conditions, based on the minimax algorithm, considering that the maximizing player is the bot and minimizing player is the opponent. The last function checks if there are any white tiles left in the board.



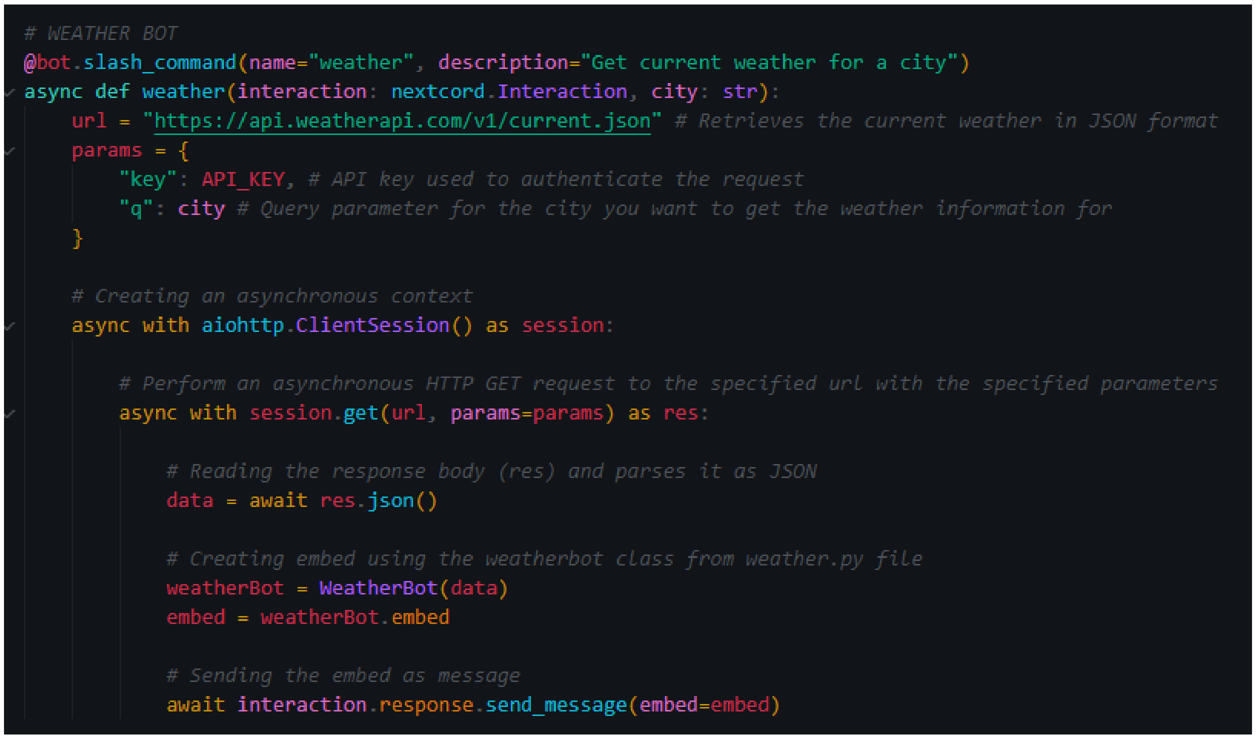
```python
def minimax(board, depth, maximizing_player):
    # Check if the game has ended or if the depth limit is reached
    score = evaluate(board)
    if score is not None or depth == 0:
        return score

    # Checking the boolean value
    if maximizing_player:
        max_eval = float('-inf') # Setting initial value to maximum evaluation score to negative infinity

        # Assume bot's move and recursively call minimax for the opponent's move
        for move in getAvailableMoves(board):
            board[move] = ":o2:"  # Bot's mark
            eval_score = minimax(board, depth - 1, False)
            board[move] = ":white_large_square:"  # Undo the move

            # Update the maximum evaluation score
            max_eval = max(max_eval, eval_score)

        return max_eval

    else:
        # If the current player is the opponent (minimizing player)
        min_eval = float('inf') # Setting initial value to minimum evaluation score to infinity

        # Assume opponent's move and recursively call minimax for the bot's move
        for move in getAvailableMoves(board):
            board[move] = ":regional_indicator_x:"  # Opponent's mark
            eval_score = minimax(board, depth - 1, True)
            board[move] = ":white_large_square:"  # Undo the move

            # Update the minimum evaluation score
            min_eval = min(min_eval, eval_score)

        return min_eval
```

This function is the main part of the minimax algorithm. To start this function, the score variable is defined, and it calls the evaluate function to check if it reached a terminal state (there is either a winner or the game ends in a draw). If score is not None and the depth (how many moves ahead the algorithm should explore) is 0, then a score is returned. After that, the program checks whether it is currently the maximizing player's turn or the minimizing player's turn. The maximizing player and the minimizing player's turn respectively have a max_eval and min_eval value set at the start. It is set to negative infinity in the maximizing player's turn to ensure that any valid move's evaluation score will be greater than the initial value and the opposite applies to the minimizing player's turn. Both the maximizing player and minimizing player's turns will iterate over the available moves on the board and for each move, the board position on that move would be replaced by the current player's mark. The minimax function is then called recursively, switching the turns until it reaches a terminal state. Once it reaches a terminal state, it would give a score that would be passed upwards the parse tree of the possible outcomes in the game which would be stores in the eval_score variable. The board position on that move would be reverted to the white tile. The value of the max_eval or min_eval variable is then replaced with the smallest or largest value between the previous max_eval or min_eval value and eval_score value. Once the loop finishes, the max_eval or min_eval values will be returned.



```python
# Function to get the best move
def getBestMove(board):
    # Initialize variables to track the best move and its score
    best_score = float('-inf')
    best_move = None

     # Iterate through available moves
    for move in getAvailableMoves(board):
        board[move] = ":o2:"  # Bot's mark
        move_score = minimax(board, 9, False)  # 9 is the maximum depth in Tic Tac Toe
        board[move] = ":white_large_square:"  # Undo the move

        # Compare move's score with the current best score
        if move_score > best_score:

            # Update the best move and its score
            best_score = move_score
            best_move = move

    # Return the best move found after exploring all available moves
    return best_move
```

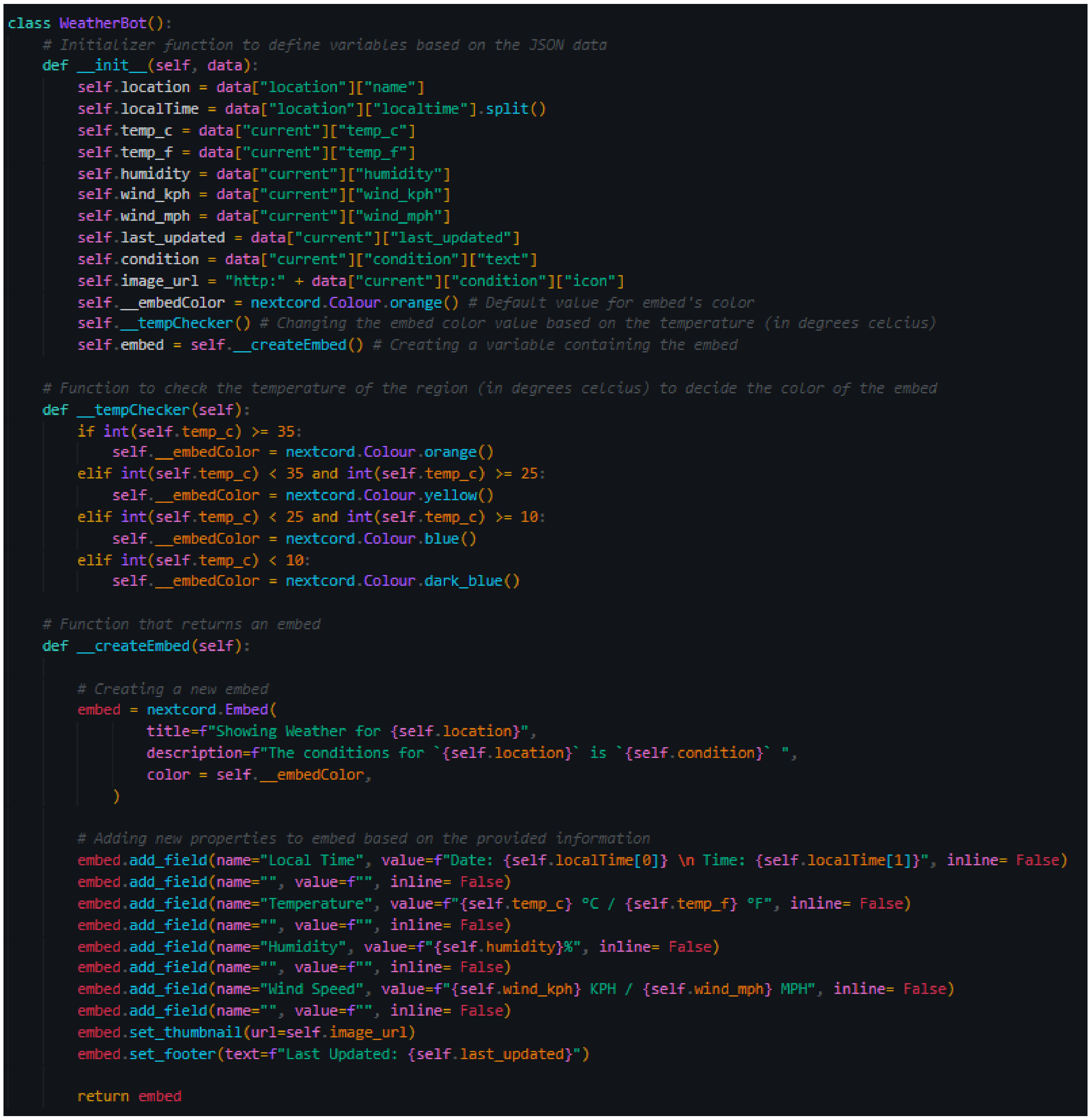
This function is the final function of the minimax algorithm section of the program. It first sets the best_score variable to negative infinity, to ensure that the evaluation scores would be greater than the initial value, and best_move to None. It then iterates over the available moves on the board like the case for maximizing player in the board. It calls the minimax function for every available move with maximizing player set to false because it is the player's turn next. This would make the minimax function run until it reaches all outcomes from making that move and storing the score value returned in the move_score variable. At the end of each loop, it updates the best_score and best_move value if the move_score value is greater than the current best_score. After the loop finishes, the program would know the best move to make based on the simulated outcomes and returns that best move, to be used in the makeBotMove function.



```python
# WEATHER BOT
@bot.slash_command(name="weather", description="Get current weather for a city")
async def weather(interaction: nextcord.Interaction, city: str):
    url = "https://api.weatherapi.com/v1/current.json" # Retrieves the current weather in JSON format
    params = {
        "key": API_KEY, # API key used to authenticate the request
        "q": city # Query parameter for the city you want to get the weather information for
    }

    # Creating an asynchronous context
    async with aiohttp.ClientSession() as session:

        # Perform an asynchronous HTTP GET request to the specified url with the specified parameters
        async with session.get(url, params=params) as res:

            # Reading the response body (res) and parses it as JSON
            data = await res.json()

            # Creating embed using the weatherbot class from weather.py file
            weatherBot = WeatherBot(data)
            embed = weatherBot.embed

            # Sending the embed as message
            await interaction.response.send_message(embed=embed)
```

Last but not least, this is the weather information display section of the program. It takes in a city name as an input, which would be used as a parameter, along with the API key defined at the top of the code, to complete the URL. The program would then perform an asynchronous HTTP GET request to the specified URL with the specified parameters. After that, a new variable called “data” is created and it contains the response body which was parsed as JSON. The data is then used as a parameter to create the weatherBot object, which is then used to create the embed that would be sent.

II. weather.py

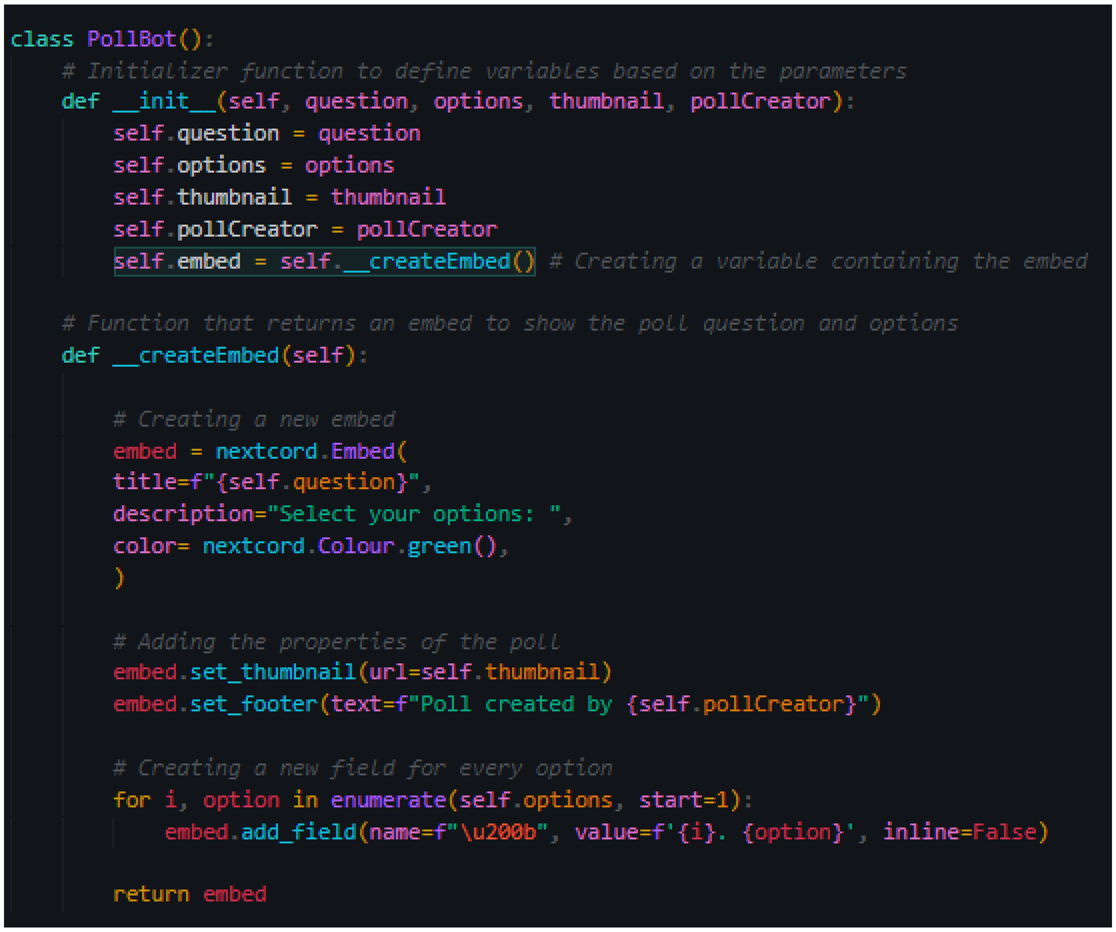


```python
class WeatherBot():
    # Initializer function to define variables based on the JSON data
    def __init__(self, data):
        self.location = data["location"]["name"]
        self.localTime = data["location"]["localtime"].split()
        self.temp_c = data["current"]["temp_c"]
        self.temp_f = data["current"]["temp_f"]
        self.humidity = data["current"]["humidity"]
        self.wind_kph = data["current"]["wind_kph"]
        self.wind_mph = data["current"]["wind_mph"]
        self.last_updated = data["current"]["last_updated"]
        self.condition = data["current"]["condition"]["text"]
        self.image_url = "http:" + data["current"]["condition"]["icon"]
        self.__embedColor = nextcord.Colour.orange() # Default value for embed's color
        self.__tempChecker() # Changing the embed color value based on the temperature (in degrees celcius)
        self.embed = self.__createEmbed() # Creating a variable containing the embed

    # Function to check the temperature of the region (in degrees celcius) to decide the color of the embed
    def __tempChecker(self):
        if int(self.temp_c) >= 35:
            self.__embedColor = nextcord.Colour.orange()
        elif int(self.temp_c) < 35 and int(self.temp_c) >= 25:
            self.__embedColor = nextcord.Colour.yellow()
        elif int(self.temp_c) < 25 and int(self.temp_c) >= 10:
            self.__embedColor = nextcord.Colour.blue()
        elif int(self.temp_c) < 10:
            self.__embedColor = nextcord.Colour.dark_blue()

    # Function that returns an embed
    def __createEmbed(self):

        # Creating a new embed
        embed = nextcord.Embed(
                title=f"Showing Weather for {self.location}",
                description=f"The conditions for `{self.location}` is `{self.condition}` ",
                color = self.__embedColor,
            )

        # Adding new properties to embed based on the provided information
        embed.add_field(name="Local Time", value=f"Date: {self.localTime[0]} \n Time: {self.localTime[1]}", inline= False)
        embed.add_field(name="", value=f"", inline= False)
        embed.add_field(name="Temperature", value=f"{self.temp_c} °C / {self.temp_f} °F", inline= False)
        embed.add_field(name="", value=f"", inline= False)
        embed.add_field(name="Humidity", value=f"{self.humidity}%", inline= False)
        embed.add_field(name="", value=f"", inline= False)
        embed.add_field(name="Wind Speed", value=f"{self.wind_kph} KPH / {self.wind_mph} MPH", inline= False)
        embed.add_field(name="", value=f"", inline= False)
        embed.set_thumbnail(url=self.image_url)
        embed.set_footer(text=f"Last Updated: {self.last_updated}")

        return embed
```

This file is used for the weather information display command. It consists of three functions: the constructor, a temperature checker to change the embed color based on the location's temperature, and an embed creator. The constructor accesses the parsed data (in JSON format) for different weather information about the city.

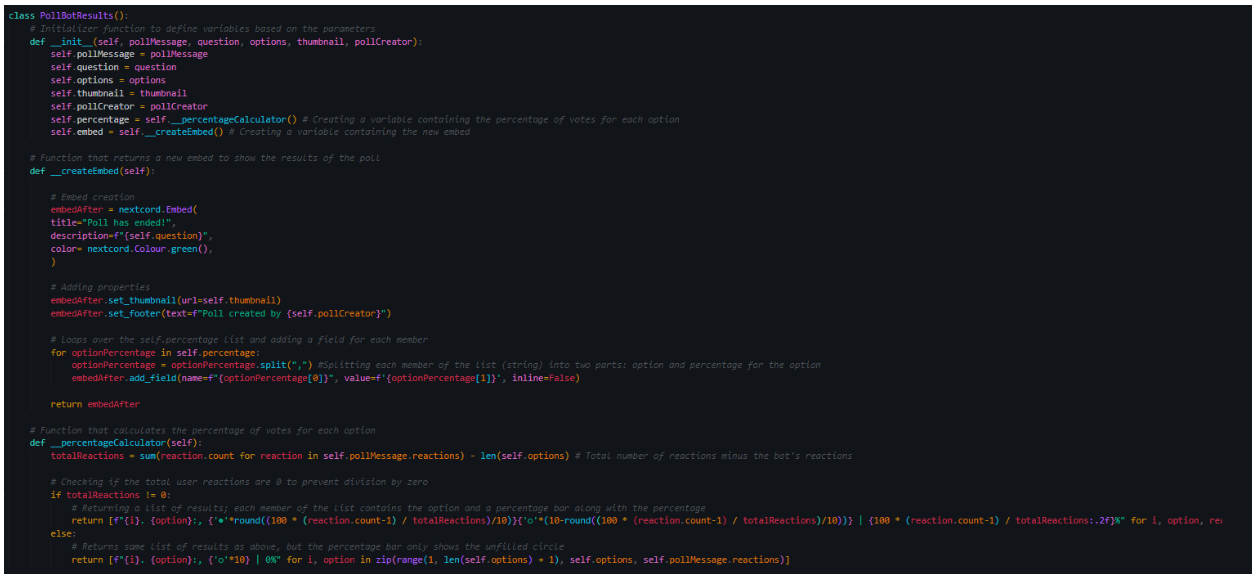
III. poll.py

The poll.py file consists of two classes: PollBot and PollBotResults, which have their own functions.



```python
class PollBot():
    # Initializer function to define variables based on the parameters
    def __init__(self, question, options, thumbnail, pollCreator):
        self.question = question
        self.options = options
        self.thumbnail = thumbnail
        self.pollCreator = pollCreator
        self.embed = self.__createEmbed() # Creating a variable containing the embed

    # Function that returns an embed to show the poll question and options
    def __createEmbed(self):

        # Creating a new embed
        embed = nextcord.Embed(
        title=f"{self.question}",
        description="Select your options: ",
        color= nextcord.Colour.green(),
        )

        # Adding the properties of the poll
        embed.set_thumbnail(url=self.thumbnail)
        embed.set_footer(text=f"Poll created by {self.pollCreator}")

        # Creating a new field for every option
        for i, option in enumerate(self.options, start=1):
            embed.add_field(name=f"\u200b", value=f'{i}. {option}', inline=False)

        return embed
```

Firstly, the PollBot class is mainly used to create the embed of the initial poll message (before it ends). It consists of a constructor and a function to create an embed.

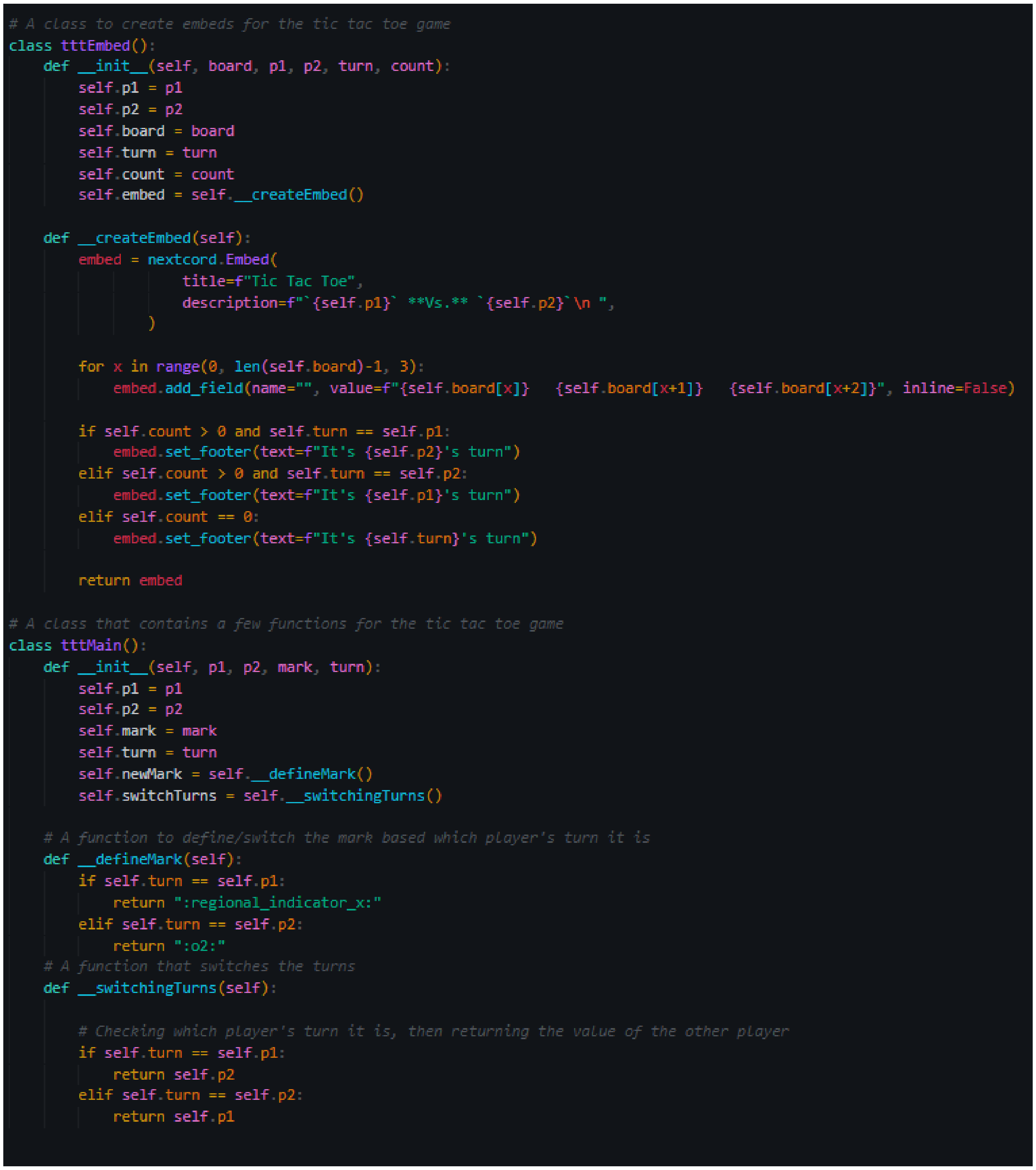


```
class PollBotResults():
    # Initializer function to define variables based on the parameters
    def __init__(self, pollMessage, question, options, thumbnail, pollCreator):
        self.pollMessage = pollMessage
        self.question = question
        self.options = options
        self.thumbnail = thumbnail
        self.pollCreator = pollCreator
        self.percentage = self.__percentageCalculator() # Creating a variable containing the percentage of votes for each option
        self.embed = self.__createEmbed() # Creating a variable containing the new embed

    # Function that returns a new embed to show the results of the poll
    def __createEmbed(self):

        # Embed creation
        embedAfter = nextcord.Embed(
        title="Poll has ended!",
        description=f"{self.question}",
        color= nextcord.Colour.green(),
        )

        # Adding properties
        embedAfter.set_thumbnail(url=self.thumbnail)
        embedAfter.set_footer(text=f"Poll created by {self.pollCreator}")

        # Loops over the self.percentage list and adding a field for each member
        for optionPercentage in self.percentage:
            optionPercentage = optionPercentage.split(",") #Splitting each member of the list (string) into two parts: option and percentage for the option
            embedAfter.add_field(name=f"{optionPercentage[0]}", value=f'{optionPercentage[1]}', inline=False)

        return embedAfter

    # Function that calculates the percentage of votes for each option
    def __percentageCalculator(self):
        totalReactions = sum(reaction.count for reaction in self.pollMessage.reactions) - len(self.options) # Total number of reactions minus the bot's reactions

        # Checking if the total user reactions are 0 to prevent division by zero
        if totalReactions != 0:
            # Returning a list of results; each member of the list contains the option and a percentage bar along with the percentage
            return [f"{i}. {option}:, {'●'*round((100 * (reaction.count-1) / totalReactions)/10)}{'o'*(10-round((100 * (reaction.count-1) / totalReactions)/10))} | {100 * (reaction.count-1) / totalReactions:.2f}%" for i, option, rea
        else:
            # Returns same list of results as above, but the percentage bar only shows the unfilled circle
            return [f"{i}. {option}:, {'o'*10} | 0%" for i, option in zip(range(1, len(self.options) + 1), self.options, self.pollMessage.reactions)]
```

The PollBotResults class is used to calculate the percentages of votes for each option and to create a new, updated embed that shows the percentages with a percentage bar. To calculate the percentages of votes, the function finds the number of total reactions made by users. It counts the sum of total reactions and subtracts it by the number of options to remove the bot's reactions. Then it checks if the total number of reactions are 0. Each of the if-else conditions return a list of reactions for each option. It loops through a range of numbers (from 1 up to the length of options), the options and the reactions of each option at the same time using the zip function which combines multiple iterable objects elementwise. For each element in the iteration, a string consisting of the percentage bar and the percentage value will be added to the list. The percentage is counted by multiplying the ratio between reactions for that option to the total reactions by 100. For the case where total reactions are 0, each of the list element would contain the string '[f"{i}. {option}:, {'○'*10} | 0%"' as it would automatically be 0% for all options.

IV. tictactoe.py

Lastly, the tictactoe.py file contains two classes with different purposes.



```python
# A class to create embeds for the tic tac toe game
class tttEmbed():
    def __init__(self, board, p1, p2, turn, count):
        self.p1 = p1
        self.p2 = p2
        self.board = board
        self.turn = turn
        self.count = count
        self.embed = self.__createEmbed()

    def __createEmbed(self):
        embed = nextcord.Embed(
                    title=f"Tic Tac Toe",
                    description=f"`{self.p1}` **Vs.** `{self.p2}`\n ",
                )

        for x in range(0, len(self.board)-1, 3):
            embed.add_field(name="", value=f"{self.board[x]}   {self.board[x+1]}   {self.board[x+2]}", inline=False)

        if self.count > 0 and self.turn == self.p1:
            embed.set_footer(text=f"It's {self.p2}'s turn")
        elif self.count > 0 and self.turn == self.p2:
            embed.set_footer(text=f"It's {self.p1}'s turn")
        elif self.count == 0:
            embed.set_footer(text=f"It's {self.turn}'s turn")

        return embed

# A class that contains a few functions for the tic tac toe game
class tttMain():
    def __init__(self, p1, p2, mark, turn):
        self.p1 = p1
        self.p2 = p2
        self.mark = mark
        self.turn = turn
        self.newMark = self.__defineMark()
        self.switchTurns = self.__switchingTurns()

    # A function to define/switch the mark based which player's turn it is
    def __defineMark(self):
        if self.turn == self.p1:
            return ":regional_indicator_x:"
        elif self.turn == self.p2:
            return ":o2:"
    # A function that switches the turns
    def __switchingTurns(self):

        # Checking which player's turn it is, then returning the value of the other player
        if self.turn == self.p1:
            return self.p2
        elif self.turn == self.p2:
            return self.p1
```

Firstly, the tttEmbed class is responsible for creating the embed for the tic-tac-toe game, just like the other embed classes from other files. It takes in the board, player 1, player 2, which player is taking the turn, and the count (how many turns have been taken) as a parameter. It has a constructor function and a private createEmbed function that returns an embed.

Next, the tttMain function is responsible for some of the main functions in the game. It takes in player 1, player 2, player's mark, and which player is taking the turn as parameters. It contains a constructor function, a function to define the mark based on whose turn it is, and a function to switch the turns.

Self-Reflection

A couple of months before starting on this project, I initially thought of making a game using the pygame module but wasn't sure how I'd be able to create my own assets quickly. Other than that, I was not even sure what type of game I would want to make. On top of that, I realized that I won't have a lot of time to work on this project, as I was already busy with tasks from other courses, so I ended up going with the idea of creating a Discord bot as I was already quite familiar with the concept of Discord bots and have tried creating one in the past using Node JS.

At first, I was excited about the idea of creating a Discord bot, because I've always wanted to create my own (at least decent) working bot. However, during the process of working on my project, I got to see a few of other student's works and started to think that my project is a bit too simple compared to others. I started to slowly lose motivation to continue it at that point, but then still tried my best to continue it as much as I could. Thankfully, throughout the rest of the journey, I started to have fun working on the project once again until I was able to finish it on time. I found more ideas for features that my Discord bot could have and tried my best to implement it with no bugs or errors.

I have learned a lot from this experience, not only in programming, but also something about myself that could be improved. In the perspective of programming, I learned a lot about the libraries and APIs required to create a discord bot using Python. I learned how asynchronous programming works, how it is implemented, and why it is used in Discord bots. Other than that, by creating the weather information display command, I discovered the AIOHTTP library that could get http requests asynchronously and learned how information in JSON format is accessed. In addition, I also learned things about myself that could be improved. Firstly, I should stop comparing my work to others, as everyone has different abilities and skills, so as long as I do my best, I should be more confident with what I have. Aside from that, I should refrain from procrastination more, because I found myself procrastinating during the process of creating this project.

Resources

I. Creating the Discord bot

- https://docs.nextcord.dev/en/stable/genindex.html
- https://docs.nextcord.dev/en/stable/index.html
- https://stackoverflow.com/
- https://www.flaticon.com/ (Logo)

II. Creating the report

- https://app.diagrams.net/ (Creating diagrams)