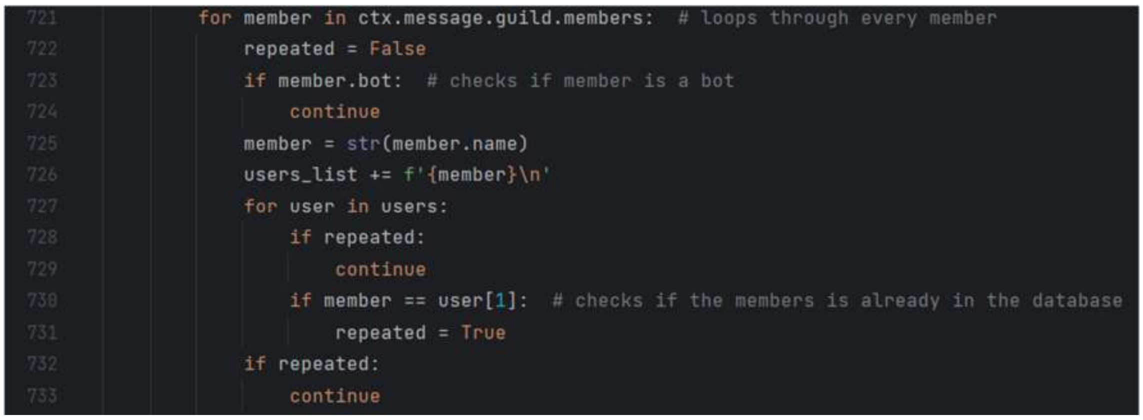
Criterion C: Development

Techniques used:

1 My main algorithms:

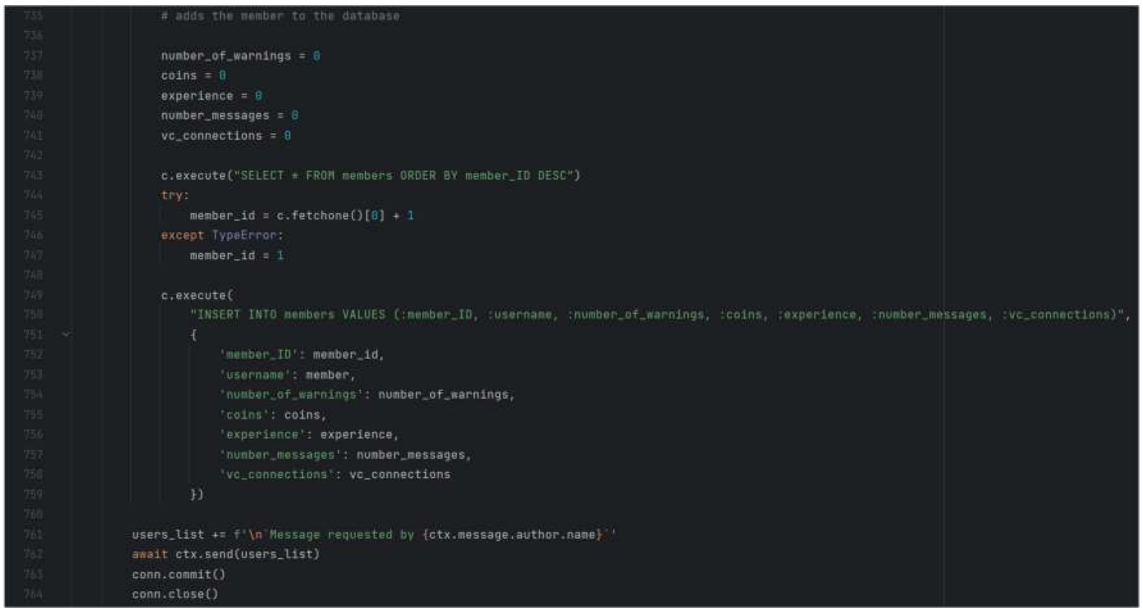
Member adding algorithm:

It’s necessary to store members in the database. When new members join the guild, it’s stored in the database. The algorithm has two parts.



```python
    for member in ctx.message.guild.members:  # loops through every member
        repeated = False
        if member.bot:  # checks if member is a bot
            continue
        member = str(member.name)
        users_list += f'{member}\n'
        for user in users:
            if repeated:
                continue
            if member == user[1]:  # checks if the members is already in the database
                repeated = True
        if repeated:
            continue
```

First checks if the member is a bot and if it’s in the database. If one conditions is true, it skips onto the following member in the outer for loop.



```python
        # adds the member to the database

        number_of_warnings = 0
        coins = 0
        experience = 0
        number_messages = 0
        vc_connections = 0

        c.execute("SELECT * FROM members ORDER BY member_ID DESC")
        try:
            member_id = c.fetchone()[0] + 1
        except TypeError:
            member_id = 1

        c.execute(
            "INSERT INTO members VALUES (:member_ID, :username, :number_of_warnings, :coins, :experience, :number_messages, :vc_connections)",
            {
                'member_ID': member_id,
                'username': member,
                'number_of_warnings': number_of_warnings,
                'coins': coins,
                'experience': experience,
                'number_messages': number_messages,
                'vc_connections': vc_connections
            })

    users_list += f'\n`Message requested by {ctx.message.author.name}`'
    await ctx.send(users_list)
    conn.commit()
    conn.close()
```

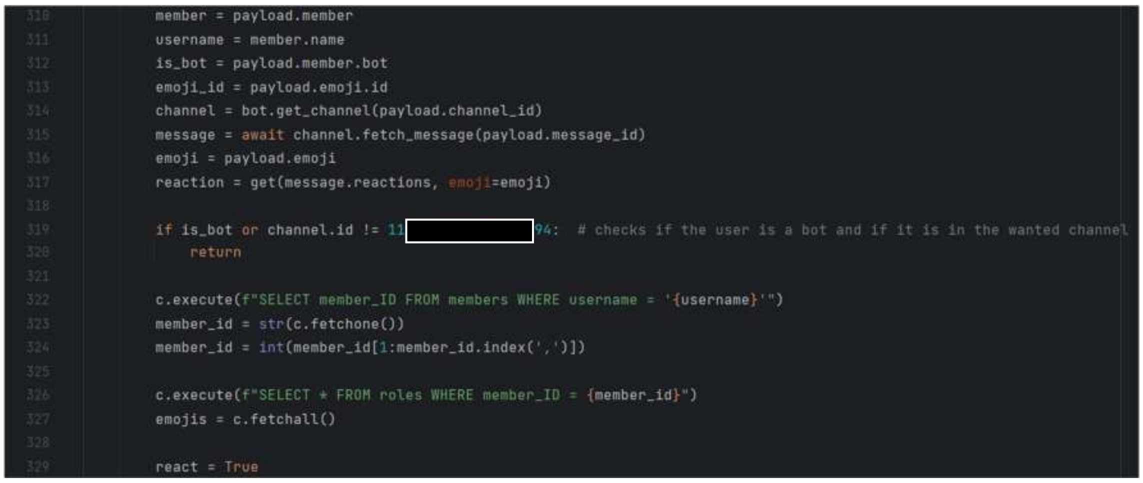
Secondly, it uses a dictionary to insert the values to the database.

It’s helpful when the program is first implemented to the guild or if the database needs to be reset for any issue. This helps achieve success criteria 5.

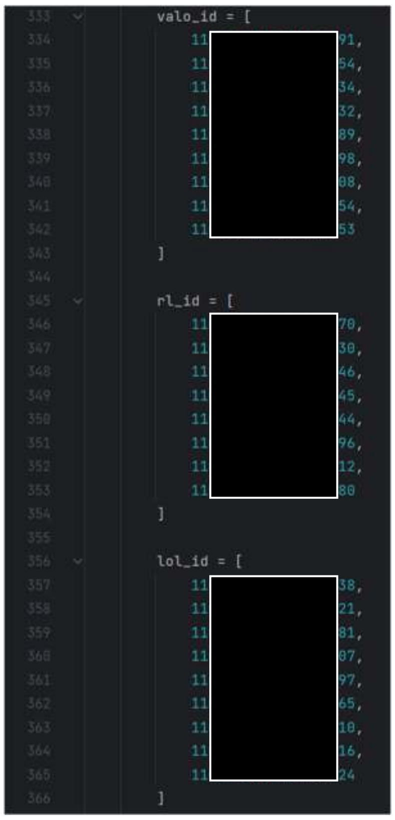
Emoji id algorithm:

This algorithm is necessary when a member wants to add a reaction to get a new role.

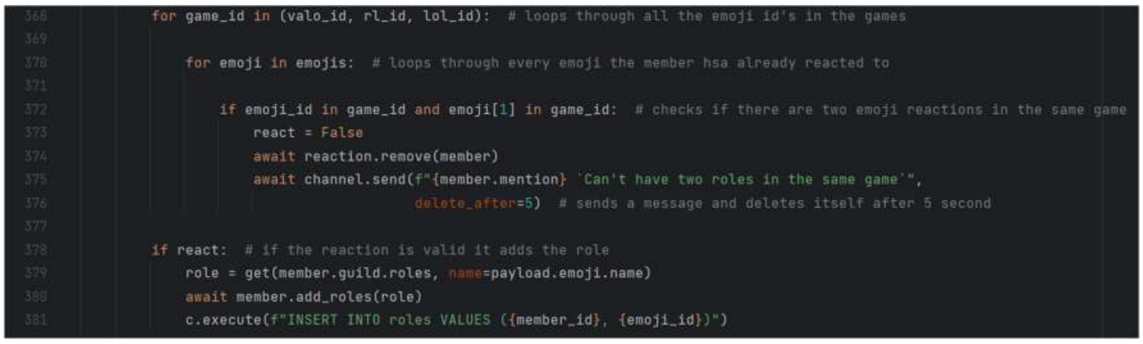
It checks if the member reacts with an emoji to the embed message sent in the wanted channel of the guild.



```python
        member = payload.member
        username = member.name
        is_bot = payload.member.bot
        emoji_id = payload.emoji.id
        channel = bot.get_channel(payload.channel_id)
        message = await channel.fetch_message(payload.message_id)
        emoji = payload.emoji
        reaction = get(message.reactions, emoji=emoji)

        if is_bot or channel.id != 11[redacted]94:  # checks if the user is a bot and if it is in the wanted channel
            return

        c.execute(f"SELECT member_ID FROM members WHERE username = '{username}'")
        member_id = str(c.fetchone())
        member_id = int(member_id[1:member_id.index(',')])

        c.execute(f"SELECT * FROM roles WHERE member_ID = {member_id}")
        emojis = c.fetchall()

        react = True
```



```python
        valo_id = [
            11[redacted]91,
            11[redacted]54,
            11[redacted]34,
            11[redacted]32,
            11[redacted]89,
            11[redacted]98,
            11[redacted]08,
            11[redacted]54,
            11[redacted]53
        ]

        rl_id = [
            11[redacted]70,
            11[redacted]30,
            11[redacted]46,
            11[redacted]45,
            11[redacted]44,
            11[redacted]96,
            11[redacted]12,
            11[redacted]80
        ]

        lol_id = [
            11[redacted]38,
            11[redacted]21,
            11[redacted]81,
            11[redacted]07,
            11[redacted]97,
            11[redacted]65,
            11[redacted]10,
            11[redacted]16,
            11[redacted]24
        ]
```

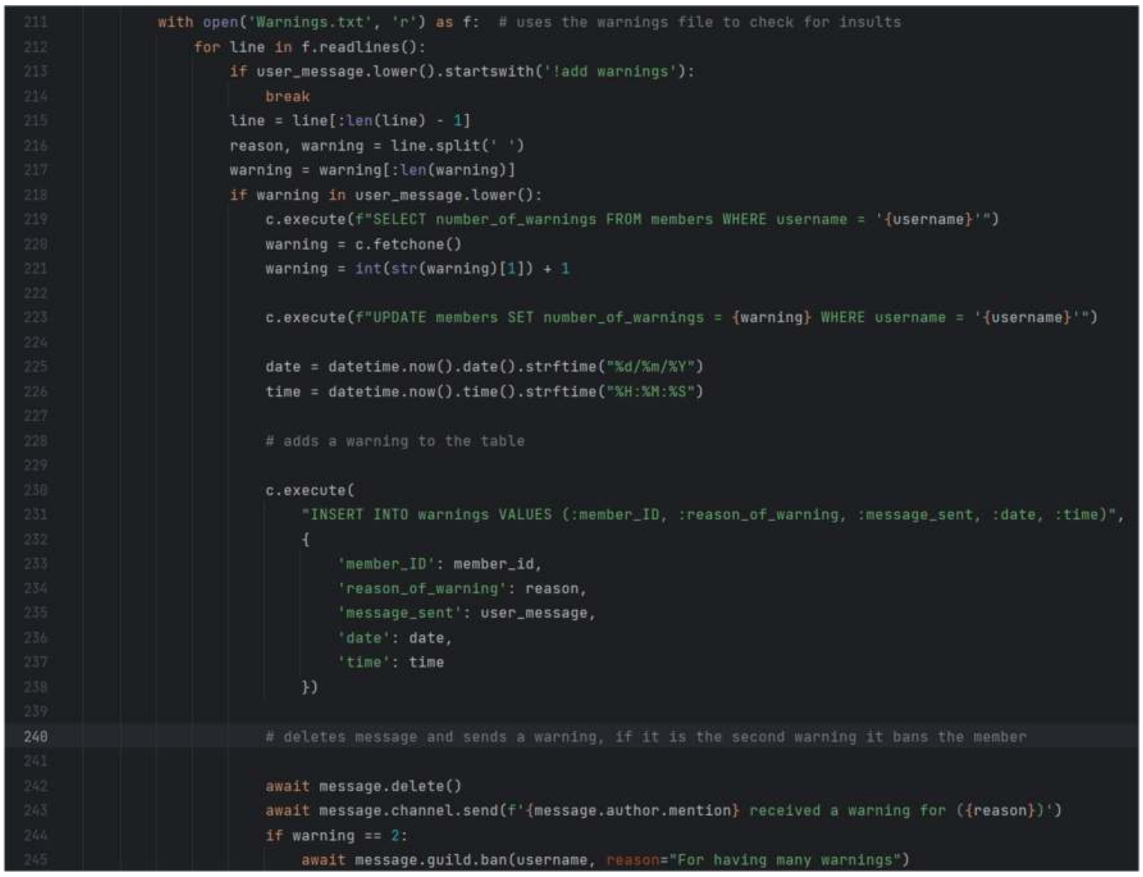


```python
        for game_id in (valo_id, rl_id, lol_id):  # loops through all the emoji id's in the games

            for emoji in emojis:  # loops through every emoji the member hsa already reacted to

                if emoji_id in game_id and emoji[1] in game_id:  # checks if there are two emoji reactions in the same game
                    react = False
                    await reaction.remove(member)
                    await channel.send(f"{member.mention} `Can't have two roles in the same game`",
                                       delete_after=5)  # sends a message and deletes itself after 5 second

        if react:  # if the reaction is valid it adds the role
            role = get(member.guild.roles, name=payload.emoji.name)
            await member.add_roles(role)
            c.execute(f"INSERT INTO roles VALUES ({member_id}, {emoji_id})")
```

The purpose is to avoid members having many roles for each game. If members have no previous reactions in that game, it gives the role corresponding to the reacted emoji.

Warning algorithm:

It activates when a member sends a message. Gets all the banned words from a text file and check if the message contains one of these words.

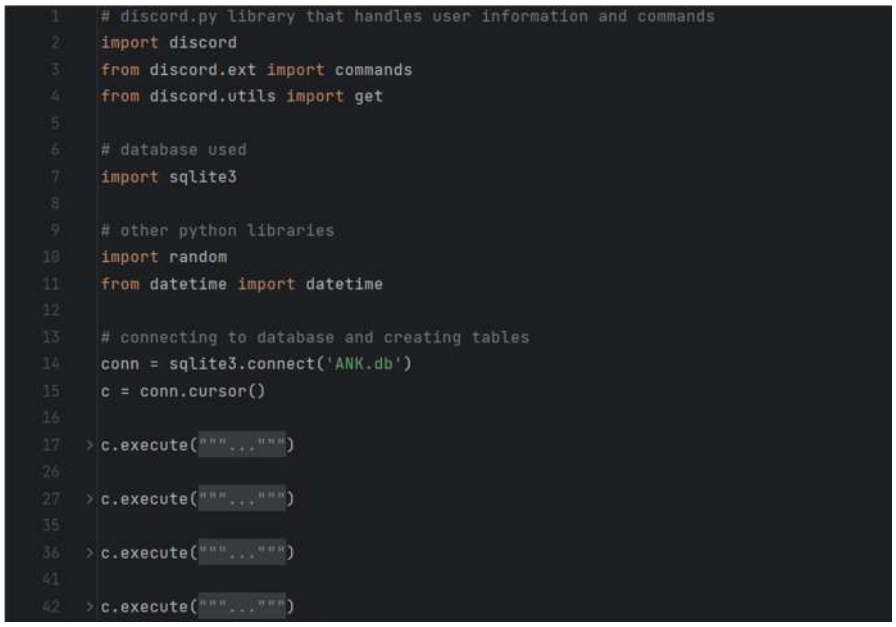


```
with open('Warnings.txt', 'r') as f:  # uses the warnings file to check for insults
    for line in f.readlines():
        if user_message.lower().startswith('!add warnings'):
            break
        line = line[:len(line) - 1]
        reason, warning = line.split(' ')
        warning = warning[:len(warning)]
        if warning in user_message.lower():
            c.execute(f"SELECT number_of_warnings FROM members WHERE username = '{username}'")
            warning = c.fetchone()
            warning = int(str(warning)[1]) + 1

            c.execute(f"UPDATE members SET number_of_warnings = {warning} WHERE username = '{username}'")

            date = datetime.now().date().strftime("%d/%m/%Y")
            time = datetime.now().time().strftime("%H:%M:%S")

            # adds a warning to the table

            c.execute(
                "INSERT INTO warnings VALUES (:member_ID, :reason_of_warning, :message_sent, :date, :time)",
                {
                    'member_ID': member_id,
                    'reason_of_warning': reason,
                    'message_sent': user_message,
                    'date': date,
                    'time': time
                })

            # deletes message and sends a warning, if it is the second warning it bans the member

            await message.delete()
            await message.channel.send(f'{message.author.mention} received a warning for ({reason})')
            if warning == 2:
                await message.guild.ban(username, reason="For having many warnings")
```

Retrieves the message sent from the member, the reason of warning from the text file and the date and time, to add the warning to the database. This helps achieve success criteria 7 as it bans the member when the second warning is given.

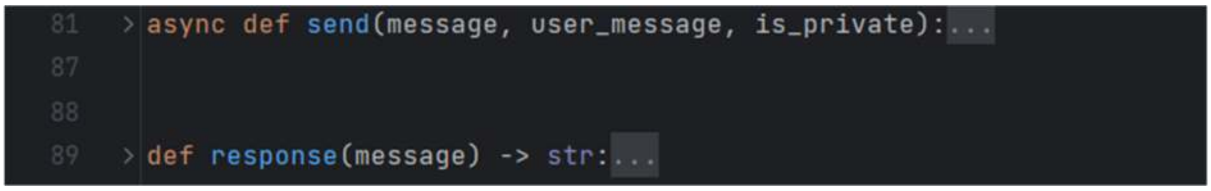
2 Coherent structure and layout of the code:

The code's structure and layout allow easy understanding of each section. This facilitates the editing and extension of the program. It starts by importing libraries and creating tables of database.



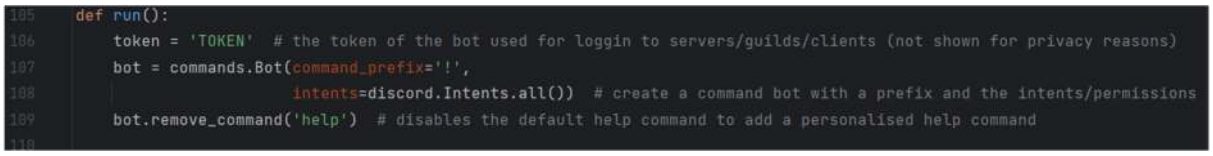
```python
# discord.py library that handles user information and commands
import discord
from discord.ext import commands
from discord.utils import get

# database used
import sqlite3

# other python libraries
import random
from datetime import datetime

# connecting to database and creating tables
conn = sqlite3.connect('ANK.db')
c = conn.cursor()

c.execute("""...""")

c.execute("""...""")

c.execute("""...""")

c.execute("""...""")
```

The code then has two secondary functions



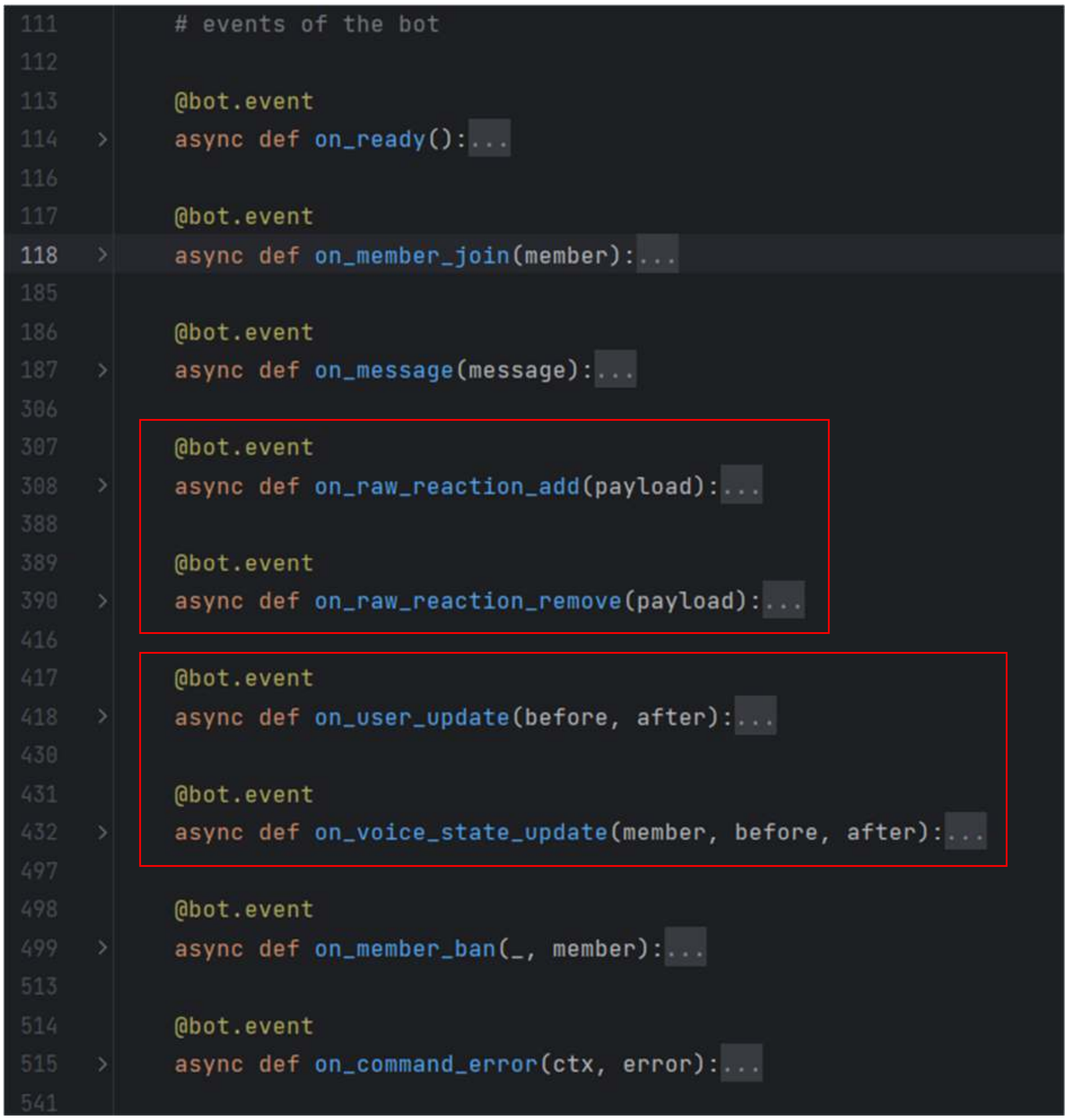
```python
async def send(message, user_message, is_private):...


def response(message) -> str:...
```

The main function that states token and bot

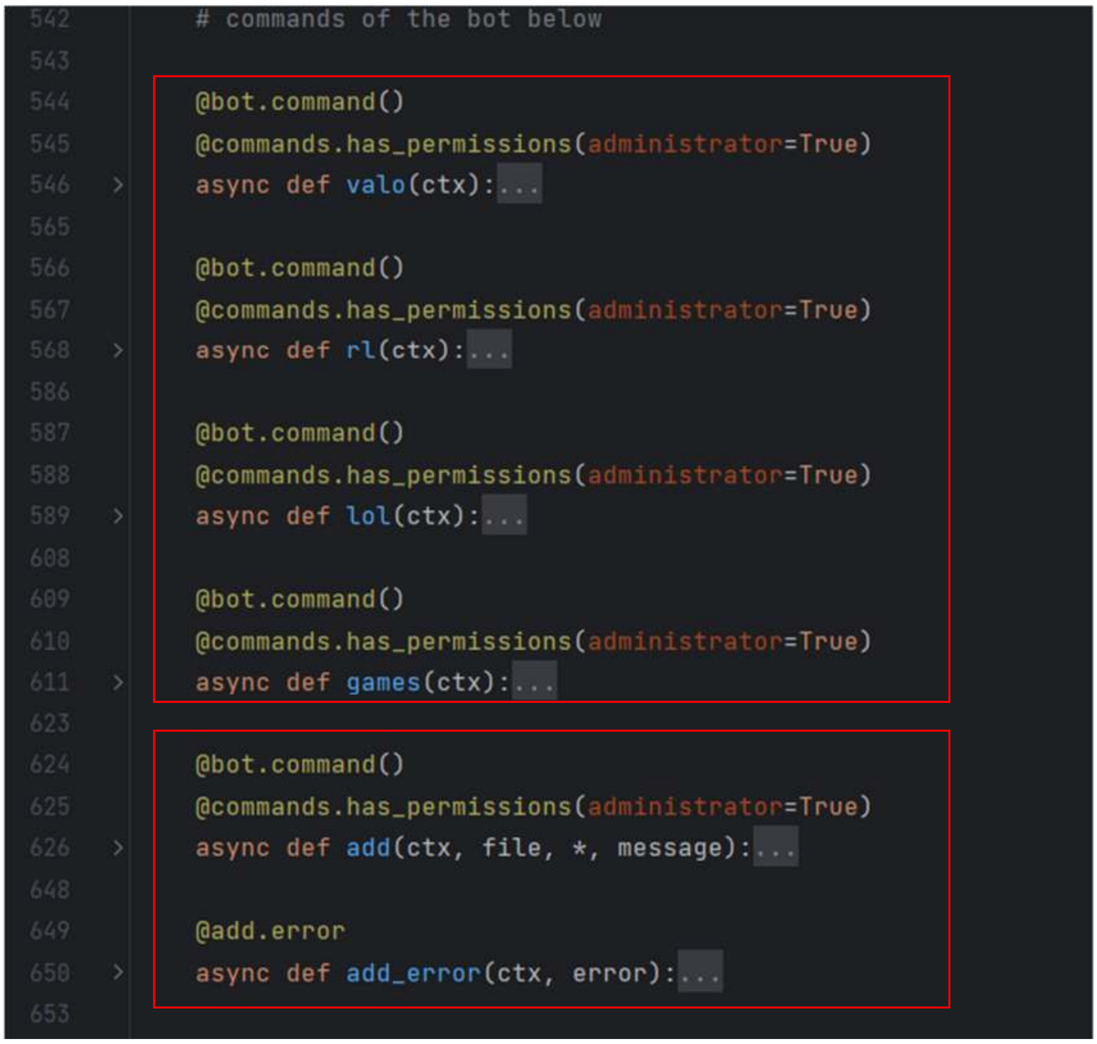


```python
def run():
    token = 'TOKEN'  # the token of the bot used for loggin to servers/guilds/clients (not shown for privacy reasons)
    bot = commands.Bot(command_prefix='!',
                       intents=discord.Intents.all())  # create a command bot with a prefix and the intents/permissions
    bot.remove_command('help')  # disables the default help command to add a personalised help command
```

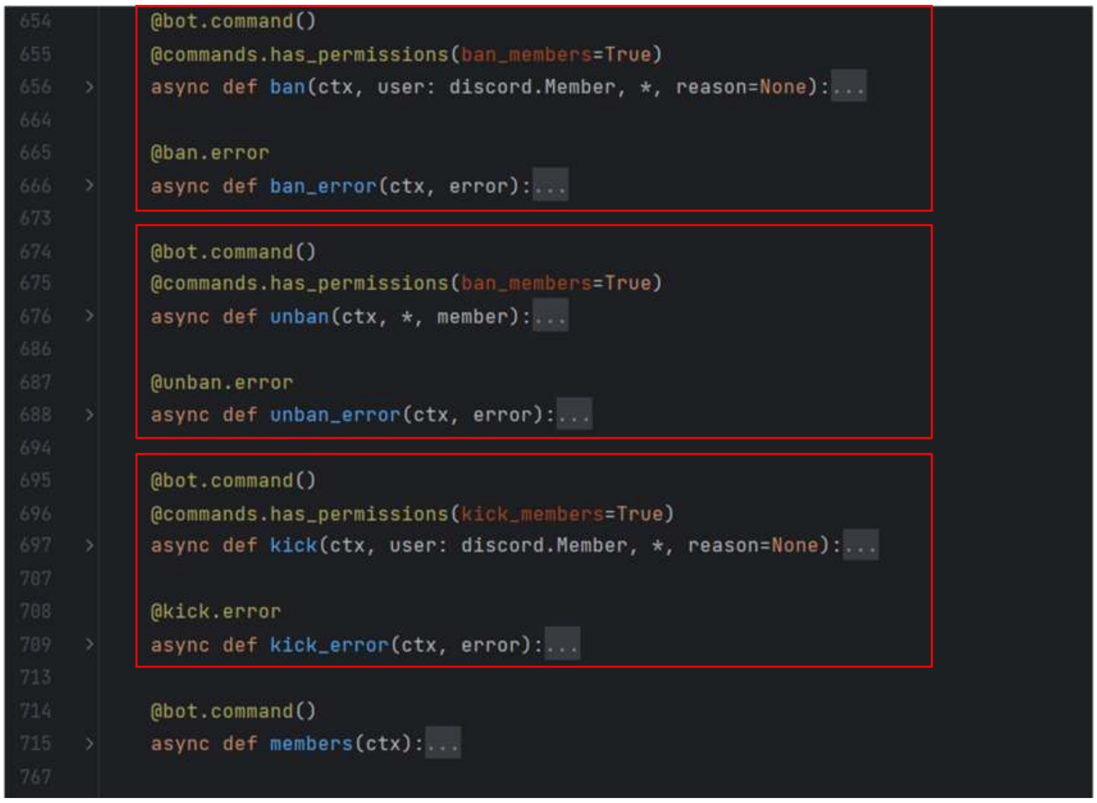
The main function contains the event handlers and command handlers. The async functions, are ordered by group when possible (groups shown in red boxes).



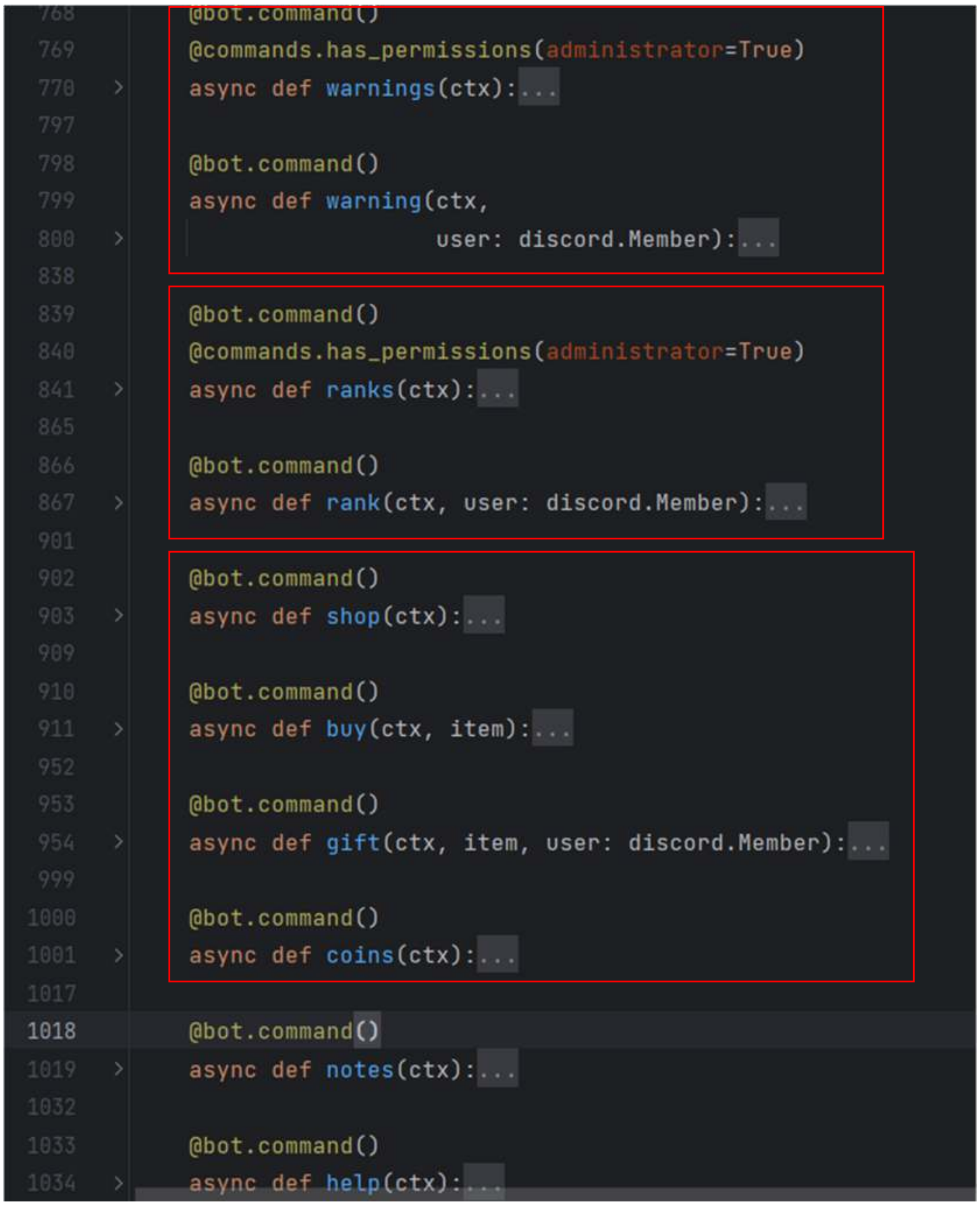
```python
# events of the bot

@bot.event
async def on_ready():...

@bot.event
async def on_member_join(member):...

@bot.event
async def on_message(message):...

@bot.event
async def on_raw_reaction_add(payload):...

@bot.event
async def on_raw_reaction_remove(payload):...

@bot.event
async def on_user_update(before, after):...

@bot.event
async def on_voice_state_update(member, before, after):...

@bot.event
async def on_member_ban(_, member):...

@bot.event
async def on_command_error(ctx, error):...
```



```python
# commands of the bot below

@bot.command()
@commands.has_permissions(administrator=True)
async def valo(ctx):...

@bot.command()
@commands.has_permissions(administrator=True)
async def rl(ctx):...

@bot.command()
@commands.has_permissions(administrator=True)
async def lol(ctx):...

@bot.command()
@commands.has_permissions(administrator=True)
async def games(ctx):...

@bot.command()
@commands.has_permissions(administrator=True)
async def add(ctx, file, *, message):...

@add.error
async def add_error(ctx, error):...

```



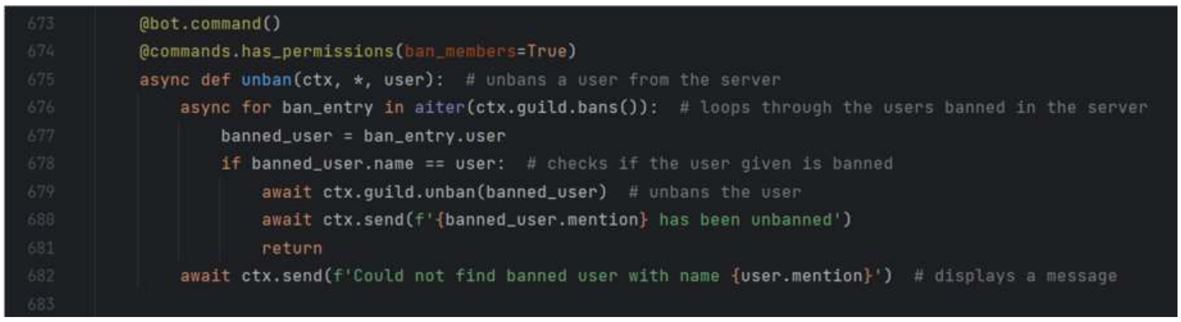
```python
@bot.command()
@commands.has_permissions(ban_members=True)
async def ban(ctx, user: discord.Member, *, reason=None):...

@ban.error
async def ban_error(ctx, error):...

@bot.command()
@commands.has_permissions(ban_members=True)
async def unban(ctx, *, member):...

@unban.error
async def unban_error(ctx, error):...

@bot.command()
@commands.has_permissions(kick_members=True)
async def kick(ctx, user: discord.Member, *, reason=None):...

@kick.error
async def kick_error(ctx, error):...

@bot.command()
async def members(ctx):...

```



```python
@bot.command()
@commands.has_permissions(administrator=True)
async def warnings(ctx):...

@bot.command()
async def warning(ctx,
                  user: discord.Member):...

@bot.command()
@commands.has_permissions(administrator=True)
async def ranks(ctx):...

@bot.command()
async def rank(ctx, user: discord.Member):...

@bot.command()
async def shop(ctx):...

@bot.command()
async def buy(ctx, item):...

@bot.command()
async def gift(ctx, item, user: discord.Member):...

@bot.command()
async def coins(ctx):...

@bot.command()
async def notes(ctx):...

@bot.command()
async def help(ctx):...
```

3 In-line comments:

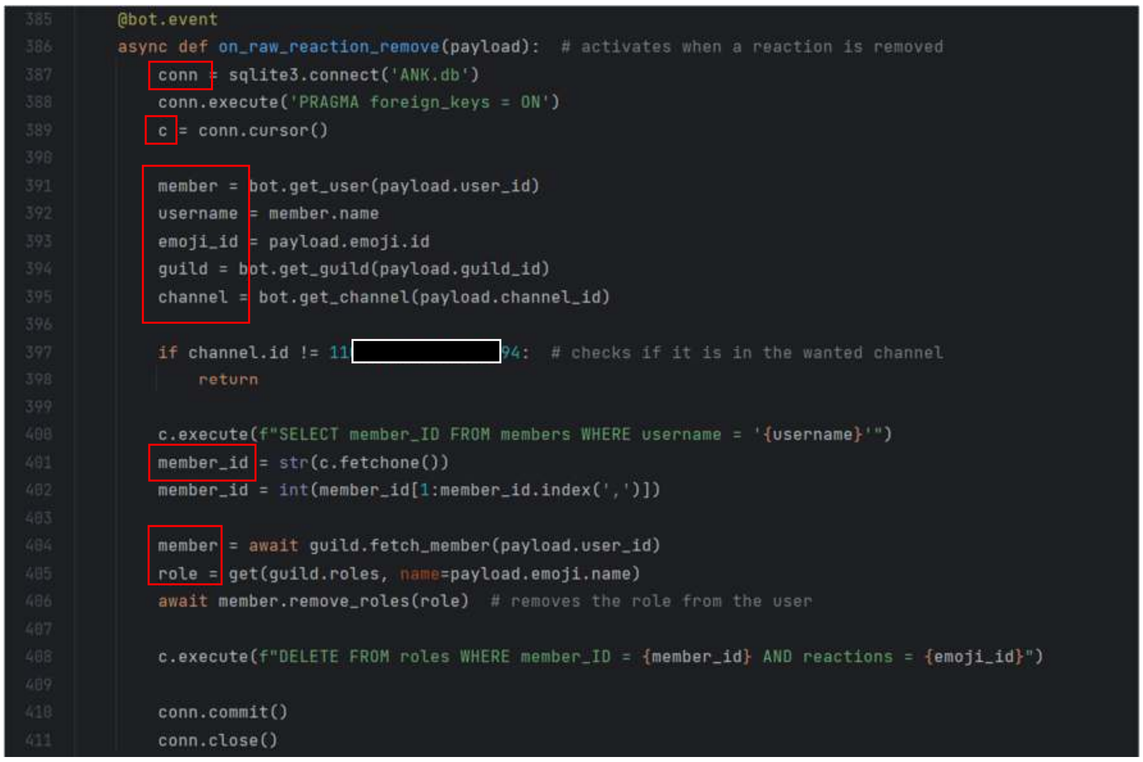
The code has comments throughout explaining every section for extensibility.



```python
@bot.command()
@commands.has_permissions(ban_members=True)
async def unban(ctx, *, user):  # unbans a user from the server
    async for ban_entry in aiter(ctx.guild.bans()):  # loops through the users banned in the server
        banned_user = ban_entry.user
        if banned_user.name == user:  # checks if the user given is banned
            await ctx.guild.unban(banned_user)  # unbans the user
            await ctx.send(f'{banned_user.mention} has been unbanned')
            return
    await ctx.send(f'Could not find banned user with name {user.mention}')  # displays a message
```

4 Meaningful names of variables and functions:

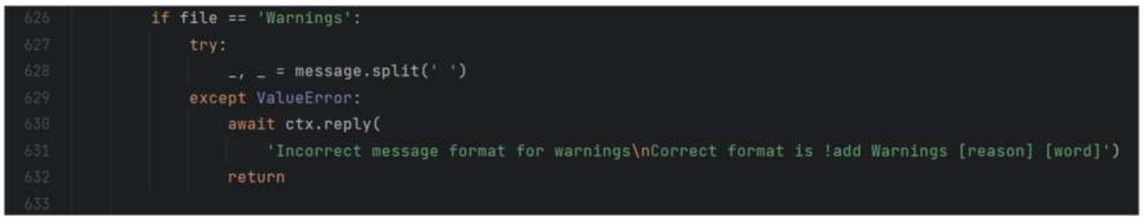
All variables and functions have a name according with the data it holds for extensibility.



```python
@bot.event
async def on_raw_reaction_remove(payload):  # activates when a reaction is removed
    conn = sqlite3.connect('ANK.db')
    conn.execute('PRAGMA foreign_keys = ON')
    c = conn.cursor()

    member = bot.get_user(payload.user_id)
    username = member.name
    emoji_id = payload.emoji.id
    guild = bot.get_guild(payload.guild_id)
    channel = bot.get_channel(payload.channel_id)

    if channel.id != 11[redacted]94:  # checks if it is in the wanted channel
        return

    c.execute(f"SELECT member_ID FROM members WHERE username = '{username}'")
    member_id = str(c.fetchone())
    member_id = int(member_id[1:member_id.index(',')])

    member = await guild.fetch_member(payload.user_id)
    role = get(guild.roles, name=payload.emoji.name)
    await member.remove_roles(role)  # removes the role from the user

    c.execute(f"DELETE FROM roles WHERE member_ID = {member_id} AND reactions = {emoji_id}")

    conn.commit()
    conn.close()
```

5 Validation techniques used for inputs:

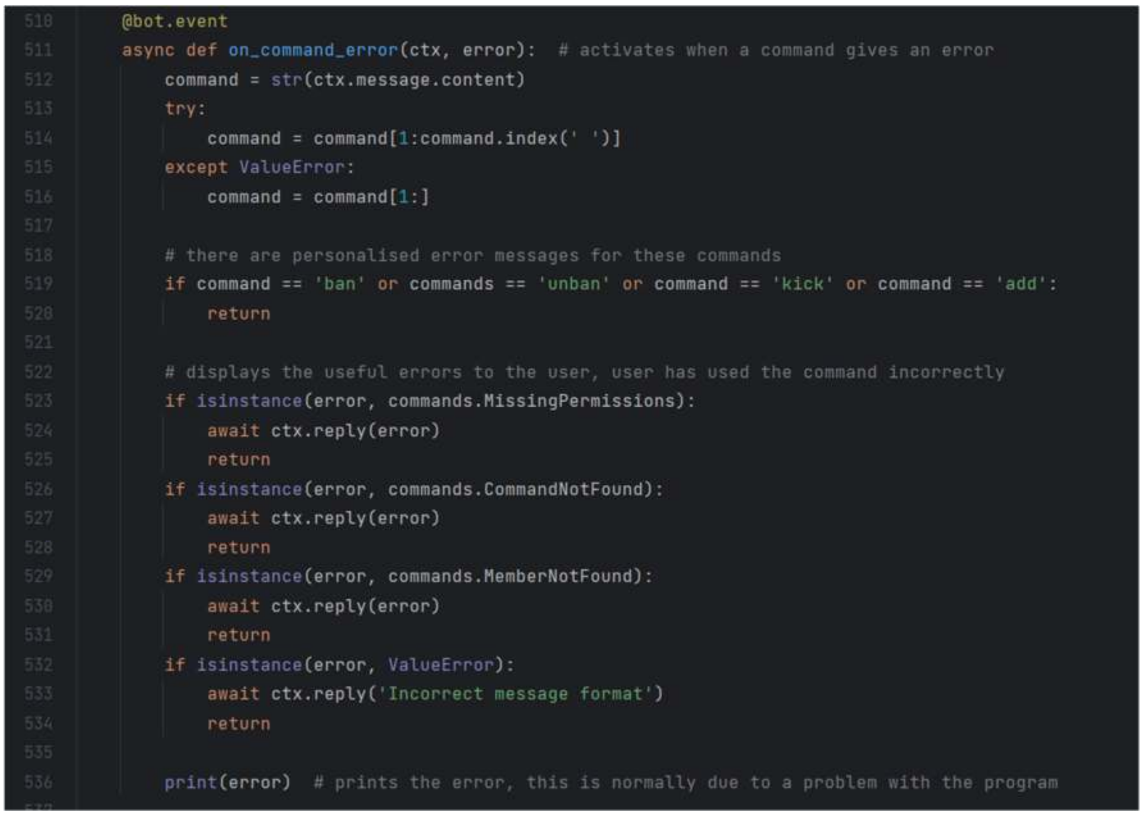
If the user is trying to add data to the Warnings.txt it checks the data being written has the correct syntax. If not, it gives a value error and sends an error message. This is necessary so the program does not crash achieving success criteria 10.



```python
if file == 'Warnings':
    try:
        _, _ = message.split(' ')
    except ValueError:
        await ctx.reply(
            'Incorrect message format for warnings\nCorrect format is !add Warnings [reason] [word]')
        return
```

6 Error checking to avoid program crashes:

When an error is given, it checks the type and sends an error message. These are the only errors that can be given by user inputs useful for success criteria 10.



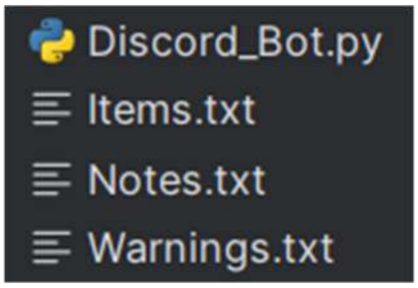
```python
@bot.event
async def on_command_error(ctx, error):  # activates when a command gives an error
    command = str(ctx.message.content)
    try:
        command = command[1:command.index(' ')]
    except ValueError:
        command = command[1:]

    # there are personalised error messages for these commands
    if command == 'ban' or commands == 'unban' or command == 'kick' or command == 'add':
        return

    # displays the useful errors to the user, user has used the command incorrectly
    if isinstance(error, commands.MissingPermissions):
        await ctx.reply(error)
        return
    if isinstance(error, commands.CommandNotFound):
        await ctx.reply(error)
        return
    if isinstance(error, commands.MemberNotFound):
        await ctx.reply(error)
        return
    if isinstance(error, ValueError):
        await ctx.reply('Incorrect message format')
        return

    print(error)  # prints the error, this is normally due to a problem with the program
```

7 Each function achieves one task:

Every function does one task. There are many simple commands, so each command activates one function, so users do not get confused. This helps extensibility and creates a clear structure.

8 Text files used to save data:

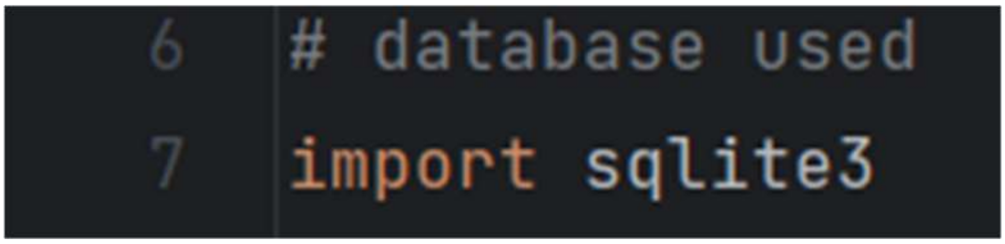
There are 3 text files. Admins can write in them for a better user experience.



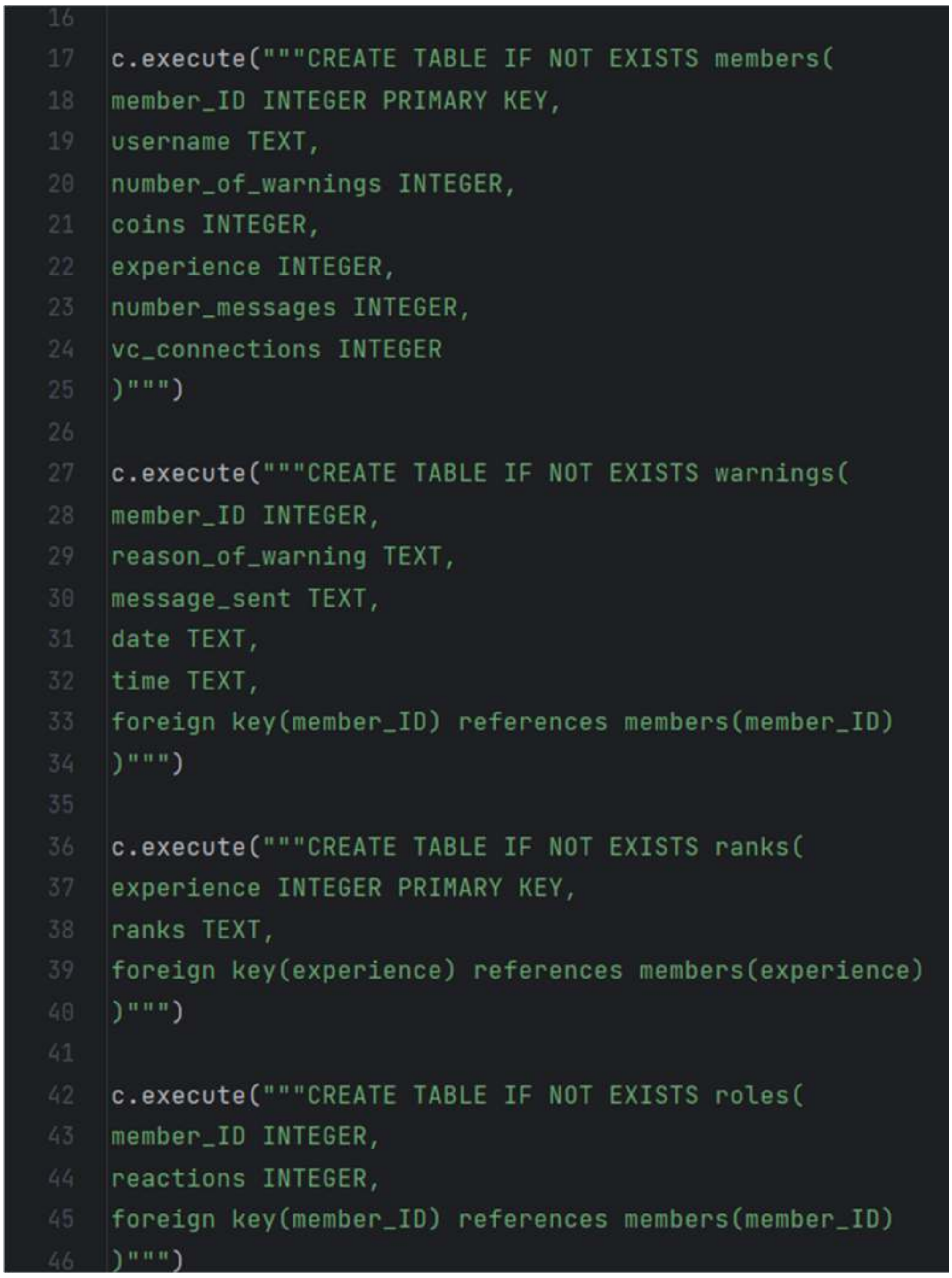
```
Discord_Bot.py
Items.txt
Notes.txt
Warnings.txt
```

9 Database used to save data:

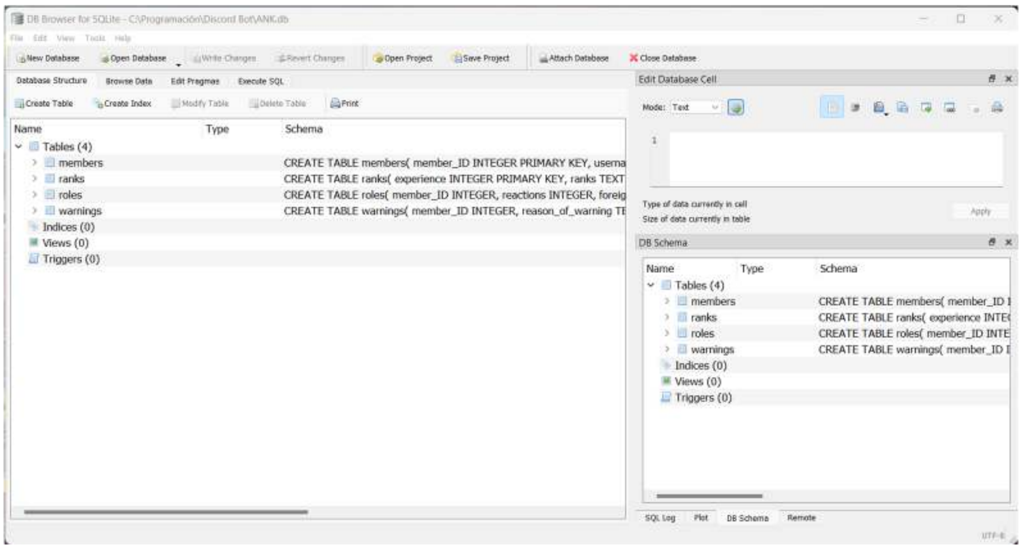
I used SQLite3 to create a database. Useful to save and update data which can be used in any subroutine and achieves success criteria 5.



```python
# database used
import sqlite3
```



```python

c.execute("""CREATE TABLE IF NOT EXISTS members(
member_ID INTEGER PRIMARY KEY,
username TEXT,
number_of_warnings INTEGER,
coins INTEGER,
experience INTEGER,
number_messages INTEGER,
vc_connections INTEGER
)""")

c.execute("""CREATE TABLE IF NOT EXISTS warnings(
member_ID INTEGER,
reason_of_warning TEXT,
message_sent TEXT,
date TEXT,
time TEXT,
foreign key(member_ID) references members(member_ID)
)""")

c.execute("""CREATE TABLE IF NOT EXISTS ranks(
experience INTEGER PRIMARY KEY,
ranks TEXT,
foreign key(experience) references members(experience)
)""")

c.execute("""CREATE TABLE IF NOT EXISTS roles(
member_ID INTEGER,
reactions INTEGER,
foreign key(member_ID) references members(member_ID)
)""")
```

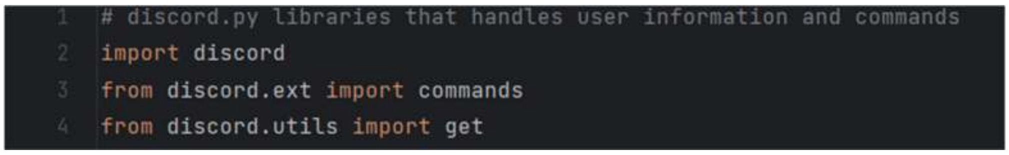


Name	Type	Schema
Tables (4)		
members		CREATE TABLE members(member_ID INTEGER PRIMARY KEY, userna
ranks		CREATE TABLE ranks(experience INTEGER PRIMARY KEY, ranks TEXT
roles		CREATE TABLE roles(member_ID INTEGER, reactions INTEGER, foreig
warnings		CREATE TABLE warnings(member_ID INTEGER, reason_of_warning TE
Indices (0)		
Views (0)		
Triggers (0)		

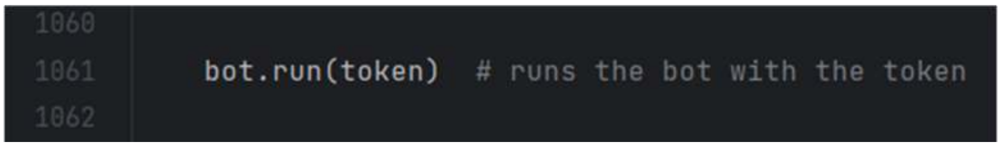
Name	Type	Schema
Tables (4)		
members		CREATE TABLE members(member_ID I
ranks		CREATE TABLE ranks(experience INTE
roles		CREATE TABLE roles(member_ID INTE
warnings		CREATE TABLE warnings(member_ID I
Indices (0)		
Views (0)		
Triggers (0)		

10 Remote API used for connection:

By using the discord.py library I connected to the Discord API. This allows my code to connect with the bot using the token. This connects the system to my client's guild.



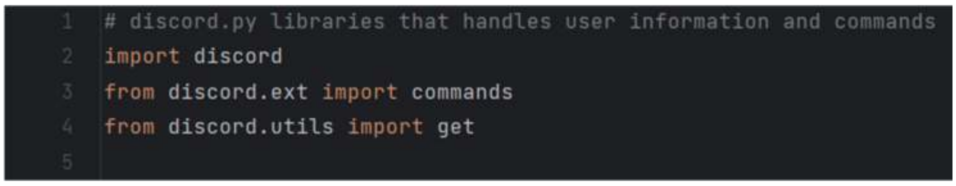
```python
# discord.py libraries that handles user information and commands
import discord
from discord.ext import commands
from discord.utils import get
```



```python

    bot.run(token)  # runs the bot with the token

```

11 Imported functionality from 3rd party libraries:

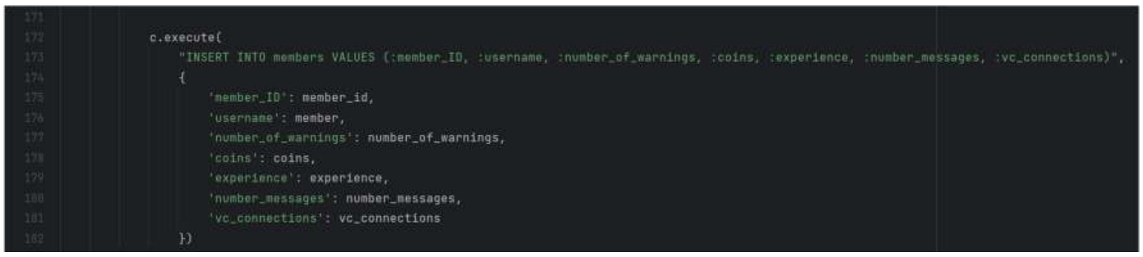
These libraries are from a 3rd party that I had to install and import to my code.



```python
# discord.py libraries that handles user information and commands
import discord
from discord.ext import commands
from discord.utils import get

```

12 Key-value pair dynamic data structure used:

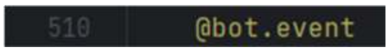
I used dictionaries to store and import data to the database. This dictionary stores the data of users when they join the guild and imports the data to the members table.



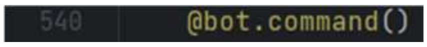
```python
            c.execute(
                "INSERT INTO members VALUES (:member_ID, :username, :number_of_warnings, :coins, :experience, :number_messages, :vc_connections)",
                {
                    'member_ID': member_id,
                    'username': member,
                    'number_of_warnings': number_of_warnings,
                    'coins': coins,
                    'experience': experience,
                    'number_messages': number_messages,
                    'vc_connections': vc_connections
                })
```

13 Decorator used for functions:

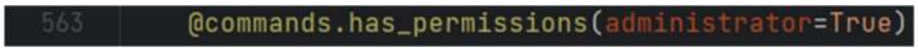
I used 4 decorators in the code. They are needed to add the functionalities required to the functions. Each decorator states the function type.



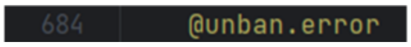
```python
    @bot.event
```



```python
    @bot.command()
```



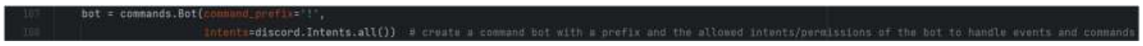
```python
    @commands.has_permissions(administrator=True)
```



```python
    @unban.error
```

14 Event and command handlers used:

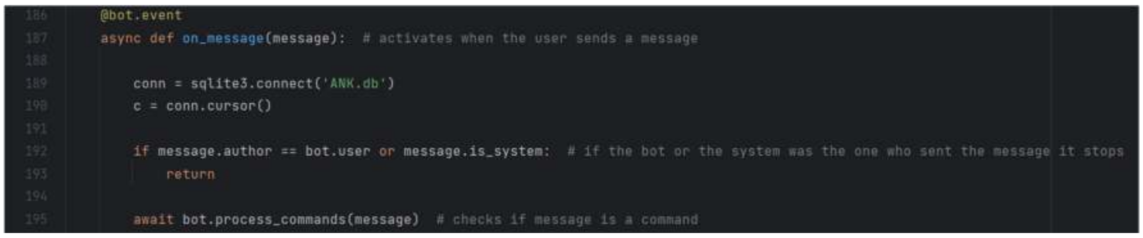
The variable bot is used as a handler for events and commands.



```python
bot = commands.Bot(command_prefix='!',
                   intents=discord.Intents.all())  # create a command bot with a prefix and the allowed intents/permissions of the bot to handle events and commands
```

The bot listens to the events and sends data through the API to discord.

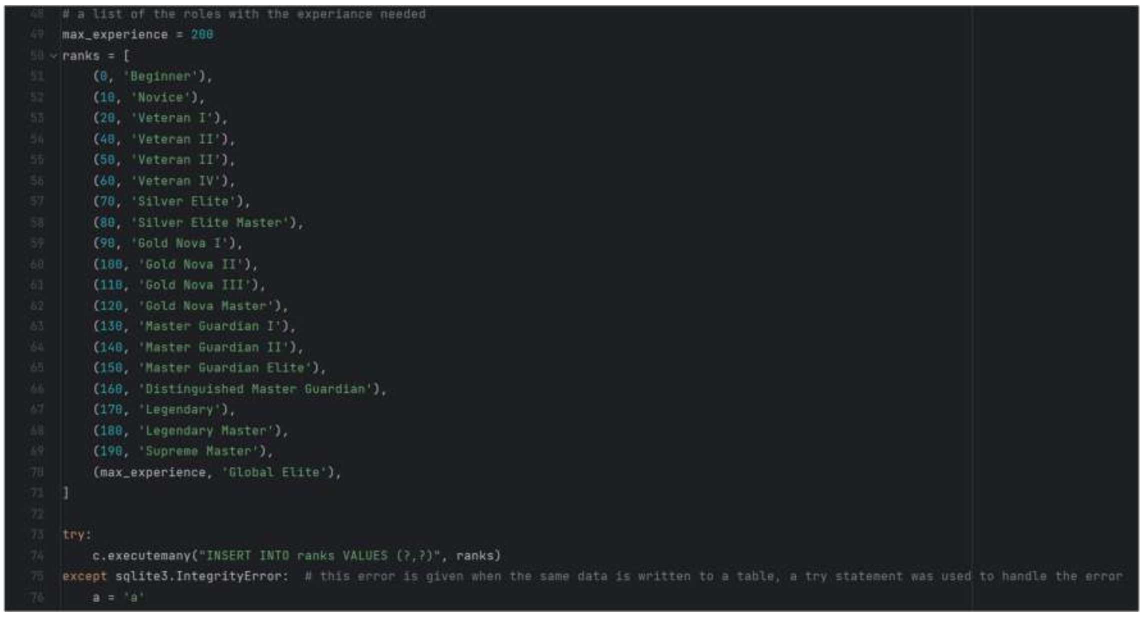
When a user types a command, it goes as a message. Therefore, I added a command processor in the event on_message.



```python
@bot.event
async def on_message(message):  # activates when the user sends a message

    conn = sqlite3.connect('ANK.db')
    c = conn.cursor()

    if message.author == bot.user or message.is_system:  # if the bot or the system was the one who sent the message it stops
        return

    await bot.process_commands(message)  # checks if message is a command
```

15 Try statement for exception handler used:

I used Try statements to avoid errors. Checking for data integrity when the code is run avoids crashes due to trying to input redundant(repeated) data and helps achieve success criteria 10.



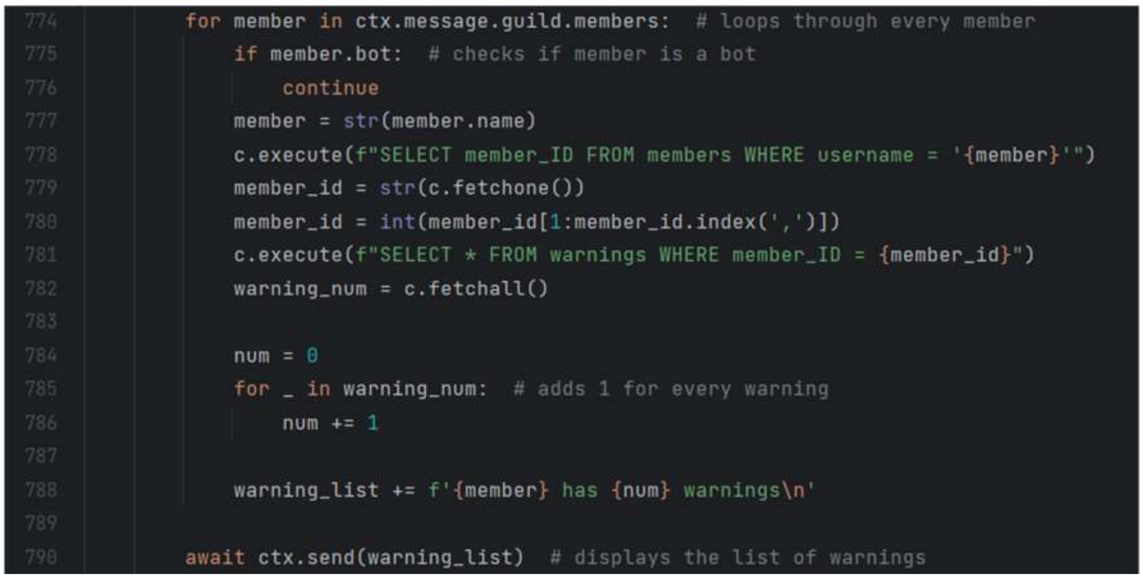
```python
# a list of the roles with the experiance needed
max_experience = 200
ranks = [
    (0, 'Beginner'),
    (10, 'Novice'),
    (20, 'Veteran I'),
    (40, 'Veteran II'),
    (50, 'Veteran II'),
    (60, 'Veteran IV'),
    (70, 'Silver Elite'),
    (80, 'Silver Elite Master'),
    (90, 'Gold Nova I'),
    (100, 'Gold Nova II'),
    (110, 'Gold Nova III'),
    (120, 'Gold Nova Master'),
    (130, 'Master Guardian I'),
    (140, 'Master Guardian II'),
    (150, 'Master Guardian Elite'),
    (160, 'Distinguished Master Guardian'),
    (170, 'Legendary'),
    (180, 'Legendary Master'),
    (190, 'Supreme Master'),
    (max_experience, 'Global Elite'),
]

try:
    c.executemany("INSERT INTO ranks VALUES (?,?)", ranks)
except sqlite3.IntegrityError:  # this error is given when the same data is written to a table, a try statement was used to handle the error
    a = 'a'
```

16 Nested loops and IF's

I used these techniques to loop or check through a value for every member of the guild.

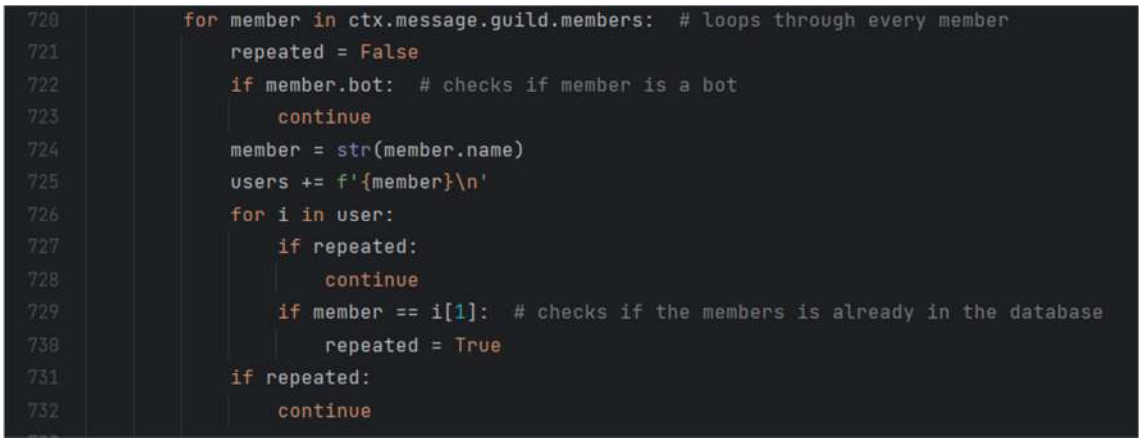
Most of these loops are in my major algorithms:

Warnings:



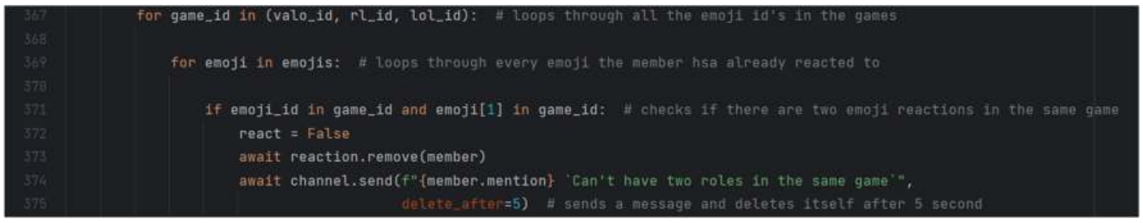
```python
        for member in ctx.message.guild.members:  # loops through every member
            if member.bot:  # checks if member is a bot
                continue
            member = str(member.name)
            c.execute(f"SELECT member_ID FROM members WHERE username = '{member}'")
            member_id = str(c.fetchone())
            member_id = int(member_id[1:member_id.index(',')])
            c.execute(f"SELECT * FROM warnings WHERE member_ID = {member_id}")
            warning_num = c.fetchall()

            num = 0
            for _ in warning_num:  # adds 1 for every warning
                num += 1

            warning_list += f'{member} has {num} warnings\n'

        await ctx.send(warning_list)  # displays the list of warnings
```

Member adding:



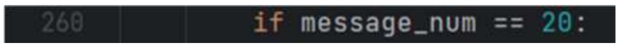
```
        for member in ctx.message.guild.members:  # loops through every member
            repeated = False
            if member.bot:  # checks if member is a bot
                continue
            member = str(member.name)
            users += f'{member}\n'
            for i in user:
                if repeated:
                    continue
                if member == i[1]:  # checks if the members is already in the database
                    repeated = True
            if repeated:
                continue
```

Emoji id:

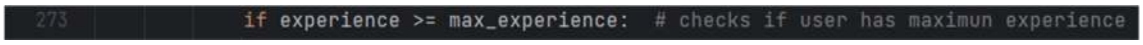


```
        for game_id in (valo_id, rl_id, lol_id):  # loops through all the emoji id's in the games

            for emoji in emojis:  # loops through every emoji the member hsa already reacted to

                if emoji_id in game_id and emoji[1] in game_id:  # checks if there are two emoji reactions in the same game
                    react = False
                    await reaction.remove(member)
                    await channel.send(f"{member.mention} `Can't have two roles in the same game`",
                                       delete_after=5)  # sends a message and deletes itself after 5 second
```

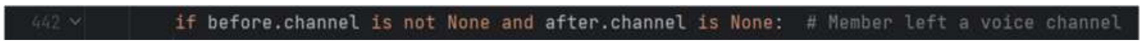
This checks only once per member or emoji, allowing the function to give the reward once per user and achieves success criteria 8. Checks when a member sends a total of 20 messages or exits a voice channel. To give a role if the experience gained is enough to reach a new rank.



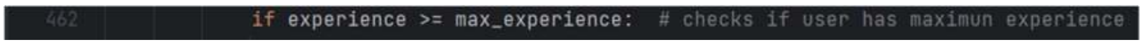
```
            if message_num == 20:
```



```
            if experience >= max_experience:  # checks if user has maximun experience
```



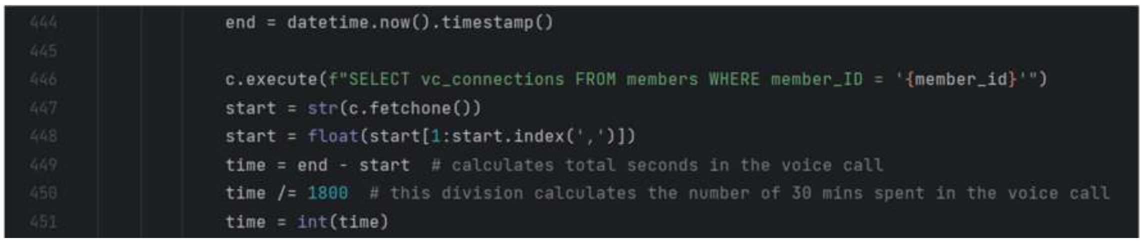
```
        if before.channel is not None and after.channel is None:  # Member left a voice channel
```



```
            if experience >= max_experience:  # checks if user has maximun experience
```

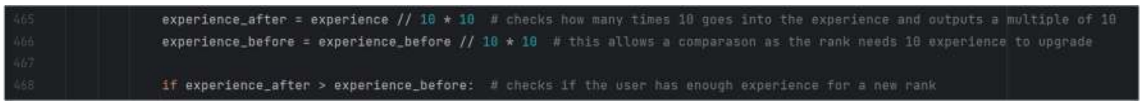
17 Calculations:

Calculates the amount of time for each member so it gives the right amount of reward and achieves success criteria 8.



```
            end = datetime.now().timestamp()

            c.execute(f"SELECT vc_connections FROM members WHERE member_ID = '{member_id}'")
            start = str(c.fetchone())
            start = float(start[1:start.index(',')])
            time = end - start  # calculates total seconds in the voice call
            time /= 1800  # this division calculates the number of 30 mins spent in the voice call
            time = int(time)
```

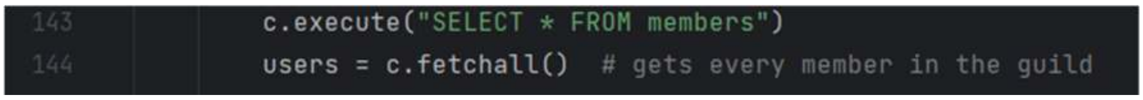
Calculations were used to check if the experience is enough to reach a new rank.



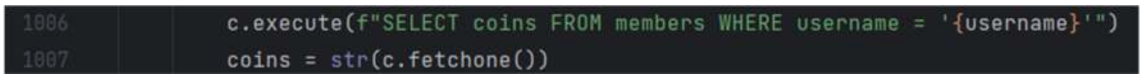
```
        experience_after = experience // 10 * 10  # checks how many times 10 goes into the experience and outputs a multiple of 10
        experience_before = experience_before // 10 * 10  # this allows a comparason as the rank needs 10 experience to upgrade

        if experience_after > experience_before:  # checks if the user has enough experience for a new rank
```

18 Searching:

This is vital to search values stored in the database. For every member or for specified members and/or value.



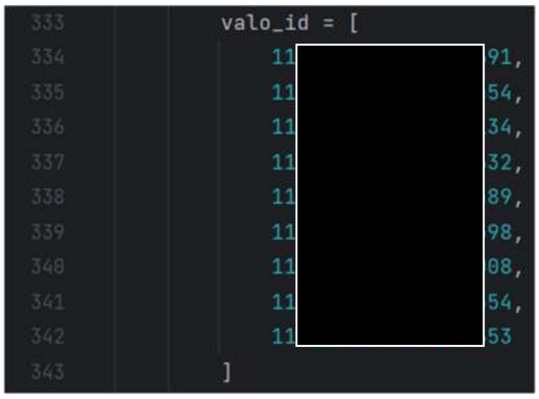
```
        c.execute("SELECT * FROM members")
        users = c.fetchall()  # gets every member in the guild
```



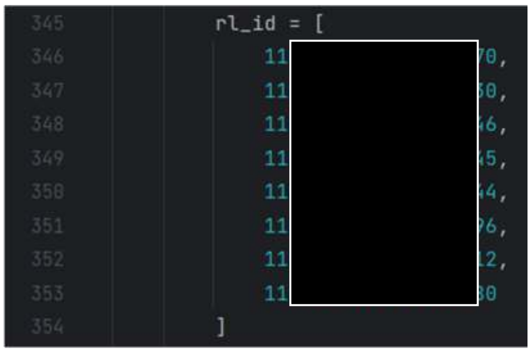
```
        c.execute(f"SELECT coins FROM members WHERE username = '{username}'")
        coins = str(c.fetchone())
```

19 Dynamic data structures:

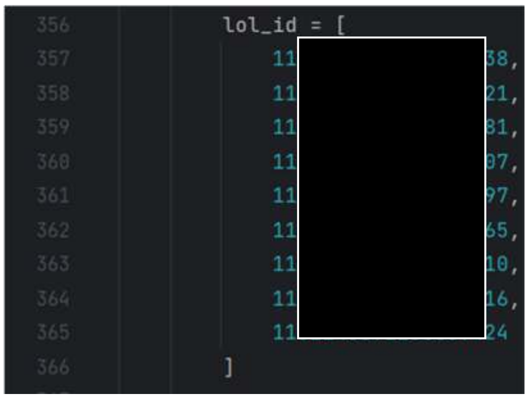
I used lists to store the id's of the emojis in the guild for the reactions.



```
    valo_id = [
        11[redacted]91,
        11          54,
        11          34,
        11          32,
        11          89,
        11          98,
        11          08,
        11          54,
        11          53
    ]
```



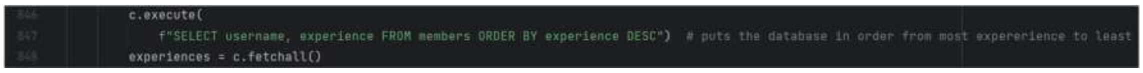
```
    rl_id = [
        11[redacted]70,
        11          60,
        11          46,
        11          45,
        11          44,
        11          96,
        11          12,
        11          30
    ]
```



```
lol_id = [
    11[redacted]38,
    11[redacted]21,
    11[redacted]81,
    11[redacted]07,
    11[redacted]97,
    11[redacted]65,
    11[redacted]10,
    11[redacted]16,
    11[redacted]24
]
```

20 Positioning:

Useful for the rank command to position members in descending order and display from highest to lowest experience.



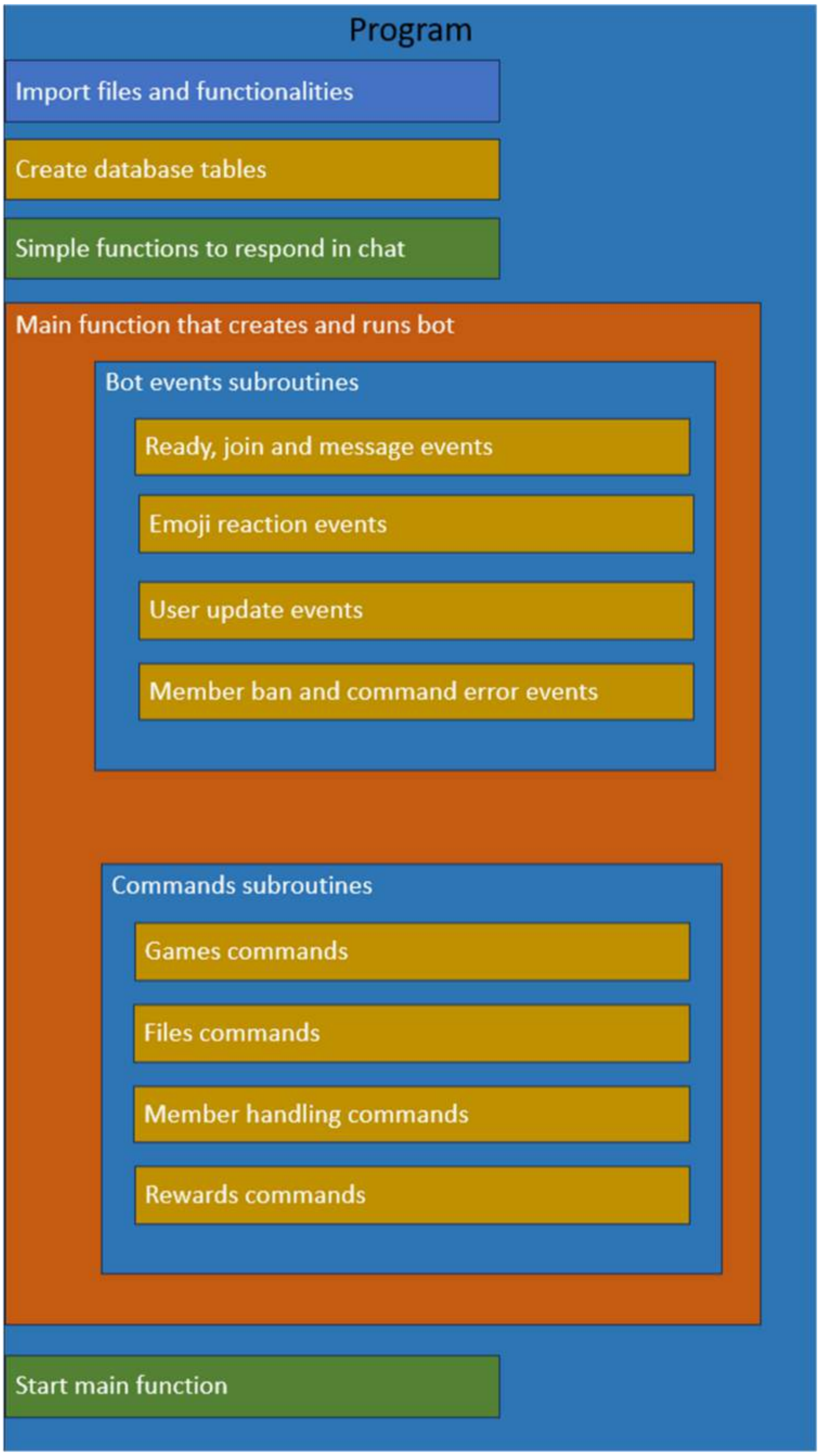
```
c.execute(
    f"SELECT username, experience FROM members ORDER BY experience DESC")  # puts the database in order from most expererience to least
experiences = c.fetchall()
```

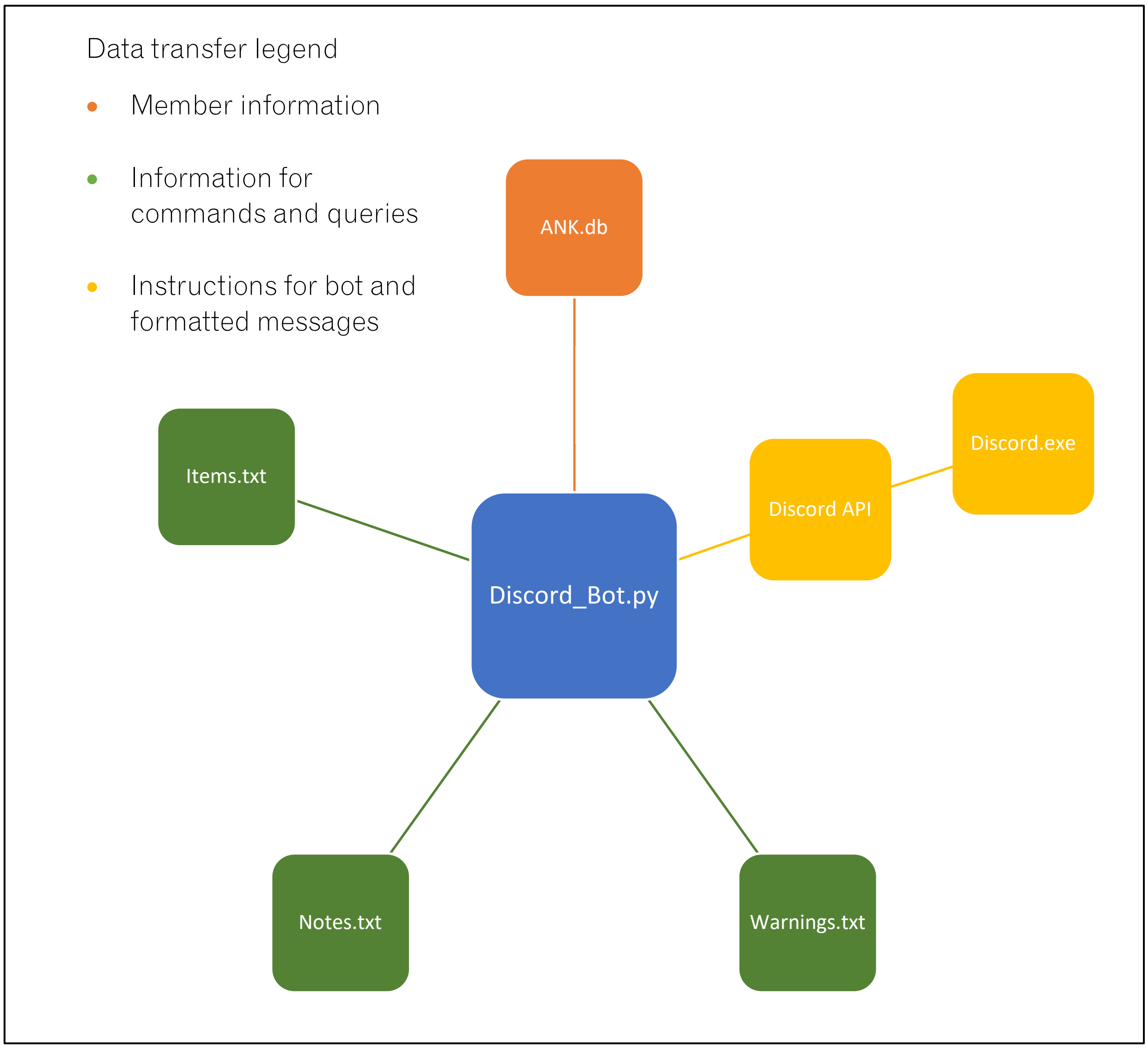
21 Web hosting used:

This allows the system to be always online achieving success criteria 3. The Replit online editor is used to host the system.

22 Diagrams of product

Diagrams help to understand the structure, increasing extensibility of the product.





Word Count: 1000

23 Bibliography:

"API Reference." *API Reference*, 2015, discordpy.readthedocs.io/en/stable/api.html. Accessed 13 May 2023.

Citron, Jason, and Stanislav Vishnevsky. "Discord Inc." *Discord*, 1.0.9013, 5 May 2023, https://discord.com/. Accessed 13 May 2023.

Elder, John. *SQLite Databases With Python - Full Course*. *YouTube*, YouTube, 12 May 2020, https://www.youtube.com/watch?v=byHcYRpMgI4&t=3986s. Accessed 7 May 2023.

Idently, director. *Create Your Own Discord Bot in Python 3.10 Tutorial (2022 Edition)*. *YouTube*, YouTube, 30 July 2022, https://www.youtube.com/watch?v=hoDLj0IzZMU. Accessed 7 May 2023.

Replit. "The Software Creation Platform. Ide, AI, and Deployments." *Replit*, 2023, replit.com/. Accessed 13 May 2023.