University of Brighton

PyBot: A Discord Assistant

Joshua Wallis

15839520

School of Computing, Engineering and Mathematics

Supervised by
James Burton

Acknowledgements

I would like to thank James Burton for his assistance and guidance during the project.

I Would also like to thank the members of the Discord.py community for being so helpful with any questions I had regarding both Python and their own API wrapper project.

Abstract

Discord is a proprietary freeware VoIP application designed for gaming communities. With over 87 million users, Discord has increased in popularity over applications such as Skype or Teamspeak and has taken the gaming world by storm[1]. Bots are created to help within the IRC servers or "guilds" that users are a part of. These bots help with their daily use of the server with commands to kick users or play music to everyone connected to a voice channel. This dissertation will cover my journey into Python and creating a new bot that will cater to the needs of users in my own personal server.

Terminology

Discord: A free VoIP software mainly used by gamers to chat via voice and text.

IRC: Internet Relay Chat. Facilitating text communication via client and server.

League of Legends: Multiplayer Online Battle Arena (MOBA) where two teams of 5 fight it out across multiple game modes and battlefields

API: A set of functions and procedures that allow the creation of applications which access the features or data of an operating system, application or other service.

API Wrapper: An API wrapper is a library that exposes a third-party API to us in a more friendly way.

Modules: A module is a file containing Python definitions and statements. The file name is the module name with the suffix ".py" appended.

Guild/Server: Guilds in Discord represent an isolated collection of users and channels, and are often referred to as "servers" in the UI.

Pip: Pip is a package management system used to install and manage software packages written in Python.

Asynchronous: Asynchronous programming is a means of parallel programming in which a unit of work runs separately from the main application thread and notifies the calling thread of its completion, failure or progress.

Coroutine: Coroutines are computer-program components that generalize subroutines for non-preemptive multitasking, by allowing multiple entry points for suspending and resuming execution at certain locations.

Contents

Chapter 1

Introduction

1.1 Project aim

The aim of my project is to explore a new language, Python, and make use of its features to develop a Discord bot that will be specialised for use in my personal Discord guild. I set out to create a better user experience within the guild by using the API for the game "League of Legends". This will let my members access player and character statistics without having to navigate through web and wiki pages to find the data they need. In addition to this I had the desire to explore Python and use its variety of modules and functions to create a fun experience as well. I also aimed to use the YouTube API to allow my guild members to enjoy music whilst they are playing games or having a casual chat in one of multiple voice channels. This gave me the added challenge of juggling multiple APIs in one application which I found to be demanding yet very rewarding once I got the hang of how they worked. I also wanted to explore how I could use asynchronous programming to allow the bot to function constantly without having to be refreshed each time a command was used or a song was played. Finally, my main goal was to deepen my knowledge of Python and take what I have learnt on my journey to pursue a career in it once I graduate.

1.2 Planning the project

At the start of my project I created a plan using a Gantt chart to have a clear image of what features I needed to implement and the time in which I had to do so. I originally decided to stick to the waterfall methodology as you can see in appendix A.1. As I progressed in my project I noticed that keeping to such a strict schedule for an application that needed to be developed in a relatively short amount of time was not ideal. This became most apparent when I got to the middle stages of my application and got delayed as writing some commands would break others meaning I was sometimes taking a step back when I expected to be moving forward at a steady pace. This caused some of my features to be delayed in the long run which led to me changing to taking a more agile approach to the project. Despite this, having the Gantt chart was still incredibly useful as I could visualise what I still had to get done and go back to reference what I estimated and how much time I can spend fixing a certain feature.

I also created a risk matrix that highlights some of the main risks I expected to face during the development of my project. The biggest risk I faced was relying on the main Discord servers being up and running during the project. If the servers went down for a week then my project would be greatly delayed due to needing connection to test and eliminate bugs. A diagram of the potential risks and their impact can be found in appendix A.2.

Along with this, I kept a diary that contained all of the issues I came across during my development of the project. I have included the meetings and diary in the appendix at A.3 and A.4 respectively.

1.3 The target user

This project is aimed at gamers in general as the population of Discord consists mainly of users that play or watch video games. Due to this, if I were to make the bot public I could create more functions and use more APIs to cater to users who would want support for a variety of other games. Anybody can access and use this application however as I have created a help guide that includes all of the commands available to the bot so you can say that this application is definitely an end user tool.

1.4 Use Case scenario

Discord bots are generally invaluable to include within a Discord guild as they can cover a great range of functions to assist the average user. However, I found that there are not that many bots that support multiple functions and only tend to focus on one specific feature at a time. Knowing this I decided to create a bot that would cover a wider range of functions that would assist users in retrieving data for games they play. By eliminating the need for people to tirelessly navigate multiple web pages just to end up getting fed up and leaving will make their lives much easier.

This would be incredibly helpful as my users will be able to write a simple 10 character message to retrieve all of the statistics they want in a single UI. My method will have its drawbacks however as the user will need to know the exact name of something they are searching for as I will not be able to use a search bar with a recommendation function like most websites do.

Chapter 2

Background Research

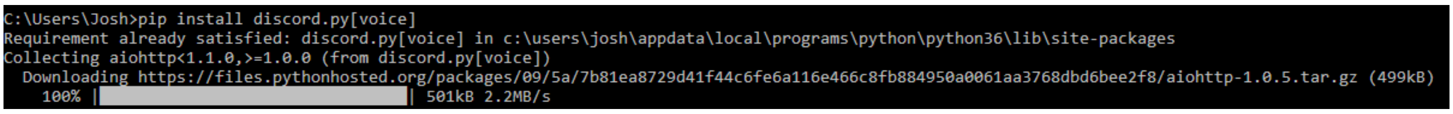
2.1 Discord.py [2]

Discord.py is the main API wrapper used during this project. It's a community created project that was initially made to allow easier writing for bots and got extended to allow the use of logs. Using Discord.py has been a great help as it handles requests for you which eliminates the need to write many lines of code.

As Discord.py is a community project, there is an online community that is active in their own Discord guild. This is the place where they mainly communicate and assist others in creating their own bots and developing the Discord.py API further.

2.1.1 So how does it work?

The first thing I needed to do was install all of the modules I was going to need by using "pip". The way I did this was by using the command prompt and navigating to my Python directory. From there I would type "pip install (packagename)" and it would handle the download of multiple modules that I need to use throughout my project such as aiohttp and Discord.py[voice] itself.



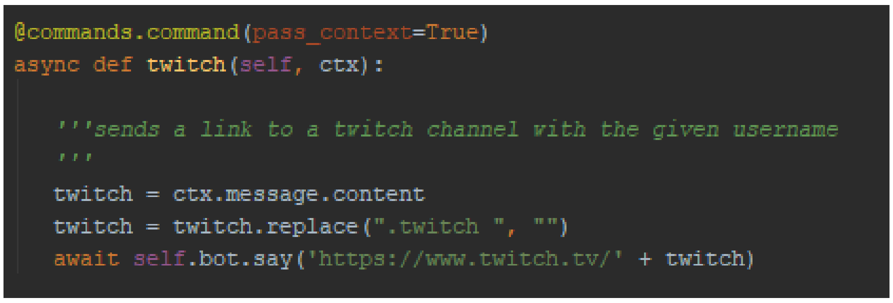
```
C:\Users\Josh>pip install discord.py[voice]
Requirement already satisfied: discord.py[voice] in c:\users\josh\appdata\local\programs\python\python36\lib\site-packages
Collecting aiohttp<1.1.0,>=1.0.0 (from discord.py[voice])
  Downloading https://files.pythonhosted.org/packages/09/5a/7b81ea8729d41f44c6fe6a116e466c8fb884950a0061aa3768dbd6bee2f8/aiohttp-1.0.5.tar.gz (499kB)
    100% |████████████████████████████████| 501kB 2.2MB/s
```

Figure 2.1: Pip install of the Discord.py[voice] package

After this I had to define the prefix I was going to use that would detect whether a command was going to be input. In my case I decided to use "." as it is memorable and looks quite clean as opposed to "?" or "-". When I created my prefix I could then go on to create my command methods.

To define a command method you need to annotate each function with "@commands.command". This will register that function to the commands of your bot and will allow you to create a function named "ping" which would be called in the Discord interface with ".ping". To get these functions to work you need to program asynchronously in Python by importing the asyncio module and defining each of your methods with "async def" instead of just "def" which is the standard Python procedure.

To ensure the bot then understand these commands you must pass the parameter "ctx" which tells the bot the context of command. For example if I wanted my bot to send a message in the general channel that would link a person to the streaming website Twitch.tv:



```python
@commands.command(pass_context=True)
async def twitch(self, ctx):

    '''sends a link to a twitch channel with the given username
    '''
    twitch = ctx.message.content
    twitch = twitch.replace(".twitch ", "")
    await self.bot.say('https://www.twitch.tv/' + twitch)
```

Figure 2.2: Twitch command taking user input as the context

The bot would be able to understand the context of the message (".twitch rdxjw") and then carry out the function as the command by taking the argument "rdxjw" and appending it to the end of the url that gets created.

In addition to this, the aiohttp module was also required to be used with this bot. This allows you to send http requests with the bot that will allow you to connect to the actual Discord servers with your bots authorisation key. It is also useful when using APIs that connect to databases and websites such as YouTube.

I have included a class diagram that shows every script and command/method contained within them to give a bit more insight on how they connect after I have discussed each module and class involved in the process.

2.2 Other APIs I have used

2.2.1 League of Legends [3]

The API for the game League of Legends allows you to extract information about in game characters and the statistics of the players themselves. The way I went about this was to use an API wrapper created by someone I spoke to within the Discord.py community and he was glad to allow me to test and use his wrapper[7].

He wrote the wrapper in a way that eliminated the need for endless lines of JSON code to retrieve the information from their databases and instead went for a more object oriented approach. This was highly beneficial to me as I have studied object oriented Java over the past years of studying so I could familiarise myself quite quickly.

The API does have a drawback however. An API key is needed to access and view the data kept by Riot Games. As I am just a general member of the League of Legends community there is a strict cap on the amount of requests you can make with your own personal API key so you can not flood the databases and cause them to crash. To combat this I had to apply for a permanent key that had its caps removed every day so the bot on my server could have continuous use.

2.2.2 Youtube dl [4]

Youtube-dl is a command line utility that allows you to stream music from YouTube, Dailymotion and Vimeo. This was the third and last API I decided to use as I did not want to juggle too many at once. The way Youtube-dl works is by downloading a video and keeping it in temporary storage so that it can be played, paused, resumed and stopped. In addition to this you can also pull information about the video from the download and view things such as the title, video duration, file size and upload date all by entering simple commands.

2.2.3 The API Design

The design of these APIs is incredibly well done and were so easy to use once you understood them. This meant that my development was very smooth and I did not come across too many issues. The design of these APIs meant that my code was extremely concise which helped make my bot run fast and commands are handled with speed and ease. This also meant that there is a much smaller delay in the response time with my commands so members of my guild would not be waiting minutes to see the statistics appear for a game or a video to load.

I have created a UML diagram demonstrating the APIs and how they connect to the mainconnections bot that can be found at appendix B.1.

2.3 Selecting the right language to use

Python has been a language I have shown interest in during my time studying at university so I decided it would be a good personal challenge for the project to learn it from scratch and apply it to this project.

In addition to this it is a good language to use when creating a bot for Discord as Python is a language that is an interpreted language rather than compiled like Java. This means that it is a lot easier to implement code in Python and code can be executed on the fly. This made my development a lot faster especially as it was a new language to me.

Python has an IDE called PyCharm which was of great use to me. PyCharm was developed by JetBrains who also created IntelliJ which has been the IDE of my choice over the past two years programming in Java. This allowed me to get really familiar with developing in Python fairly quickly as I did not have to get stuck figuring out how to run my application and how to debug it.

2.4 Learning Python

When it came to actually having to learn Python I decided it would be best to spend the best part of November reading ”Learning Python” [8] and watching online lectures to get a good grasp of the basics to understand how Python works. This also allowed me to get a picture of how I can implement all of the requirements for my bot and if I would need to make adjustments to my project plan.

After this I created a few small programs and got to grips with programming using the syntax of Python as I was still so used to Java. This then lead me to getting a start on my project and getting a deeper understanding of Python by creating a larger application and working on errors I came across on the fly.

2.5 The Discord.py community

The Discord.py developer community has its own guild that is used to discuss, rewrite and help other developers with their projects. You can be a completely new programmer or a veteran and still have access to many professionals and original creators opinions and guidance.

My most notable interaction in the community was when I had my entire project in a single script which looked very messy and inefficient. I sought out help from various members within the guild and some of the main suggestions I had were to run my bot using modules such as cogs to create classes and run my bot asynchronously so that I could split my code up into modules and run them all at the same time.

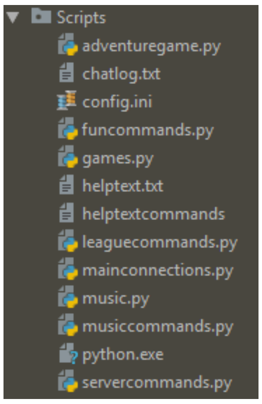
Despite the amount of help and guidance available in the guild, the members would never spoon feed you the code you can implement. You would always get given recommendations on the best ways to implement the feature you want which allowed you to understand the API wrapper itself rather than just relying on others to work it out for you. This pushed me to become a much more efficient and professional developer and gave me the independence and confidence in my abilities even though I received a small amount of guidance.

Chapter 3

Implementation

3.1 Code Structure

At the start of my development I had all of my code contained in a single script which made my bot inefficient and stopped certain commands from functioning entirely. After some meetings and seeking help within the Discord.py community I managed to split my code up into multiple modules and have them run asynchronously. This made my bot much faster and able to process multiple commands at once from many different users.



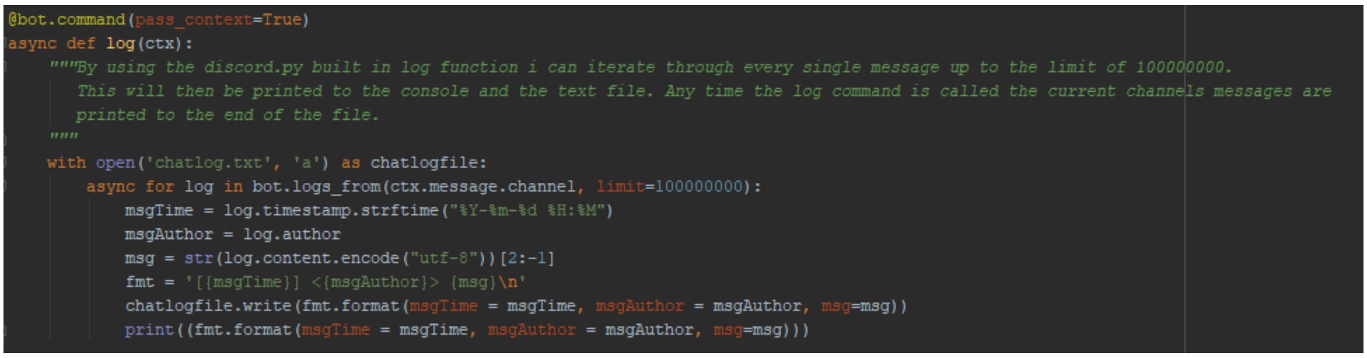
```
▼ Scripts
    adventuregame.py
    chatlog.txt
    config.ini
    funcommands.py
    games.py
    helptext.txt
    helptextcommands
    leaguecommands.py
    mainconnections.py
    music.py
    musiccommands.py
    python.exe
    servercommands.py
```

Figure 3.1: Code structure for PyBot

3.2 Modules

3.2.1 Main connections

Mainconnections is the module that handles connecting to the Discord servers to allow the bot to go online. This is a security measure as you need an individual key that your bot uses to access the server to run its commands and send them to your own guild. This is also where the prefix for the bot is defined as being "." which is used before any command as an identifier. In addition to this, the logging of the text chat rooms is handled here and the asynchronous events that always need to have a listener such as a new member joining the server to send them a welcome message and loading all of the individual modules through the use of cogs.



```python
@bot.command(pass_context=True)
async def log(ctx):
    """By using the discord.py built in log function i can iterate through every single message up to the limit of 100000000.
      This will then be printed to the console and the text file. Any time the log command is called the current channels messages are
      printed to the end of the file.
    """
    with open('chatlog.txt', 'a') as chatlogfile:
        async for log in bot.logs_from(ctx.message.channel, limit=100000000):
            msgTime = log.timestamp.strftime("%Y-%m-%d %H:%M")
            msgAuthor = log.author
            msg = str(log.content.encode("utf-8"))[2:-1]
            fmt = '[{msgTime}] <{msgAuthor}> {msg}\n'
            chatlogfile.write(fmt.format(msgTime = msgTime, msgAuthor = msgAuthor, msg=msg))
            print((fmt.format(msgTime = msgTime, msgAuthor = msgAuthor, msg=msg)))
```

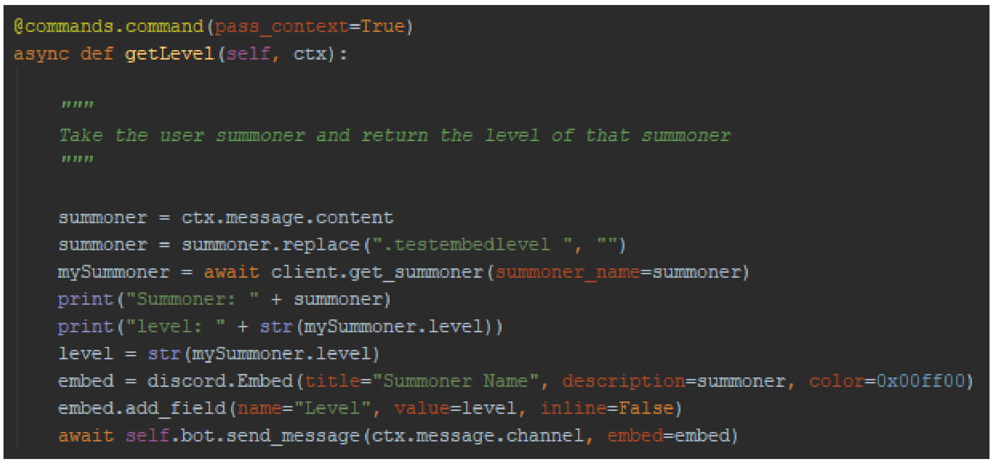
Figure 3.2: Asynchronous event of logging the text channels

3.2.2 Games

The games module of my bot just contains some games such as rock paper scissors and a magic 8 ball. These were created first in my bot so I could experiment with different ways to handle input and output with the Discord.py API. I also made use of the random module that would take a random integer to determine what item to select as the result of the rock paper scissors game or the magic 8 ball response.

3.2.3 League commands

This is where the code for the League of Legends API is handled. It starts off by reading the config file to get the API key that is used to access the Riot Games database. The commands in this module will take the input of the user in either the form of a character name or a summoner (player) name. This is then returned as an object that is then used to access the database on the back end and retrieve data such as the character statistics or the information about a summoner. The data retrieved is then taken and printed back by the bot to display the information in the Discord client without the user needing to navigate through multiple web pages to find this information themselves. Due to the way the API wrapper works it was not possible to iterate through entries within their database so I had to retrieve them all manually and embed them into a response by the bot separately.



```python
@commands.command(pass_context=True)
async def getLevel(self, ctx):

    """
    Take the user summoner and return the level of that summoner
    """

    summoner = ctx.message.content
    summoner = summoner.replace(".testembedlevel ", "")
    mySummoner = await client.get_summoner(summoner_name=summoner)
    print("Summoner: " + summoner)
    print("level: " + str(mySummoner.level))
    level = str(mySummoner.level)
    embed = discord.Embed(title="Summoner Name", description=summoner, color=0x00ff00)
    embed.add_field(name="Level", value=level, inline=False)
    await self.bot.send_message(ctx.message.channel, embed=embed)
```

Figure 3.3: Object oriented approach to Python programming in leaguecommands

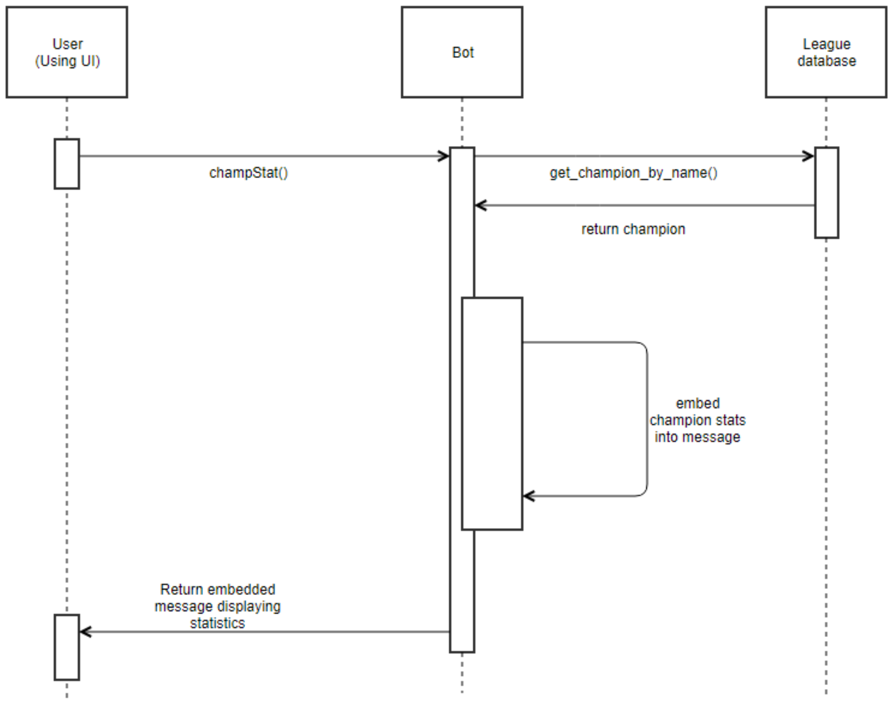
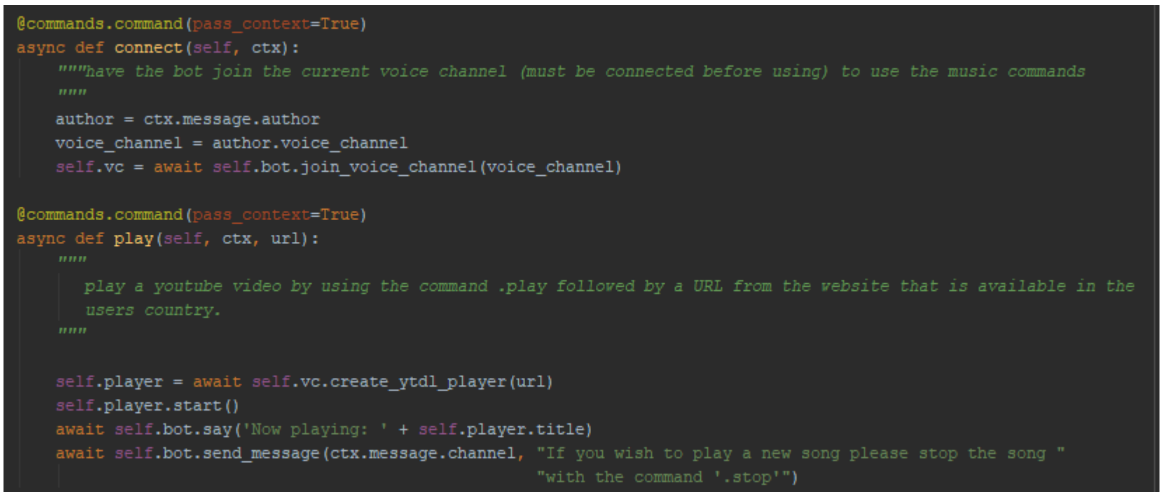


Figure 3.4: sequence diagram of retrieving a champion object from the league database and returning the statistics. It takes the input of the user command ".champStat (champion name)" and will search the API database. The champion is then returned as an object and the bot will retrieve the statistics from the object. This information is then embedded into a message and returned to the user.

3.2.4 Music commands

The music commands module is pretty short due to the design of the Discord.py API. With the youtube-dl module I can pull the video information and play the audio by creating a "player" object using the youtube-dl API and then having the bot join the voice channel from where the bot was called from. Originally I could not pause or stop the song as I was not able to reference the player from other commands. This is what led me to use the Cogs module to create classes and store them in an array in the mainconnections module. These cogs then get loaded up as mentioned earlier and the bot is able to run multiple scripts at once. In doing this I could reference the player in other commands by using the "self" argument. This would allow me to control the player by using stop, start and pause functions built into the Discord.py API in their own commands.



```
@commands.command(pass_context=True)
async def connect(self, ctx):
    """have the bot join the current voice channel (must be connected before using) to use the music commands
    """
    author = ctx.message.author
    voice_channel = author.voice_channel
    self.vc = await self.bot.join_voice_channel(voice_channel)

@commands.command(pass_context=True)
async def play(self, ctx, url):
    """
      play a youtube video by using the command .play followed by a URL from the website that is available in the
      users country.
    """

    self.player = await self.vc.create_ytdl_player(url)
    self.player.start()
    await self.bot.say('Now playing: ' + self.player.title)
    await self.bot.send_message(ctx.message.channel, "If you wish to play a new song please stop the song "
                                                     "with the command '.stop'")
```

Figure 3.5: First use of Python classes to use the music player in other methods

3.2.5 Server commands

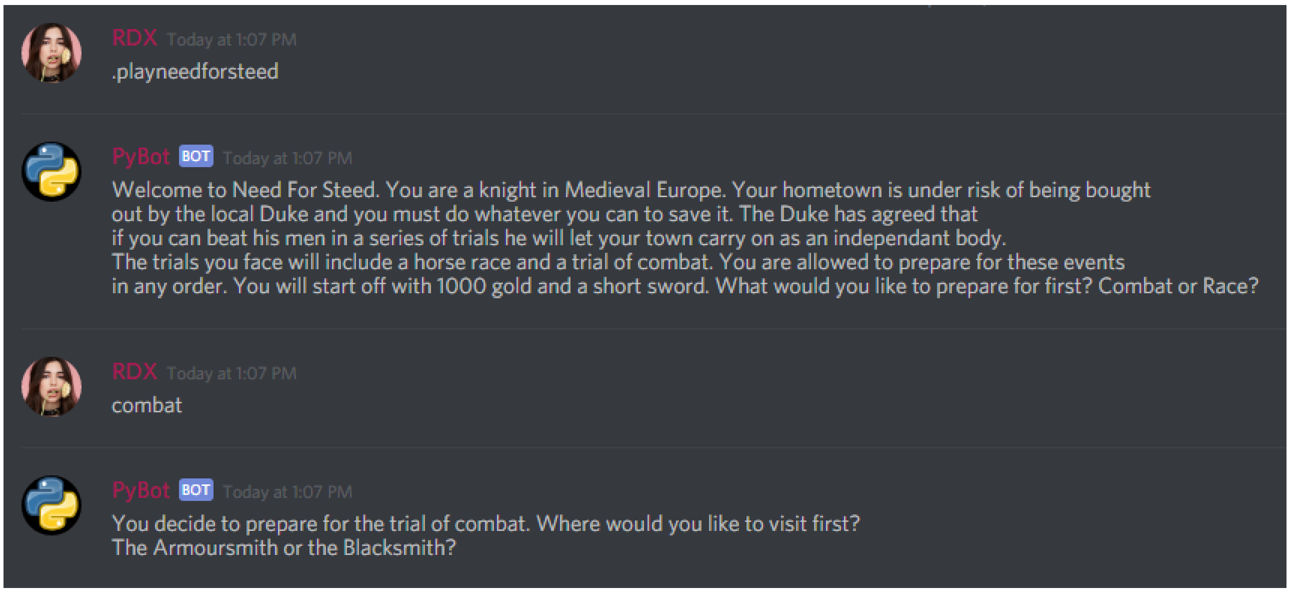
The server commands module was split off from the mainconnections module after I realised that the sheer amount of things I could do with management commands was so high that it needed its own module. The main commands in this module deal with management such as kicking a member, banning a member, moving a member from one channel to another, unbanning members, and displaying information like the users avatar and the game they are currently playing.

3.2.6 Fun commands

This module was created with the intention of having commands that aren't particularly related to anything in the API wrappers. They mainly server as a bit of fun or convenience such as being able to search for a twitch channel, an instagram page or posting a reaction image. They work by taking the relevant command ".instagram" or ".twitch" and taking the argument of the username which would be the second word in the commands message I.e ".instagram joshhwallis". The only drawback to this is that you need to know the exact username of the page you are creating a link to as this does not support actual searches.

3.2.7 Adventure Game

I decided to fiddle around with some more functions in Python and create a text based adventure game. For this I imported the time module that would have a delay between the bot's response and the user input to give it a more realistic conversation feel. This creates the distinction between an action being described such as a door being opened and the realism of a character's dialogue. This module uses the "on_message" function in the Discord.py wrapper which constantly listens for a certain string of characters contained within a message. For example the first line to start the game is shown below in figure 3.5. This will start the game and take



RDX Today at 1:07 PM
.playneedforsteed

PyBot BOT Today at 1:07 PM
Welcome to Need For Steed. You are a knight in Medieval Europe. Your hometown is under risk of being bought out by the local Duke and you must do whatever you can to save it. The Duke has agreed that if you can beat his men in a series of trials he will let your town carry on as an independant body.
The trials you face will include a horse race and a trial of combat. You are allowed to prepare for these events in any order. You will start off with 1000 gold and a short sword. What would you like to prepare for first? Combat or Race?

RDX Today at 1:07 PM
combat

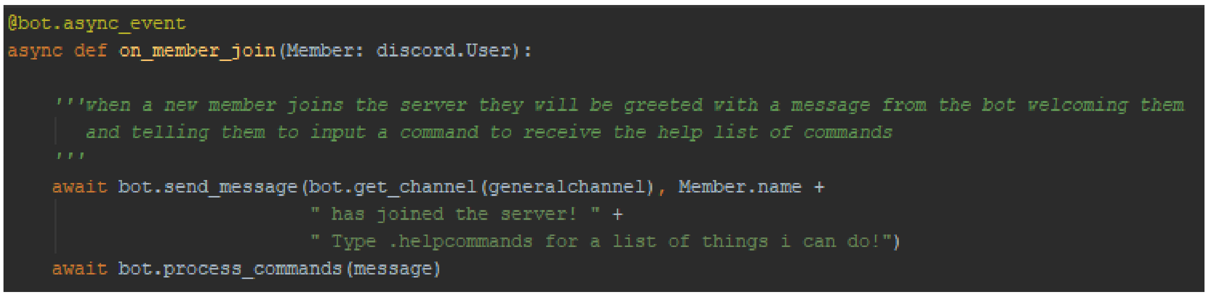
PyBot BOT Today at 1:07 PM
You decide to prepare for the trial of combat. Where would you like to visit first?
The Armoursmith or the Blacksmith?

Figure 3.6: Detection of user input without using commands

responses based on user input as opposed to using the commands feature of Discord.py which emulates a more true command line based text game.

3.2.8 Asyncio[5]

Asyncio is the most important module in this project as without it nothing would work properly. In asynchronous programming you will never know what order the execution of your commands will be, so to solve this asyncio allows you to run multiple commands concurrently on a single thread. This is done by using event loops and coroutines that will pause and resume execution of code in a program. By doing this I was able to have all of these commands essentially be in a paused state until I needed to run one, then when it finishes executing the program looped back to its previous state to wait for another command all while potentially executing another 5 for example from other users at the same time.



```
@bot.async_event
async def on_member_join(Member: discord.User):

    '''when a new member joins the server they will be greeted with a message from the bot welcoming them
       and telling them to input a command to receive the help list of commands
    '''
    await bot.send_message(bot.get_channel(generalchannel), Member.name +
                           " has joined the server! " +
                           " Type .helpcommands for a list of things i can do!")
    await bot.process_commands(message)
```

Figure 3.7: Function that has a constant uptime that runs concurrently with commands

3.2.9 Aiohttp[6]

Aiohttp is another important module that allows my program to function by handling http requests to send and retrieve data. This module allows me to send requests out with the youtube API to retrieve song information, the Discord.py API to set up the main connection to the Discord server and the League of Legends API to pull the objects from their database and retrieve specific information related to said object.

3.2.10 Other files

Config

When I was creating my bot I had a lot of hard coded variables such as the specific channels I wanted a message to be sent to or the API keys that were used for verification. Python is good at working with files so I created a .ini config file to store variables that could be changed and accessed by the program. This eliminated my hardcoded variables and will allow other users to apply their own API key if they decided to use my bot for their own guilds. This eliminates the need for the end user to have to access the source code of the application and can change the config file.

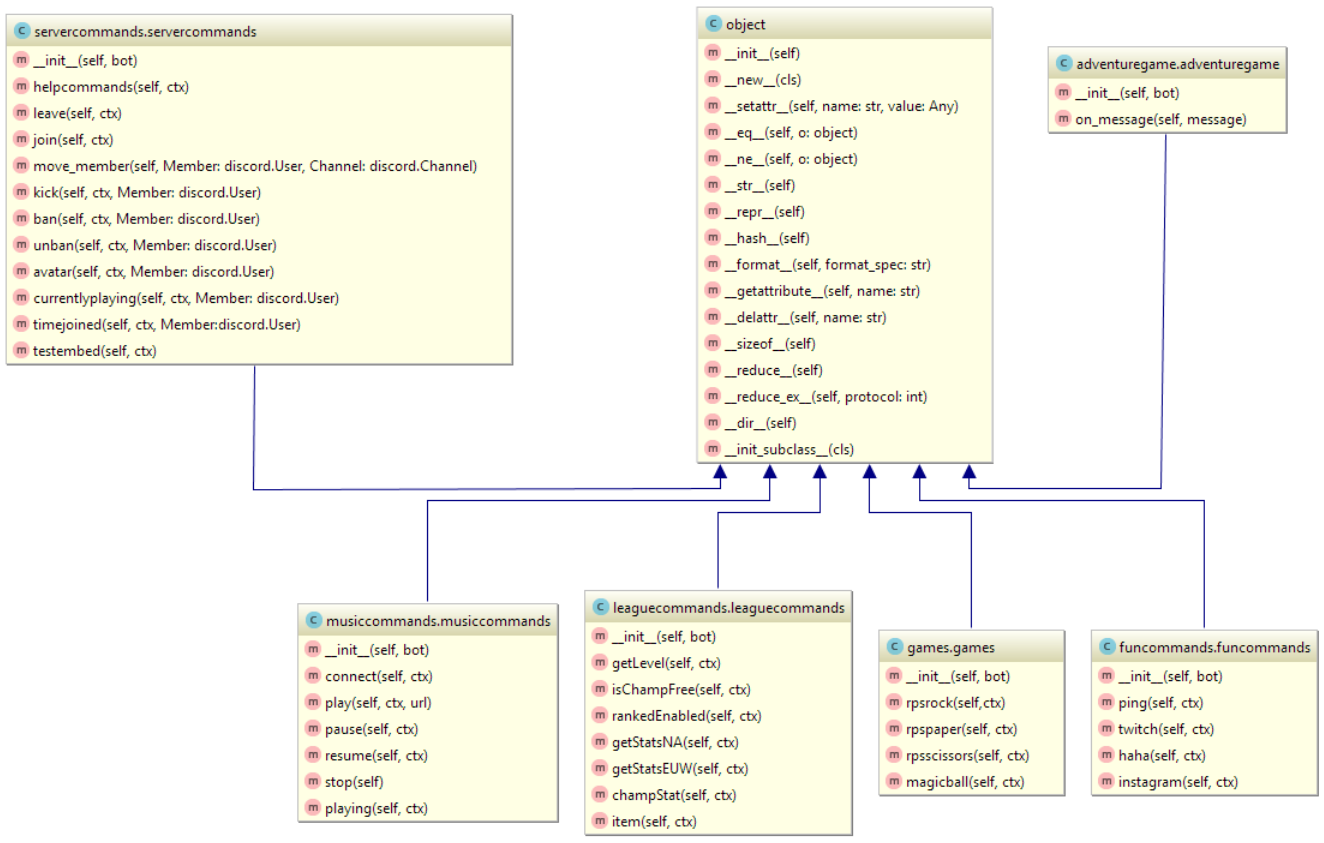
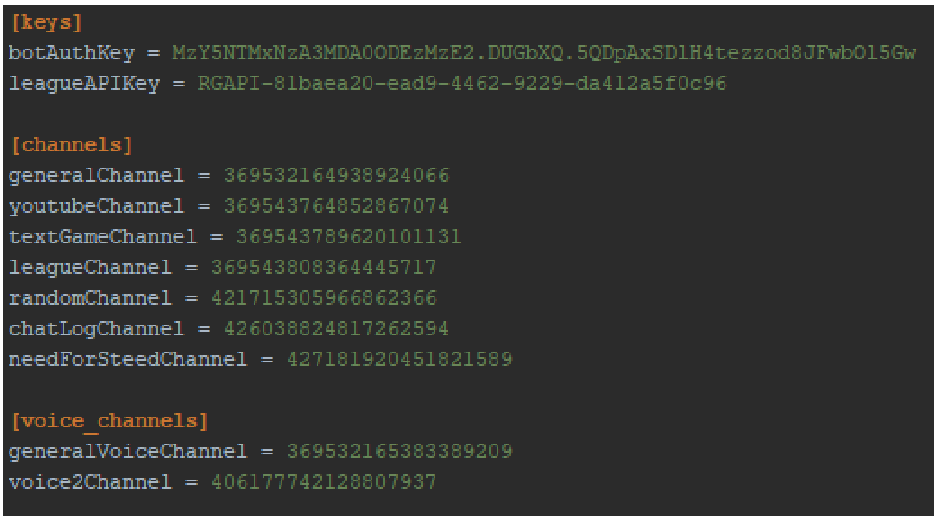


Figure 3.8: Each script within my application including every function connecting to the main bot object

Chat log

The chat log is used to keep track of what is said and done within the guild. This can be useful if there is a certain dispute and someone decided to delete their message that was worthy of a ban. By keeping a log you can keep the guild nice and friendly and give a user the proper punishment they deserve such as a temporary kick or permanent ban. This is done by using the ".log" command found in the mainconnections module. It iterates through every message that was posted in a specific channel and writes it to the end of a .txt log file with the date, time and message.



```
[keys]
botAuthKey = MzY5NTMxNzA3MDA0ODEzMzE2.DUGbXQ.5QDpAxSDlH4tezzod8JFwbO15Gw
leagueAPIKey = RGAPI-81baea20-ead9-4462-9229-da412a5f0c96

[channels]
generalChannel = 369532164938924066
youtubeChannel = 369543764852867074
textGameChannel = 369543789620101131
leagueChannel = 369543808364445717
randomChannel = 421715305966862366
chatLogChannel = 426038824817262594
needForSteedChannel = 427181920451821589

[voice_channels]
generalVoiceChannel = 369532165383389209
voice2Channel = 406177742128807937
```

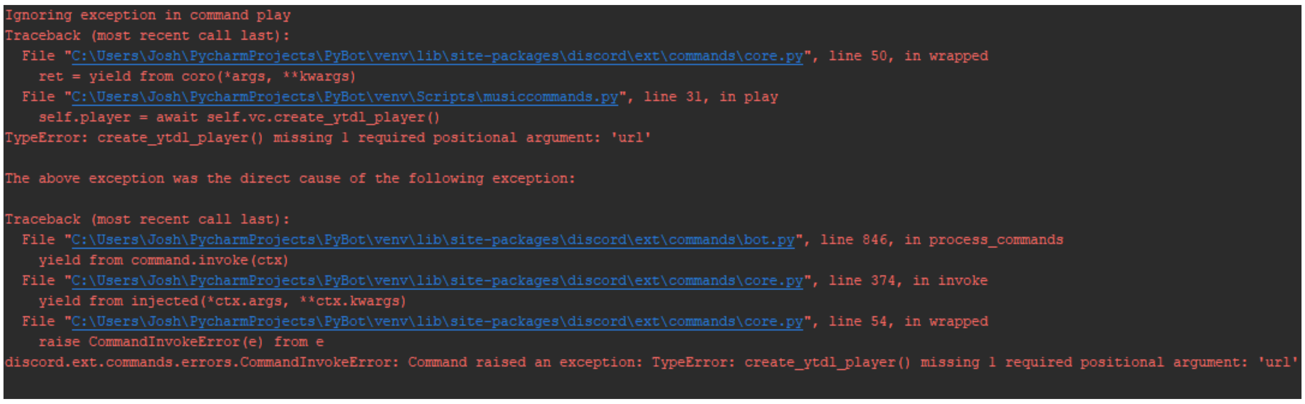
Figure 3.9: .ini config file to store variables

Help commands

The helpcommands text file just contains a list of every commands and what it does within the guild. This is sent to the user when they type in the ".helpcommands" command to show them the entire list. If they want a more extensive list they can just use ".help" to post the commands in a guild text channel, but that is a lot more inconvenient as other people may flood the channel with messages and the user will not be able to read the help text.

3.3 Logging

Discord.py and PyCharm have incredible logging features that would show me everything that is wrong with my code that was preventing a certain command from working. This console output helped immensely when developing my bot as I could correct errors in a matter of minutes if it was just a small syntax error. For example in Figure (whatever) I tried to run the command ".play (youtube link)" but I forgot to pass a url in the player method. The console showed me exactly what was wrong with the solution being to just add "url" to the parameters needed in the player.



```
Ignoring exception in command play
Traceback (most recent call last):
  File "C:\Users\Josh\PycharmProjects\PyBot\venv\lib\site-packages\discord\ext\commands\core.py", line 50, in wrapped
    ret = yield from coro(*args, **kwargs)
  File "C:\Users\Josh\PycharmProjects\PyBot\venv\Scripts\musiccommands.py", line 31, in play
    self.player = await self.vc.create_ytdl_player()
TypeError: create_ytdl_player() missing 1 required positional argument: 'url'

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "C:\Users\Josh\PycharmProjects\PyBot\venv\lib\site-packages\discord\ext\commands\bot.py", line 846, in process_commands
    yield from command.invoke(ctx)
  File "C:\Users\Josh\PycharmProjects\PyBot\venv\lib\site-packages\discord\ext\commands\core.py", line 374, in invoke
    yield from injected(*ctx.args, **ctx.kwargs)
  File "C:\Users\Josh\PycharmProjects\PyBot\venv\lib\site-packages\discord\ext\commands\core.py", line 54, in wrapped
    raise CommandInvokeError(e) from e
discord.ext.commands.errors.CommandInvokeError: Command raised an exception: TypeError: create_ytdl_player() missing 1 required positional argument: 'url'
```

Figure 3.10: Error message as a result of not passing a URL in the youtube player

3.4 Issues and what I have learnt

The biggest issue that I faced during my development of the bot was that having one single script with all of your code is incredibly messy and made the development of my project extremely slow. To combat this I made all of my code modular and introduced the cogs module as mentioned before. This allowed me to access the scripts I made modular from the main script so that I could develop each section separately and identify problems much easier.

I also learned that engaging in communities has such a huge benefit when it comes to developing applications and any input is invaluable. I was stuck on using the youtube API for about a week which held me back on developing my other modules by a significant amount. By asking for assistance in the Discord.py developers guild I was able to overcome this trouble by applying the use of classes to reference objects created outside of a specific command. This enhanced my understanding of Python to another level and I feel like I have grown a lot as a developer during this project.

As a result of this, I learned the hard way that sometimes things do not go to plan or how you would like them to. That does not mean all that is lost however, as putting other requirements before others and taking a step back from a problem will potentially broaden your horizons and reveal a solution you may not have thought of at the time of the issue. Being able to adapt to these issues and work around them without hindering my progress will prove to be extremely useful in the future.

3.5 Technical achievement

My technical achievement over the course of this project mainly consists of becoming a capable Python developer in such a short amount of time. From having messy code and hard coded variables due to being new to whitespace coding throwing me off a little bit to having a more dynamic application that depends on config files, classes and modules has shown I have come a long way from the start.

Most Python projects just use the "import" command to use other modules and their variables but this wasn't possible using the Discord.py API wrapper. This meant that I had to use a higher level of asynchronous programming in Python to use multiple modules with classes and cogs so that everything could run concurrently. To be able to do this I needed to study how this new way of using Python worked as it was not something I was initially introduced to during the online lectures and books.

Asynchronous programming in Python is something that is only a few years old so it is a fairly new concept. It's not easy making a project this size run concurrently on a single thread and I'm amazed I actually made it possible using the asyncio module. I expected my program to constantly crash and throw errors but it performs incredibly well.

I also feel like getting used to and implementing multiple APIs into a single project in a short amount of time is also a very big achievement. As they all work differently and have their own wrappers it means I have to adapt to the syntax and functionality of each one and end up writing the League module totally differently to the games or commands modules.

Finally, I feel like tackling the challenge of learning an entirely new language with a different syntax and no compiler put me out of my comfort zone and is an achievement in itself.

3.6 improvements

The most important improvement for my bot would be to have support to run on an unlimited amount of Discord guilds. This would allow every single member of the Discord community to run my bot on their servers. Given the time and hardware limitations of this project I would have never been able to achieve this goal as a PC or laptop alone would not be able to handle the amount of connections being made to it at a single time. This means I would have to set up a dedicated server which would incur too large a cost for a university project.

Another improvement I would make for my bot is to support a bigger number of APIs from other games such as Counter-Strike or Player Unknown's Battlegrounds. These games are huge in the gaming community and would be extremely beneficial to include support for them. It would have been possible to include all of these games in my current project, however I decided it would be best to pick a single one and perfect it rather than have even more APIs to juggle and have them all developed to a mediocre standard.

Finally, some more minor improvements would include being able to queue up YouTube songs for continuous play. As the bot stands currently it likes to crash when another request is made when a song is playing as the two requests clash with eachother. I would also be able to implement more features like the levelling system by creating a complex database. Again, due to time limitations I was not able to accomplish this task.

Chapter 4

Conclusion

4.1 Python and my future

I have had so much fun developing this application using Python that I want to pursue it after graduation and dig even deeper into the world of scripting. Python is the second most language in demand after Java and I feel like after completing my time at University I will be able to step into the wider world of computing and find my place with ease within the Python development community.

4.2 Agile development is the way forward

Using a waterfall schedule will rarely, if ever help with programming projects. Knowing when something will go wrong is impossible to tell and you will be held up on a specific part for a lot longer than expected. Using another methodology such as Scrum will make your project go a lot smoother than using the waterfall methodology. Scrum works by building fixed-length iterations called sprints that give teams a framework for shipping software on a regular cadence [9]. Burn down charts are used to track progress and account for days where nobody in the team may be developing at all.

4.3 Success?

At first I thought that I would get held back using a project with multiple APIs to cater to the needs of a lot of people but in the end I pulled through and delivered on almost every goal I set myself. The only exception to this would be creating the levelling system I included in my interim report. Due to time limitations I wasn't able to create a separate database and have it up and running to a good enough standard so I had to cross that implementation off the list. Thankfully it was one of the smallest things I wanted to implement and did not effect my project at all as it was just a fun feature. Therefore, I deem my project to be a great success and has helped me become a stronger developer as a result.

Appendices

Appendix A

Project Planning

A.1 Gantt Chart

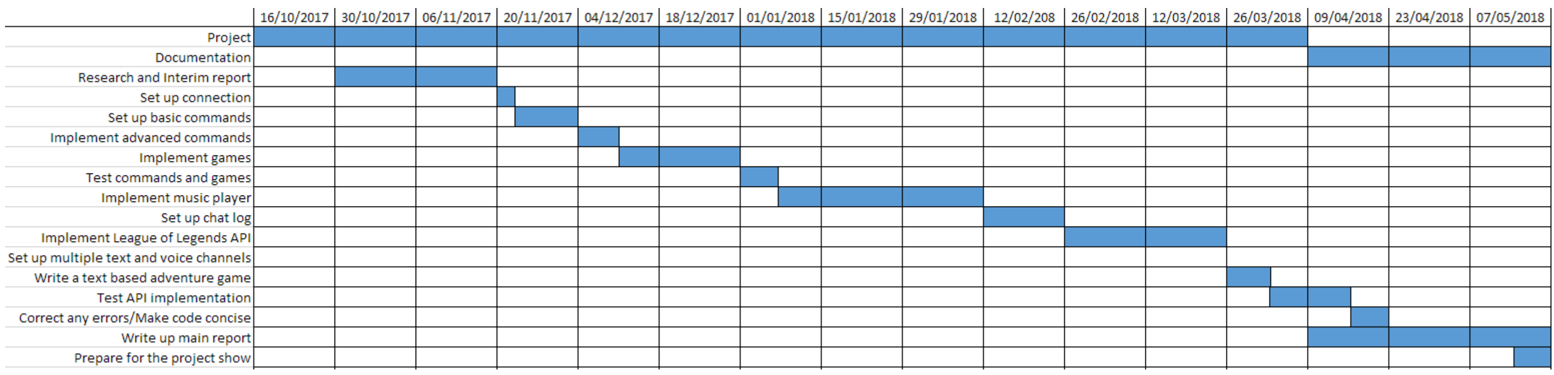
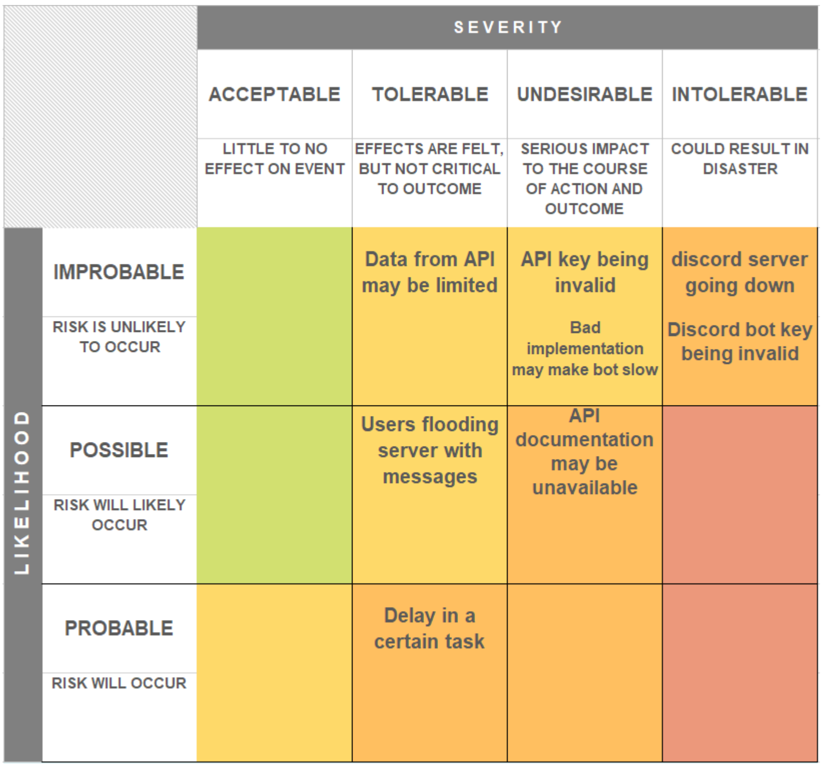


Figure A.1: Revised Gantt chart of the waterfall methodology

A.2 Risk Matrix diagram



LIKELIHOOD	SEVERITY			
	ACCEPTABLE LITTLE TO NO EFFECT ON EVENT	TOLERABLE EFFECTS ARE FELT, BUT NOT CRITICAL TO OUTCOME	UNDESIRABLE SERIOUS IMPACT TO THE COURSE OF ACTION AND OUTCOME	INTOLERABLE COULD RESULT IN DISASTER
IMPROBABLE RISK IS UNLIKELY TO OCCUR		Data from API may be limited	API key being invalid Bad implementation may make bot slow	discord server going down Discord bot key being invalid
POSSIBLE RISK WILL LIKELY OCCUR		Users flooding server with messages	API documentation may be unavailable	
PROBABLE RISK WILL OCCUR		Delay in a certain task		

Figure A.2: Risk matrix derived from my assessment in the interim report

A.3 Monthly Meetings

I tend to get stuck into my work and do large chunks at a time, therefore I decided monthly meetings would be the best course of action to give myself some space to make sure everything works before I met with Jim to discuss further improvements.

Meeting 1 (in Person) (5th October)
- Discussion of goals
- Discussing how to make project a challenge (Python)
- Signing and finalising project proposal
- Write up full proposal
- Get started on planning my project

Meeting 2 (Email) (27th November)
- Discuss Viva
- Talk about any changes

Meeting 3 (Viva) (6th December)
- Discuss my project
- Go over project plan and make sure it is sound
- Act on feedback from the Viva and make any amendments

Meeting 4 (Delay due to christmas break) (2nd February)
- Go over code format and make sure it is tidy
- Turn code into modules, one giant program is not good
- Comment on every function
- Finish bulk of program

Meeting 5 (Email) (12th March)
- Getting a grasp of where I stand compared to my original plan
- Work on any potential additions
- Further refine what I have done so far to ensure it works
- Transfer work to laptop using Virtual Environment

Meeting 6 (in Person) (27th March)
- Discuss all of my completed code
- Convert hard coded variables into its own file (I went with .ini)
- Discuss what to include in my report (Diagrams etc.)
- Discuss any further improvements I can make to wrap up my project

Meeting 7 (Email) (5th April)
- Go over what I have done in my report plan so far
- Check if there are any suggestions that Jim would give to add to my plan
- Get started on the full write up

Meeting 8 (Email) (22nd April)
- Finish write up
- Verify with Jim that it is to a good standard

A.4 Work Diary/Schedule

I created this schedule at the start of my project and have merged it with my diary at the time of writing the report to highlight the key issues I came across on certain weeks of development.

WB 16/10/17 - Week 1

- Write up idea and propose to Jim
- Organise my schedule
- Read the Discord.py documentation
- Get engaged with the Discord.py community and join their guild

WB 23/10/17 - Week 2

- Start exploring Python
- Make a couple small programs like calculators to get used to the syntax

WB 30/10/17 - Week 3

- Watch online lectures on Python
- Read ”Learning Python” by Mark Lutz

WB 6/11/17 - Week 4

- Start interim report
- Get in touch with Jim and John to agree to a Viva date

WB 13/11/17 - Week 5

- Get started on the main connections part of my bot
- Had some trouble setting up the connection to the Discord servers, contacted members on the Discord.py community to assist with this

WB 20/11/17 - Week 6

- Finished the main connection to the Discord server and wrote a simple rock paper scissors game to test the bot to make sure it works.

WB 27/11/17 - Week 7

- Agreed on a date to meet for the Viva (6th December)

WB 4/12/17 - Week 8

- Attend Viva and discuss project plan
- Everything went smoothly, need to go back over Gantt chart
- Stick with waterfall methodology (later realised I could switch to agile as I explained previously)
- Organise with Jim the best times to meet that were convenient to him

WB 11/12/17 - Week 9

- Get started with some basic commands (kick, ban) to get used to the Discord.py API
- Had some trouble understanding how to pull context from the user(ctx)
- After a couple days break I went back and figured it out almost instantly
- Wrote more commands and finished the basic commands section early

WB 18/12/17 - Week 10

- Christmas break so development was slow
- Started on some advanced commands

WB 25/12/17 - Week 11

- Still Christmas break so I took this week off

WB 1/1/18 - Week 12

- Finished the advanced commands and moved on to implementing the rock paper scissors and magic 8 ball games
- Split the original RPS game into its own commands to make bot perform better

WB 8/1/18 - Week 13

- Got started on implementing the music play
- Could only manage to get the music to play and had no control in pausing and resuming
- After a few days of working on this I was still stuck so I moved on to the chat log

WB 15/1/18 - Week 14

- Start on the chat logging by writing to a txt file
- Had an issue where the file would claim to write but it was left blank
- Fixed by creating a new file and changing the directory

WB 22/1/18 - Week 15

- Moved on to working with the League API
- Got stuck trying to work with JSON as it was taking up so much space and was performing very slowly
- Was pointed to an API wrapper that helped immensely
- Started to create some commands with temp API key (runs out after 24 hours)
- Registered for an API key with unlimited use

WB 29/1/18 - Week 16

- API key registration was accepted so the bot can have a full up time now
- Created multiple text and voice channels to test and avoid flooding of messages
- Created more League commands

WB 5/2/18 - Week 17

- Went back to working on the YouTube API
- Discovered the use of Cogs and converted all of my scripts into classes
- Fixed the API and can now pause, resume, stop and get information such as the title

WB 12/2/18 - Week 18

- After getting feedback from Jim I created a .ini file to store all of my variables such as the API keys that could change between multiple users
- Created a text based adventure game - Tested all of my methods to make sure they are working as intended
- Some of my cogs wouldn't load due to incomplete methods so I had to go back and fix them

WB 19/2/18 - Week 19

- Finished testing and everything now works
- Got started on my documentation
- Some commands stopped working again due to conflicts in the scripts

WB 5/3/18 - Week 20

- Went back and fixed all of my code for the final time
- Bulk of code finished
- No need to continue my diary as the rest is just writing the documentation

Appendix B

Diagrams

B.1 UML Diagram of API Wrappers

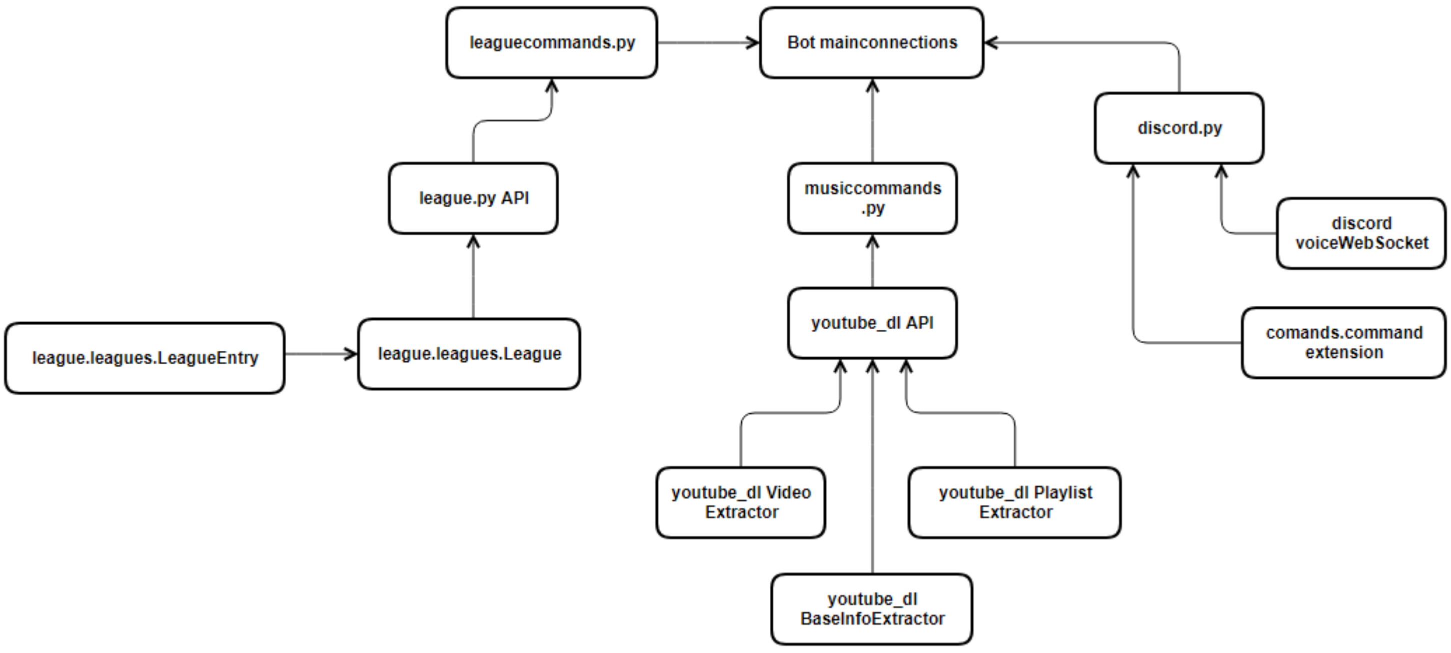

leaguecommands.py
Bot mainconnections
discord.py
league.py API
musiccommands .py
discord voiceWebSocket
comands.command extension
youtube_dl API
league.leagues.LeagueEntry
league.leagues.League
youtube_dl Video Extractor
youtube_dl Playlist Extractor
youtube_dl BaseInfoExtractor

Figure B.1: Each API Wrapper and how it connects to the mainconnections bot

Appendix C

Code

C.1 PyBot

- USB provided
- Code in the folder 'PyBot'

Chapter 5

Biliography

Bibliography

[1] Discord Inc. Discord. `https://discordapp.com/`, (2015 (Initial release)). Accessed: 2017-10-16.

[2] Rapptz. discord.py. `https://github.com/Rapptz/discord.py`, 2017. Accessed: 2017-10-16.

[3] Riot Games. Riot games developer. `https://developer.riotgames.com/`, 2017. Accessed: 2017-12-08.

[4] rg3. youtube-dl. `https://rg3.github.io/youtube-dl/`, 2017. Accessed: 2017-12-08.

[5] Python. asyncio asynchronous i/o, event loop, coroutines and tasks. `https://docs.python.org/3/library/asyncio.html`, 2017. Accessed: 2017-10-16.

[6] aio libs. aiohttp. `https://github.com/aio-libs/aiohttp`, 2017. Accessed: 2017-10-16.

[7] datmellow. League.py. `https://github.com/datmellow/League.py`, 2017. Accessed: 2017-12-08.

[8] Mark Lutz. *Learning Python: Powerful Object-Oriented Programming.* " O'Reilly Media, Inc.", 2013.

[9] Dan Radigan. Scrum: A brief look into using the scrum framework for software development. `https://www.atlassian.com/agile/scrum`, 2017. Accessed: 2017-12-08.

[10] Rapptz. discord.py docs. `http://discordpy.readthedocs.io/en/latest/index.html`, 2017. Accessed: 2017-10-16.

[11] Riot Games. League of legends api docs. `https://developer.riotgames.com/api-methods/`, 2017. Accessed: 2017-12-08.

[12] datmellow. League.py documentation. `http://leaguepy.readthedocs.io/en/latest/index.html`, 2017. Accessed: 2017-12-08.

[13] rg3. youtube-dl documentation. `https://github.com/rg3/youtube-dl/blob/master/README.md#readme`, 2017. Accessed: 2017-12-08.

[14] aio libs. Asynchronous http client/server for asyncio and python. `https://docs.aiohttp.org/en/stable/`, 2017. Accessed: 2017-10-16.