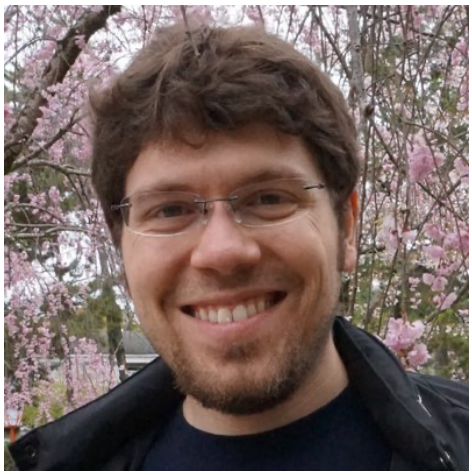
Introduction

Nowadays everyone uses social media and one of the most popular ones is famous “*discord*”. Created and currently ruling by Jason Citron, the platform got its hype since 2015, when it was hopefully released. The main idea of the CEO was to make a cross-platform application which will give an opportunity for people all over the world text each other not only with simple messages, yet also create audio or even video calls, with the targeted auditory of gamers and streamers. The leading reason and motivation for him was that in the childhood, Jason with his friend Stanislav Vishnevskiy, who is current chief technology officer of discord, showed a really strong love for the video games, yet did not have a chance to communicate with his friend while playing. There is no point of explaining the way it works because if you are reading this book, then probably you are already acquainted with the app itself, therefore, let us get to the point. People always wondered if robots and artificial intelligence can replace humanity, while it already started replacing plenty of things in discord which were earlier done by a person. For example: moderation commands, reaction roles, creating and deleting channels, choosing the giveaway winner, and much more. Citron didn’t only give an opportunity to talk and contact other people, he also gave an opportunity to develop and create bots by ourselves and people, they found numerous ways to do it. Consequently, currently, almost every popular programming language has a library, a module to program their own bot, while we will talk only about one, more specifically “python”. You might ask, why python? The answer is simple, because python is one of the easiest and popular languages in the modern days, unlike javascript or c++, leaning which you will spend months or even years. However, unfortunately this book will not help you with the learning language itself, as it contains documentation of the module, made by discord only. On that account, you are kindly asked to learn at least the basics of it by yourself, or any other source you find useful. The book will have divisions and chapters as well. Each function of the library, each module, each class will have its own chapter to make it easier for you and at the end of the book you can find numbers of the pages for needed section. Additionally, all information,



CEO of discord, Jason Citron

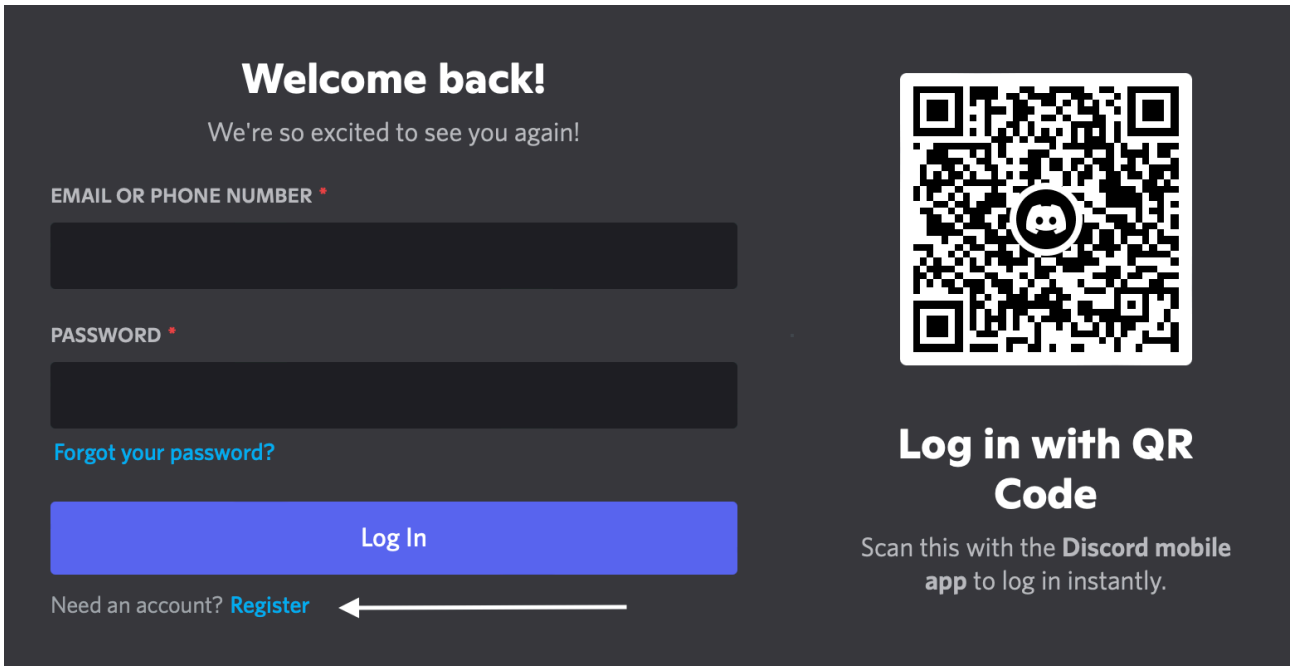
CTO of discord, Stan Vishnevskiy

versions, pictures, links and all other useful data is going to be given in the “useful information” section of the book to alleviate the work.

Chapter 1
Preparation

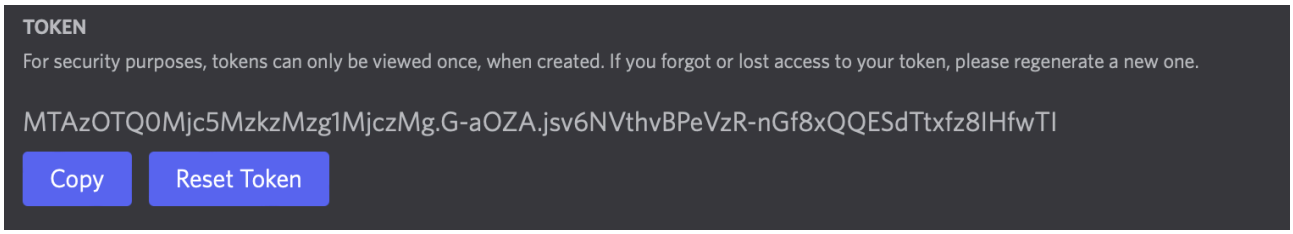
Before starting to learn the coding part, we have to create the object of a bot itself, to customize and add to the testing server for future review. If you are already familiar with discord and the system of creating bots here, you may skip this part. While if you have not registered to the discord or you are new to it, you are warmly asked to do it, by entering the official discord website “https://discord.com/" and pressing the “*login*” button as it is shown below.

Then filling up the blank sections email and password, in case if you already own an account, otherwise press the “*register*” button as it is shown under.

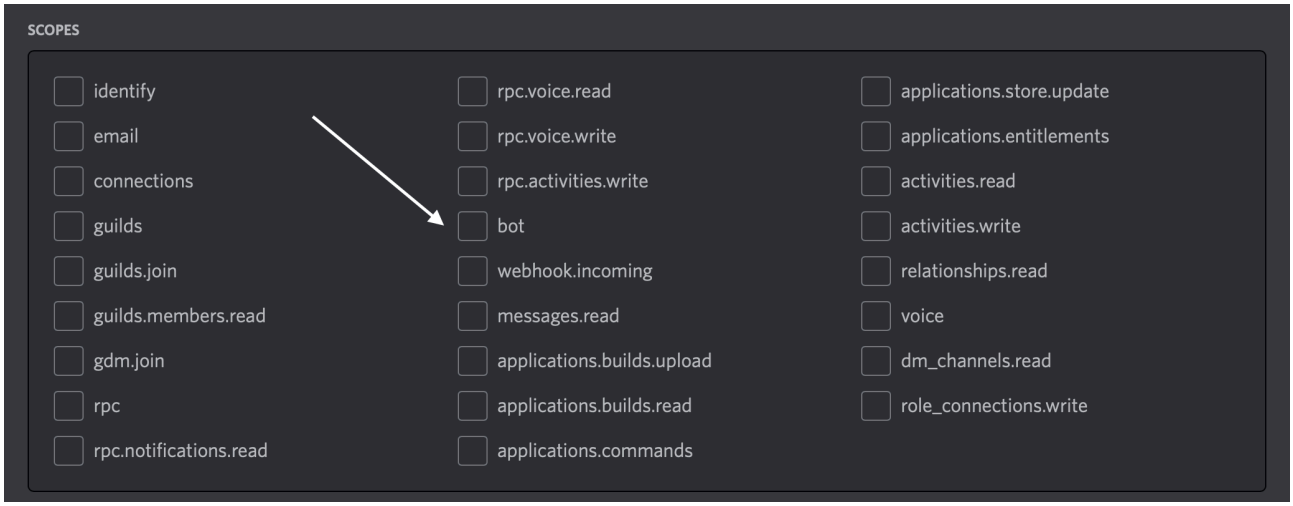


Succeeding the registration you will need to create the official discord application and let discord know that it is going to be a discord bot. First of all, enter the discord portal for the developers by the following link, “https://discord.com/developers/applications". Then, in the application section press the “*New application*” and fill in the blank with the name you want your application to have. After finishing it, one of the most important steps, which was already mentioned higher, is to register a bot and assign it to your application. You have to enter a different section by pressing the “*bot*” button on the left side of the website and only then pressing “*create bot*”.

Following, not less important step is to customize it. You are free to choose your own name, profile picture, and the permissions bot will require, and need for further work, about which we will talk later. For now, it is strongly recommended to choose only "*administration*" permission as it will give everything needed for our course. In the same section, do not forget to save the token and make sure no one has any access to it as it gives full control over the bot. If you do not have the token, press "*reset token*" for it to appear



After hiding the token, there is a need to copy the link for adding the bot as well. In the left panel, select the "*Auth2*" and "Url generator" section, then click in the "*bot*" box in the appeared list, make sure you chose the right option as other variants will not work correctly.



In the wake of it, once more choose the permissions for it to have, for now you can choose only "*administrator*" and by scrolling down you will see a link that in future we will use to add the bot to our server. We have the link, yet the server itself is missing. To create it, just enter the discord, not the developer portal and press the plus which is located in the left part of the screen, choose the name, upload icon if you want and press "*create*". Now, when we have the server, we can add our bot here using the link we copied earlier. Woefully, discord did not invent any other way but to paste the link into your browser search field or send it to the server and click to add the bot. Afterwards, when you finally added it, you will be able to see it in the member list, on the right of the application. Obviously at first, it will be offline as we haven't launched it yet, nevertheless we will do it in the next chapter together with first command.

Chapter 1.1

First Launch

Finally, we came to more engaging part of our program. For the bot's first launch we will need to download a programming language named "*python*" which we are going to use all over the course. You can install it from official website "https://www.python.org/downloads/" or Microsoft Store. Note the version as we will use only 3.11.0 as other versions could contain differences and not interpret your program correctly. Despite it, we also need an IDE (Integrated development environment) where we will edit our code. There are hundreds of them while it is recommended to use either "*Sublime Text*", "*Atom*" or "*PyCharm*" as their functional will make the work much easier, yet you can use any of them. When you have downloaded everything mentioned higher, it is time for the most significant part, installing the module for discord. If your IDE includes a module installer in it, you can do it through it, while others will have to do it through command prompt. Firstly, open the command prompt by pressing "*WIN + R*" if you use windows, or opening the "*Terminal*" if you are a MacOS user. Afterwards, type "*python pip install discord*" and wait until it will be installed. If it does not work for you, instead try replacing "*python*" with "*python3*" and "*discord*" with "*discord.py*". After everything is done, it is time for our first code to launch our bot.

```
#importing the discord module for further usage
import discord
from discord.ext import commands

#assigning bot's object to the variable
bot = commands.Bot(command_prefix = "prefix")

#launching it
bot.run("token")
```

As it is shown in the picture higher, we will have to import the modules and the "*commands*" class from discord for creating the object of the bot. Then we create a variable in which we assign the "*Bot*" method of class we imported earlier, and give one argument to assign the prefix, therefore, instead of "*prefix*" you have to write a letter, symbol or anything which is going to be used before the command to call the function and bot. Besides, you will have to replace "*token*" with the token we saved in *Chapter 1.1* to

connect our code with the discord API (application programming interface) and launch it with the "*run*" method. Last point is just to launch the code and congratulations, you have launched your first discord bot! Subsequently, before creating a command we will create an event. You might have a question, what is event? Well, in discord bots, event is something what happens in given occasions. In our case it is going to be a simple notification when bot gets online. To do it, firstly we need to create a decorator, as it is shown on the following screenshot we call the "*event*" method from our class and create a asynchronous function with "*async def*". Name is going to be "*on_ready*", so that it launches only when bot is getting online. It does not take any parameters at all. And the last, simple "*print*" function is just to print any text you want into console for visualization of our event. Voila! Our event is ready for the work! Which means it is time to create our first command. For the educational intentions, we will create a simple command which is going to reply to the sender with a greeting message. We start with the same decorator, yet we have to replace "*event*" with "*command*" as it is not going to be an event anymore. And name for the command you have to choose yourself. The only argument our command will take is "*ctx*" which stands for "*context*". You actually can add more parameters yet for now there is no need as we are creating a simple command for greeting the sender. "*ctx*" has plenty of attributes, such as "*author*" which returns author of the message, "*channel*" which returns the channel where message was sent, "*content*" which returns message's content, "*message*" which returns the object of the message sent, "*guild*" which returns the server where message was sent. As well as numerous methods, for example "*send*" which sends entered text into the same channel and "*reply*" which replies the send message with entered text. In our case we will only use the first one and greet the author with any text you want. Additionally, "*send*" method accepts 6 parameters. "*file*" or "*files*" to assign one or few files, "*embed*" to create an embedded message, "*delete_after*" for bot to delete sent message in entered amount of time and the last, "*mention_author*" which takes only boolean parameters and determines whether bot is going mention author in his reply or not. Now you are free to launch and test your first command by typing its name with prefix in the beginning. In case if something did not work try checking your code for syntax or technical mistakes and compare it the code shown in screenshots above and that is basically it!

```python
#event for notification
@bot.event
async def on_ready():
    print("Launched")
```

```python
#greeting command
@bot.command()
async def hello(ctx):
    await ctx.send("Hello!")
```

Chapter 1.2

Decorators

Now when you have your bot and you have learned how to create at least simple commands it is time to work with decorators. If you are good at programming you already know what that is, yet for people who are new to it, decorator is a design pattern that allows a user to add new functionality to an existing object without modifying its structure. In our case we are going to create cooldown for the command, making it require admin commands and creating aliases for command itself. Like you can see, we already have used a decorator in our first event and even command. We did that for discord.py to recognize our command and connect it with API and maybe you already understood or not, decorator always starts with a "@". So first thing we are going to create from our list is an alias for our command. What is alias? It is a substitute typing which is also going to call our command. To do that, we have to enter a parameter in our already existing decorator and name it "*aliases*". Notice that our parameter has to be a list, yet if you want you can add even one. As it is shown on the picture, my command contains only two aliases which are "*hi*" and "*hey*", while you can add anything you want. After finishing it you may check it, doing the same what you did before yet replacing name of your command with an alias. There actually many parameters which can be used in this decorator, such as "*description*" to create a description for our command, "*enabled*" which gives an opportunity to enable/disable the command. Next in our list is permissions requirement. For that we will create new decorator. After our first decorator, however before the function itself, create a decorator and name it "*@commands.has_permissions()*" what simply asks for permissions to use the command. In the brackets, we will give a parameter which will determine permissions it asks for. In our course we will use "*administrator*", therefore write there "*administrator=True*", like it is on the figure below.

```python
#greeting command
@bot.command(aliases=["hi", "hey"])
async def hello(ctx):
    await ctx.send("Hello!")
```

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

Despite administrator permissions there are countless amount of them, in general they are "*send_messages*", "*manager_server*", "*manage_roles*", "*manage_channels*", "*manage_messages*" and much more. Well done! Now

only those who have administrator permissions on the server are going to have permissions to use the following command. And the last in our plan is to create a cooldown. For this we also will to create a decorator. Start also with "*commands*" but replace "*has_permissions*" with "*cooldown*". Next, there are 3 parameters to be filled. First is amount of uses which will call the cooldown, second is amount of second cooldown will last for and third is type of cooldown and all are used from "*bucketType*" class. There are 7 types of cooldown. "*default*" which will cooldown the command for everyone on every server, "*guild*" which will only cooldown it on the entered server, "*user*" which will cooldown it only for the user who has sent the command on every server he is, "*channel*" which will cooldown it only in the following channel, "*member*" which is also going to cooldown it for user yet only on the server it was sent, "role" which is going to cooldown it for the everyone who has specific role and "*category*" which will cooldown it in every channel in the category. On the picture with our ready command with all 3 decorators we used today, you can see that only default type was used while you are free to choose and experiment with each of them. Unfortunately bot is not going to notify user with the cooldown in the chat, for now, nevertheless we are going to pass that in future chapters while dealing with the error handler.

```
#greeting command
@bot.command(aliases=["hi", "hey"])
@commands.has_permissions(administrator=True)
@commands.cooldown(1, 30, bucketType.default)
async def hello(ctx):
    await ctx.send("Hello!")
```

Chapter 1.3
Moderation

We have already passed how to require administrator permissions for our command and now we will pass how to use them. Moderation commands are generally used by server moderation for punishing, controlling the users and chats on the server. For example banning from the server also considers as a moderation command. Right now we are going to examine only three of them "*ban*", "*kick*" and "*clear*". With the first two it is already clear, yet you might wonder what is the for? "*Clear*" or in other words "*purge*" command is going to delete entered amount of messages in the chat. It is often used by moderators to delete off topic messages, offensive words, generally what broke the rules of their server. First step is to create an empty command. We already passed that, so there is no point of showing it again. Then, add some permissions requirements, for "*ban*" and "*kick*" commands it is going to be as it was, the administrator permissions however for the last one we will use "*manage_messages*" as some moderators might not have the administrator role. This time you can choose aliases by yourself, nonetheless this time we are going to use 2 parameters. Familiar to us "*ctx*" for context and "*amount*".

Second one is going to determine amount of messages bot needs to delete and will be entered by a moderator who uses the command. It is going to accept only integer value so we have to give it a type, just put the semi-colon and data type you want to set. Additionally, we have to give a value, for future checking if user entered the amount or no, like it is on the picture.

```
@bot.command(aliases=['purge', 'delete'])
@commands.has_permissions(manage_messages=True)
async def clean(ctx, amount:int=None):
```

After creating the function we will create a if-statement to check whether the amount was entered. If everything passes successfully we will purge entered amount by using the "*purge*" method of context, otherwise we will inform the user with a message. Simple checking if amount has any value or no will not be enough, we will also check whether it is more than 0 or not, as some people might enter negative values. The method we are using also requires one parameter which in our case is amount of messages, therefore we will simply put the variable there. In the else part, we will send a message to the user where bot will ask to enter a parameter. On the picture there is no text, yet you can add anything you want. That's it, now we will slowly move to the ban command. For it, we will start the same, withal, we will change the aliases, the function name, permissions and parameters. We already talked about permissions so we will skip this part. Name and aliases are also your decision, you can name it anything you want, while the parameter part will be more specific. This time we will add 3 parameters, as always the "*ctx*" and two new, which are "*user*" and "*reason*". We have to ask for user for bot to know which user he is going to ban and the reason for formality. As the reason is not optional we will not require user to enter it. This time, the second parameter will take a unique type, it is going to be "*discord.Member*" which is an object of user and goes either by an id or mentioning them. Repeat the same steps as we did earlier, but instead of "*amount > 0*" type just "*user*".

```
if amount > 0:
    await ctx.purge(amount)

else:
    await ctx.send("error")
```

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

    if user:
        await user.ban(reason)
    else:
        await ctx.send("error")
```

Like you can see above, we only check if user was entered as it is the only mandatory parameter here. Also, we replaced the "*ctx.purge*" with "*user.ban*" to ban the entered user and entered a parameter which is going to be our reason. In case if there is no reason, it will be "*None*". Done! Our command is also ready. For the kick command do literally the same with replacing the "*ban*" with "*kick*", changing the name and that's it. You of course have to customize it for yourself, the text, name and aliases, while here you can see just an example. It is also highly recommended to send a message after banning the user to notify the moderator about the success of the command. Despite the "*discord.Member*", there are plenty of types in the discord API. Starting from "*discord.Channel*", "*discord.Guild*", "*discord.Role*" and ending with "*discord.Message*", "*discord.Reaction*" that you also can use as a type for the parameter in your command. And even give a preassigned value by writing it instead of "*None*".

Chapter 1.4
Member Class

Discord gives an opportunity to get user's info by simple having their ID. Now we are going to pass all attributes and methods as well as uses of user class. First of all, how can we extract the user object? How can we get it? It is pretty easy. There are multiple ways to do it such as getting the ID, name + tag/discriminator or mention. Our moderation commands contains only two methods, which are "*ban*" and "*kick*" while now we will look for all of them. To make the work easier, below you can see a table with all attributes.

banner	returns user's profile banner
name	returns user's name without discriminator
id	returns user's id
discriminator	returns user's discriminator (tag)
avatar	returns user's avatar url
mutual_guilds	returns list of mutual guilds with the user
accent_color	returns user's profile color
status	returns user's status
roles	returns list of user's roles on the guild
mention	returns user's mention in format of <@id>
top_role	returns user's top role on the guild
guild_permissions	returns list of user's permissions

That is basically it, yet also member class contains few methods which we are also going to cover as they are really useful for our course. Parameters which are acceptable for the methods are also going to be mentioned below

ban (reason)	bans the entered user on the guild
kick (reason)	kicks the entered user from the guild
unban (reasaon)	opposite of the ban
edit (name, mute, roles, voice, reason)	edits user's parameters
move_to (channel)	moves user to the entered voice channel
timeout (time, reason)	cuts user's permissions to text on the guild
add_roles (list of roles)	adds entered roles to the user
remove_roles (list of roles)	removes entered roles from the user

Done! These are all methods which user class contains, this chapter will not contain any practice with them, however you are free to test them whenever you want and add them to your code.

Chapter 1.5
Embedded messages

Discord bots are not only the technical part. Design here also matters a lot. Therefore, this chapter will teach you how to create embedded messages, include images, change colors, add fields, descriptions and titles. First of all, what is embed? An embedded message is another component of discord messages that can be used to present data with special formatting and structure. It can contain images, author information, links or any other media. To send an embedded message we firstly need to create an object of embed and assign it to the variable. To do it, simply create a variable and name it whatever you want, in our case it will be "*embed*". Then assign the embed class to it by typing "*discord.Embed()*". It will take 3 parameters. "*title*", "*description*" and "*color*". First one is obviously for the title of the message, notice that it is not allowed to use mentions there as they will not work anyways. Second one is also understandable, yet the third one is a little bit specific. You have to enter either a color code or choose a color amongst discord's offers. To use them, call the "*discord.Color*" or "*discord.Colour*" class and use the method of color you want. Generally available ones are red, grey, blue, green and

```
embed = discord.Embed(
    title = "Title",
    description = "Description",
    color = discord.Color.gold())
```

gold. To use your own color, instead of calling exact color method, try calling "*from_rgb*" method and entering the color code as a parameter. Next, we also can add fields to the embed, amount of which you can choose yourself and create up to 25 different fields. For this, you will have to call "*add_field*" method after assigning the embed. It also takes 3 parameters. "*name*" for the name, "*value*" for the value of the field and "*inline*" which is a boolean requiring parameters determines whether the field will take the full line or not.

```
embed.add_field(name = "name", value = "value", inline=True)
```

Like you can see above, we called the variable which we created earlier. One more point is how to add an image to the embedded message. We will also need to call a new method called "set_*image*". Unlike other method this one requires only one parameter which is image's url. It can be any format including "*.gif*". Parameter's name is obviously "*url*". Moreover, you can add not only a huge picture but also a small thumbnail. Simply repeat the same with replacing "*set_image*" with the "*set_thumbnail*" method here. Although on the picture it is written "*or*", it is totally okay to use both of them simultaneously. To add an author, you will have to use the "*set_author*" method. It also takes 3 parameters such as "*name*", "*url*" and "*icon_url*" which are going to contain authors name, url and url for the icon, the same with "*set_footer*". The only difference is that it takes one less parameter. Only "*text*" for the text of the footer and "*icon_url*" for the image. Besides, there are two more methods. They both remove entered part of the embed. One for the author and second for the field. They are "*remove_author*" and "*remove_field*". Finally the last point is how to send the embed along with the message. While using the "*send*" method, add the "*embed*" attribute and give the value of our variable, like it shown below.

```
embed.set_image(url="url")
#or
embed.set_thumbnail(url="url")
```

```
embed = discord.Embed(
    title = "Title",
    description = "Description",
    color = discord.Color.gold())

embed.add_field(name = "name", value = "value", inline=True)
embed.set_image(url="url")
embed.set_thumbnail(url="url")
embed.set_author(name="name", url="url", icon_url="url")
embed.set_footer(text="text", icon_url="url")

await ctx.send(embed=embed)
```

In the code above, there is no message content except for embed while you can add anything you want, including files by adding a parameter "*file*" and giving the value of picture or video you want to send, or simple text by using the parameter "*content*".

Chapter 1.6
Error Handler

We have discuss commands enough, it is time to cover events as well. As you already know, event happens when specific action is being executed. In the beginning we have used the "*on_ready*" event, and in this chapter you will get familiar with other ones and have some practice. Let us start with the member category and the "*on_member_join*" event, which executes its job when a user joins server. Note that bot has to be on that server for the event to be fulfilled. The literal opposite is the "*on_member_remove*" which happens when a user leaves the server. They both take 2 parameters, one stands for the user which entered/left the guild and the second for the guild user left/entered. It can be useful when you are creating greeting messages or vice-versa, to create a farewell message. Similar to them are "*on_member_ban*" and "*on_member_unban*" which are almost the same, the only difference is that they apply when a user gets banned and for the second one unbanned. "*on_member_update*" turns on in case if a user changed their nickname, avatar, or anything related to the server and themself. Next in the list is the message category. The most common event from this category is the "*on_message*" which simply works when a message was sent. People often use "*on_message_edit*" for the audit-log system as it completes the code when a message is being edited by the author, as well as the next event, "*on_message_delete*" yet this one works when the message is being deleted by anyone. They all take the same parameters which are the message and guild. One more category is reactions. Some events from there are basically "*on_reaction_add*", "*on_reaction_remove*" and "*on_reaction_clear*". Judging by the name it is already clear what they stand for and again take same parameters, reaction emoji and guild. With the roles it is also clear, the simplest and most popular ones are following, "*on_guild_role_create*", "*on_guild_role_remove*", "*on_guild_role_update*", as always they also take the same parameters for the guild and role object. Finally the last event that we are going to cover is the one that handles all errors. There is no specific name for the category as it contains only one event, "*on_command_error*". It responds when our code gives an error. It might be a missing parameter, wrong data type or anything connected with the discord part of our program. There are going to be few general types of error that we will discuss however in reality there are much more. They will also be mentioned yet not examined in details. As it is going to be called

without user intervention, eventually we have to create the event itself first. Simply create an event and necessarily call it "*on_command_error*" as we cannot choose the name by ourselves. Add two parameters, familiar to us "*ctx*" and a new one "*error*". You already know what first one is for, while the second one is to identify which error has occurred. Then we will have to create an if-statement and check which error with the "*isInstance*" method. It also requires two parameters as well. First one is our error variable and the second one is type of error. There are several types and all of them are attributes of the "*commands*" class. Following list will contain general types of the errors. "*CommandNotFound*", "*MemberNotFound*", "*BotMissingPermissions*", "*CommandOnCooldown*", "*DisabledCommand*", "*MissingPermissions*", "*UserInputError*" and "*NoPrivateMessage*". There is no point in explaining each of them as their function is understandable from their name. You can create any action which bot is going to do and even include embedded error. Picture below contains ready for work event with all general types of errors.

```
if isinstance(error, #error):
    await ctx.send("error")
```

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

    if isinstance(error, commands.CommandNotFound):
        await ctx.send("error")

    if isinstance(error, commands.BotMissingPermissions):
        await ctx.send("error")

    if isinstance(error, commands.DisabledCommand):
        await ctx.send("error")

    if isinstance(error, commands.CommandOnCooldown):
        await ctx.send("error")

    if isinstance(error, commands.MissingPermissions):
        await ctx.send("error")

    if isinstance(error, commands.UserInputError):
        await ctx.send("error")

    if isinstance(error, commands.NoPrivateMessage):
        await ctx.send("error")
```

You also might add an else division where you will not specify the error yet at least inform the user that something happened and command can't be done.

It is very useful for the developers to see the error. To send the error itself to the chat just include the "*error*" variable to the text or send it individually. Now whenever your bot will receive an error, does not matter if it is a command used on cooldown, user entering a wrong id or anything, it will notify the sender in the chat with the text you have written. Older versions of the library don't support some types of errors, therefore make sure everything was updated to the newest version.

Chapter 1.7
Status Position

After creating an error handler this will seem a really easy topic as it does not require any effort to do. Before starting to code, let us figure out what do we mean by "*status position*". It is not only the online/offline/idle/do-not-disturb status. Status also includes the activity which can be seen from the profile. It can be a game, music or a live stream. In our case we will examine all of them and even edit and write our own text instead of writing the name of our activity. For that, we will add a new line in our "*on_ready*" event. It is going to be an asynchronous calling of the "*change_presence*" method from the "*bot*" class. First parameter that is going to take will obviously be the status type. Name of parameter is "*status*" and it accepts only attributes of discord's sub-class "*discord.Status*". There are 5 attributes which are "*idle*", "*online*", "*dnd*", "*offline*" and the last, "*invisible*". They're all also available for a user to use. However, the second one is more specific and cannot be used by a simple user, unless they are really playing a game, listening to a music or running a live stream. To add that into our program you simple add a second parameter, name of which is "*activity*". Unlike the previous one it directly accepts only methods of discord class, such as "playing", "*streaming*", "*listening*", "*watching*", "*custom*" or a simple "*game*". And you can notice on the picture that you can add any message you want it to show and bot will interpret it to the activity you want. For example code on the picture is going to show "*playing in some text*". There is also a trend that creates changing activities. You have to create an asynchronous loop by using the "*asyncio*" library and which is going to call the "*chance_presence*" and each time have a different text. There is no point in doing that here as this course teaches you function of a discord API for python, not asyncio. However, you can easily find thousands of free tutorials and lessons for that on any platform. Do not forget that changing the status too often can create a delay for the bot from discord's side as it will can be counted as spamming

```
status = discord.Status.online,
```

```
activity = discord.Game('some text')
```

Chapter 1.8

Channel Types

Discord generally has 2 types of channels, text channels and voice channel. Logically, the text channel is supposed to deliver text messages of the users and the voice channel was created for the calls or streams. If we dive into the details, text channel can be divided into two parts which are the announcement channel and simple text channel. The tricky part of it is that for simple users it is 2 types yet the library accepts the categories also as channels. Therefore, we will include it in our program. For now we will only examine how to create, delete, edit and the classes of the channels themself. Let us start from the methods and attributes of the "*TextChannel*" class. First two attributes are connected to the categories as they show the category name and id in which the channel is. They are basically the "*category*" and "*category_id*". There is also one more attribute related to the id and it's called "*id*" which obviously returns the id of channel. Some basic ones like "*name*" or "*mention*" also exist and return the name and mention in the format of "<@*id*>",of the channel. Methods like "*clone*" will trivially clone the channel, the "*delete*" will delete it. Notice that all actions are permanent and there is now way to return them back. And the last three for now are "*move*", "*edit*" and "*send*". If the third one was already studied before, the first one is a little bit more confused. It takes 6 parameters. First one represents whether the channel will be moved to the beginning of the category, second one to the end, third the number of the channel that should be before our channel, fourth the number of channel that should be after our channel, fifth is amount of channels to skip and the last is category which determines in which category the movement will be. The example is below.

```
channel = #your text channel
await channel.clone()
await channel.delete()
```

```
await channel.move(True, False, 0, 2, 0, category)
```

The channel will be moved to the beginning and be the second from the end in the category which we mentioned in the sixth parameter. We have already discussed all attributes and methods, however forgot the main part. How to create the channel? Easy. We are going to use the "*create_text_channel*" method of the category or guild class. And the only parameter it takes is the name. As you can see in the picture it is really simple and quick to do.

```
await guild/category.create_text_channel(name)
```

The creation of voice channel is literally the same with cutting the "*text*" part and adding the "*voice*". For the category it is applicable as well yet it is not going to work if you try to call it from category class, only from the guild. The attributes for the category have small differences that is why you will see the usage of all of them on the screenshot with the commentary of explanation.

```
category.id #its id
category.guild #guild where it is located
category.mention #mentioning of the category
category.name #returnal of its name
category.channels #list of its channels
category.text_channels #list of its text channels
category.voice_channels #list of its voice channels


category.delete() #deletes it
category.edit() #edits the parameters of it
category.move() #moves its position
category.invites() #invites to the category
category.create_text_channel() #creates a text channel
category.create_voice_channel() #creates a voice channel
```

This is generally everything about channel part of the discord as the announcement channel is included into text channel and does not have any differences from the previous one. To get the object of the channel you may use the method of the bot class which is called "*fetch_channel*" and takes only one, mandatory parameter and it is the id of the channel you are looking for. Also the method "*get_channel*" may be used yet unlike previous method, this has to be called from the guild class.