The official 0x102 docs

* By 0x32, the creator of 0x102

Table of Contents

[Code conventions and guidelines 2](#_Toc115601237)

[Intro 3](#_Toc115601238)

[Styling conventions 3](#_Toc115601239)

[spacing 3](#_Toc115601240)

[Brackets 3](#_Toc115601241)

[type hinting & anotations 3](#_Toc115601242)

[The typing module 3](#_Toc115601243)

[when to use type annotations 3](#_Toc115601244)

[using syntax 3](#_Toc115601245)

[Intro 3](#_Toc115601246)

[try and except 4](#_Toc115601247)

[match case 4](#_Toc115601248)

[list comprehensions 4](#_Toc115601249)

[async and await 4](#_Toc115601250)

[Performance and efficiancy 4](#_Toc115601251)

[Intro 4](#_Toc115601252)

[speed 4](#_Toc115601253)

[Under the hood 5](#_Toc115601254)

[Intro 5](#_Toc115601255)

[main.py 5](#_Toc115601256)

[Intro 5](#_Toc115601257)

[dependancies 5](#_Toc115601258)

[What is done 5](#_Toc115601259)

[debug.py 5](#_Toc115601260)

[Intro 5](#_Toc115601261)

[dependancies 5](#_Toc115601262)

[What is done 6](#_Toc115601263)

[cache.py 6](#_Toc115601264)

[Intro 6](#_Toc115601265)

[.gitignore 6](#_Toc115601266)

[intro 6](#_Toc115601267)

[What has been excluded 6](#_Toc115601268)

[The assets folder 7](#_Toc115601269)

[Intro 7](#_Toc115601270)

[Uses 7](#_Toc115601271)

[assets/deathMessages.json 7](#_Toc115601272)

[Intro 7](#_Toc115601273)

[what is done 7](#_Toc115601274)

[assets/debug.json 8](#_Toc115601275)

[Intro 8](#_Toc115601276)

[use and syntax 8](#_Toc115601277)

[assets/rickrollinks.json 8](#_Toc115601278)

[intro 8](#_Toc115601279)

[How do I expand on this 8](#_Toc115601280)

[assets/truthDare.json 8](#_Toc115601281)

[Intro 8](#_Toc115601282)

[How do I expand this 8](#_Toc115601283)

[cogs/ui/\_botSync.py 8](#_Toc115601284)

[Intro 8](#_Toc115601285)

[cogs/ui/\_encryption.py 9](#_Toc115601286)

[intro 9](#_Toc115601287)

[What is done 9](#_Toc115601288)

[cogs/ui\_encryption.py 9](#_Toc115601289)

[intro 9](#_Toc115601290)

[What is done 9](#_Toc115601291)

[cogs/ui/\_triviaView.py 10](#_Toc115601292)

[intro 10](#_Toc115601293)

[What is Done 10](#_Toc115601294)

[TriviaView class 10](#_Toc115601295)

[cogs/ui/\_truthOrDare.py 11](#_Toc115601296)

[intro 11](#_Toc115601297)

[TruthButton 11](#_Toc115601298)

[Dare button 11](#_Toc115601299)

[TrutghOrDareUi 11](#_Toc115601300)

# Code conventions and guidelines

## Intro

The 0x102 bot’s code follows the pep8 styling guide, this is achieved with the black formatter.

## Styling conventions

### spacing

In the bot’s source I have used spacing when:

* There are values separated by commas, this applies to lists, tuples, ech…
* Values separated by an operator (e.g > or <=), this is most often used in if statements

### Brackets

There are people who put parenthesis around if statements, however this is not necessary to encompass the entire if statement. Brackets should only be used in this circumstance if the walrus operator is being used, =:.

## type hinting & anotations

Type-hinting is a method that allows python developers to know what type a variable is, this can help to eliminate bugs. For example, we might have a variable “user\_id” and we would annotate that with type int, this use case could eliminate a bug where we have a database and we would need to return a string, we could type annotate that so that we would remember to convert to an int when setting it to “user\_id”, this would keep old code from working and is one example of why type hinting and annotations are used in the codebase.

### The typing module

Python has a build in “typing” module, this module was used when some type-hints could cause errors. An example of this would be, “list[int]”, having this code in old modules would cause an error, hence why the typing module came to be. The aspect of the typing module where “List[Int]” would have been used is now obsolete because the once broken syntax now works. The typing module is not used for class annotation because I don’t know how to use generics yet.

### when to use type annotations

When to use type annotations:

* When defining parameters
* When defining the return type of a function
* When defining a variable

This does cover most use cases for type hinting, however type hinting is not used to define empty variables, for example: a: int is still valid syntax although it is not assigned to anything.

## using syntax

### Intro

Depending on what you have used python for, you might have different ways of using syntax, such as using nested try and except blocks, or using else statements everywhere in your code. However, the guidelines listed below are to try and improve code readability and minimise bugs.

### try and except

Some people like to do a lot of error handling in their code and use try and except blocks when opening a file. However, wrapping an entire command in try and except is not welcome, especially when the command is extremely long. This could cause bugs and is again why wrapping a try end except block is not wanted. I rather use many if statement to test against possible errors that may lead to bugs. However, for some cases that cannot be tested in advanced or where many if statements are unproductive, then try except statements are allowed.

### match case

In python 3.10 a new keyword has been introduced called the match statement, the syntax is like a switch case statement but more pythonic. However, as pointed out by mCoding, the python changelog and docs, the purpose of the statement is for structural pattern matching and not as a switch case. Therefor the match case is going to remain for its intended purpose and not for faster if statements.

### list comprehensions

An older feature of python is also list comprehensions or shorter for loops. List comprehensions are used in the bot’s source and are a cleaner way to create any list-like item without needing to convert types or append objects. Some python programmers do say that the syntax is un-pythonic and less readable. However, if you do accidentally not use or refuse to use list comprehensions in the codebase then do note that they may be switched out of replaced.

### async and await

Many people who use discord.py or any other alternative are aware that the codebase is async await. (For fairly obvious reasons I will not be explaining what async await it) This also means that the bot is also async await. This does mean that the libraries that are used should support async await. This means that instead of using the requests library, we use the newer httpx library.

## Performance and efficiancy

### Intro

As many programmers know speed, maintainability and expandability are very important for any project. In the bot’s code I have made sure to keep the bot as fast as possible. There are also close to no restrictions on how expandible the bot is, although ways on making more generic commands are going to be worked on.

### speed

#### expanding python with c and/or c++

Cython is an extremely popular way to speed up python code, this would be encouraged if the source and ways to compile to code to multiple platforms. This means you include cython source files and the setup.py file that you have used to build the extension. Regarding C and C++, they should be linked with a cython file so that they can be imported with little hassle. C and C++ should only be included for extremely computationally intensive tasks that just take too long with python or are not practical with python.

#### code clendliness

Code should be refactored or tested before creating a pull request. Or in other words, before a pull request, the code should be tested at the minimum. If the code is in such a state where it is almost unreadable or slow, then a pull request may lead to a requested change.

# Under the hood

## Intro

This section is going to cover how the bot functions, this will also include how the commands work and the rolls of each file, for the most part.

## main.py

### Intro

This file is in the root directory of the bot. The function of this file is to initialize the bot and load all the cogs.

### dependancies

This file requires the vars.ini and the constants.ini file to access data. These files contain the bot’s token and the bot’s testing guild id.

### What is done

In this file we start by importing all necessary libraries. We then initialize the bot with everything that we need, this includes the intents, command\_prefix and the application\_id. The intents is required so that discord’s api knows what we want the bot to do, the api now know that we want the bot to read messages, get information about discord users and more message stuff.

We then set two attributes to the bot, we give the bot a httpx.ClientSession attribute, and a console attribute. This is so that we can access a httpx client from anywhere in the code. This means that we don’t need to reinitialize a client session every time we need to do a https request. The console attribute is so that we can print debug messages. The console attribute is also explained later.

Next we create an on\_ready function this is called when the bot is online, and loads all the cogs. The function also calls the create\_help\_command function. This is used to create a more generic help command; this then outputs a mark down file that is remade when the bot is run.

We finally call the bot.run method which will run the bot, this will call the on\_ready function loading all the cogs and creating a generic help command.

## debug.py

### Intro

Previously I said that I would cover this file and it’s uses. This file has one purpose which is to make debugging easier.

### dependancies

This file requires the assets/debug.json, this contains all the debug messages that the bot needs and is also expandable.

### What is done

This file has one class that is used and one class for setup.

First, we create a meta class this class is the meta class for the Debugger class. The role of the DebugMeta is to read the assets/debug.json and add special print functions and to set the console attribute. This is run whenever we create a Debugger class.

#### How to use the debugger class

The special print statements are dictated by the debug json file and follow this format.

{  
 "name": "what you want to print",  
 “nother\_name”: “more stuff to print”  
}

The meta class would read this and give the Debugger class two extra attributes:

* print\_name
* print\_another\_name

The print\_name would print “what you want to print” to the console. This may seem useless at first, but arguments and any rich text features can be used such as [underline] and [/underline] to underline text and much more. Arguments can also be passed with the syntax below:

{  
 “arguments”: “{arg0} for the 0th argument and {arg1} for the 2nd argument”  
}

This class is set with the bot so you could do this:

bot.console.print\_arguments(“a”, “b”)

## cache.py

### Intro

The most of us have heard of caching databases such as Redis. I have done something similar which is to create a cache, the cache is just a dictionary that is stored in memory. I have only used the cache to store the guild\_id of the testing server.

I am deprecating this so there is no need for further documentation.

## .gitignore

### intro

The .gitignore file is used for when excluding files from a repository. Because this project is open source and there are files that are important for my bot to work, I have hidden many file types.

### What has been excluded

|  |  |
| --- | --- |
| File extension or name | Reason |
| \_\_pycache\_\_ | This is a folder that is created whenever importing a python file. These folders include binaries that are excluded. |
| .idea | This is also a folder that contains data for the pycharm IDE, in terms of run requirements, this folder is not needed and is therefore excluded. |
| venv or virtual environments | These can be used by developers to install libraries locally in a directory, these folders can become really big verry quickly and does not need to be included in the bot’s source. |
| \*.db | This and .sqlite3 are common file extensions for sqlite databases and because these databases are binary they also don’t need to be included. |
| \*.pyc | These are the binaries used by python as mentioned in the \_\_pycache\_\_ explanation, I have decided to explicitly ignore this file extension. |
| \*.pyo | These files are the python2 equivalent of .pyc files and I have decided to ignore these too. |
| .vscode | Similar to the .idea folder this folder has metadata for vscode, this does also not need to be included. |
| \*.o | Object files are commonly used while linking or building any compiled language. This was used when I was linking C++/C to python. |
| vars.ini | This file has api keys in it. |
| \*.save | I found this in a list, and I don’t know what it is for. |
| \*.swp | This is used when opening a file with vim, every once in a while, one of these files can be committed to a repo, and because these files also have binary data in them, I have excluded them. |

## The assets folder

### Intro

This folder holds database like files, in my case these files are json files that hold large lists or dictionaries. This data can be accessed by python when using the json library.

### Uses

I have placed death messages, debug, truth or dare and rick roll links in the assets folder each having their own name respective to their use.

## assets/deathMessages.json

### Intro

In my bot I have a feature that displays a Minecraft like death message, these messages would be too long and confusing if they were in the source for the bot. So, I have placed the messages in a file that has all the messages, the file is expandable with an unlimited amount of possible death messages.

### what is done

The file contains a dictionary with to lists. One list has the key “self” and the other key is “other”. These keys are used when a play or user kills another player/user. The self key is used when a user uses /kill without giving the name of another person, this will then select the “self” key and then picks a random death message, the {player} part of the string is then replaced by user’s name and then sent.  
When /kill @you is used then the “other” list is accessed and then a random death message is chosen and the {player} and {killer} values are then substituted for their respective values.

## assets/debug.json

### Intro

I have also mentioned the debug.json file in one of the dependencies. This file is however used to define the debug messages. This can save repeating the same message repeatedly. The syntax and how to use is in the next section.

### use and syntax

The debug.json file is only a dictionary that contains strings as keys and strings as values. The key to the dictionary is concatenated with “print\_” this means that if there is a key that is called message, there would be a function called “print\_message”, this eliminates an argument called key in the console.print, a bad implementation for example would require:  
print(“hello”, type=”print\_message”)

This syntax is longer than it should be and is also quite frustrating to work with. However, Metta classes don’t offer good type hinting, but this can be solved by having the base Debugger class inherit from a class with all attributes left blank (because they won’t need to be used) and over written.

## assets/rickrollinks.json

### intro

Another aspect of my bot is being able to rick roll people, this is done by having a list of dictionaries, with each dictionary containing a “url” and an “excuse”, the url is the url to the rick roll and the excuse is to try and make the link sound less like a rick roll.

### How do I expand on this

You can add another rick roll link by appending a new dictionary to the list, this dictionary should have the appropriate excuse of type str and a url also with type str.

## assets/truthDare.json

### Intro

This file is used when running the /truthordate command. This file is a dictionary with two keys, a “truth” key and a “false” key, these keys do their own respective thing and are a list of truth or dare questions.

### How do I expand this

Categorize your sentence/question as truth or date, then put it as a string inside of the section that would make the most sense as a string. That is all.

## cogs/ui/\_botSync.py

### Intro

I am currently collaborating with arty studios to make discord bots, the bot sync is a user interface that is used to change settings, for the moment the \_botSync.py file is not being used.

## cogs/ui/\_encryption.py

### intro

My bot is also able to crack Caesar Ciphers and create them. This module creates the encryption methods but has the “EncryptionView” imported instead of the individual functions. The view is a ui element with a single dropdown attached to it. This dropdown shows all forms of encryption that the bot supports and encrypts the given message. The message is then sent back to the user in an ephemeral message.

### What is done

We start by creating a class called EncryptionDropdown which has the following attributes: \_\_init\_\_, callback, create\_ceaser\_cipher, create\_hex\_code, create\_base2\_encode and create\_base\_32\_encode.

#### The \_\_init\_\_ function

This function is called when the class is initialized these calls the Select class’s \_\_init\_\_ method, last the message is set as an instance attribute and the characters are also set as an instance attribute. The message is so that we can access the original message from anywhere in the class and the alphabet is important for the Caesar cipher.

#### callback

This function is invoked when the user has chosen what to select. This function sends an encoded message respective for what was selected. This then sends an ephemeral embed and the encoded message.

#### create\_ceaser\_cipher

This function takes no arguments and returns an encoded string that has been encoded with a Caesar cipher.

#### create\_hex\_encode

This function is like the create\_creaser\_cipher function however this function returns a string that has been encoded by a hex code, this is still reversable though.

#### create\_base2\_encode

This is also like the hex encode but the base is different.

#### create\_base32\_encode

This is also like the hex encode but returns a base 32 version of hex encode.

## cogs/ui\_encryption.py

### intro

Many bots are dedicated to a ticketing system of some sort, but I have added my own ticketing system to my bot.

### What is done

#### on\_submit

When the modal is submitted then this function is called, this function looks at a configuration database and then selects all rows where the guild\_id column is the same as the interaction’s guild id. It then selects the moderation channel ids and then sends recommendation sent by the user to the modal. So, the modal prompts the user for a description of their complaint and then the description is sent to one of the mod channels.

#### The TicketingModalView class

This class inherits from the View class, the view class must be inherited from when trying to make a ui component, so we initialize an empty ui component and then we add the TicketingModal, that is it.

## cogs/ui/\_triviaView.py

### intro

The bot has also got a /trivia command, this sends a trivia dropdown where the question is the placeholder and the user can select their answer from a dropdown, they then get an embed telling them that they are correct or an embed that tells them what the answer should be.

### What is Done

#### \_\_init\_\_

We start by initializing the Select’s \_\_init\_\_ function, we then pass the maximum selectable amounts of fields (1) and the minimum number of selectable fields (also 1), we then pass the possible answers as a SelectOption. We also need to shuffle the answers because the first answer would be the answer if we didn’t, for this we can’t use the random.shuffel function because the function just mutates the array instead of returning it, so I have created the shuffle function as a member of the class. We also pass the question as a placeholder so the user can see the question. Finally, we also set the correct answer as an instance attribute so we can check it.

#### shuffle

This method takes any array calls the random.shuffle method and returns the array. This is because the random.shuffle function returns None instead of the array, however my function does return the array.

#### callback

This class calls the callback function when an option has been selected. This will check if the user has selected the correct answer and then send an embed according to what they have selected.

### TriviaView class

This class is derived from the discord View class, this class needs to be instantiated from the view class so that is turned into a functioning view instead of the string view of the class e.g <TriviaView at 0x000000> would be sent in discord if it wasn’t derived from the view class.

#### \_\_init\_\_

This will call the view’s \_\_init\_\_ method, we then add an item to the view and pass all necessary data. The question’s argument is the “question” attribute of the dict that was returned by the api, this api however returns html encodings for special characters e.g &qout; which translates to “. The answers param is passed as a list of possible questions, I add the correct question to the incorrect questions, I use the syntax where you can add lists instead of appending them because appending to a list will return None instead of the list with an item appended to it. We then add the correct answer as the correct parameter.

## cogs/ui/\_truthOrDare.py

### intro

I have also seen bots that do truth or dare questions, I have also added that to my bot. The implementation works by sending two buttons after the command has been invoked (I am using ui components more to make the bot more self-explanatory) these buttons are green and red with “Truth” or “dare” as the button label. These will then offer a truth or dare question/command respective to what the user put in.

### TruthButton

#### intro

This button is derived from the Button class, this is required so that the button will be sent as a button and not <TruthButton object at 0x000000000> .

#### \_\_init\_\_

This function is called when the class is instantiated and does two things, first the button calls the Button ‘s \_\_init\_\_ method and sets the label as “Truth”, the \_\_init\_\_ method also sets the button’s color to be green.

#### callback

This function is called when the button is pressed. This function will open the dict of all truth and dare questions and look for the truth key, this will return a list of lots of truth questions of which one is chosen and sent.

### Dare button

#### intro

This button is derived from the Button class, this is required so that the button will be sent as a button and not DareButton object at 0x000000000> .

#### \_\_init\_\_

This function is called when the class is instantiated and does two things, first the button calls the Button ‘s \_\_init\_\_ method and sets the label as “Truth”, the \_\_init\_\_ method also sets the button’s color to be red.

#### callback

This function is called when the button is pressed. This function will open the dict of all truth and dare questions and look for the dare key, this will return a list of lots of dare questions of which one is chosen and sent.

### TrutghOrDareUi

#### intro

This class is derived from the view class. This ui will contain two buttons which are the Truth and Dare button.

#### \_\_init\_\_

This function will call the view’s \_\_init\_\_ function where we don’t need to pass any arguments. We add two items which are instances of the TruthUiButton and the DareUiButton.

# notes

### Politeness

From reading the documentation I may have come across as rude in places, this is because documenting code or anything programming related needs to be explicit. The nature of this documentation’s explicitness is however good for people who want to understand how the bot works and build on it, but this is not a guide on how to use the bot or a home page.