Distest Documentation

Release 0.6.3

Jake Cover, Joseph Knight

May 28, 2021

GETTING STARTED

Distest makes it easy to write application tests for discord bots.

Distest uses a secondary bot to send commands to your bot and ensure that it responds as expected.

See the *interface* reference for a list of assertions this library is capable of.

Note: Two quick note about recent changes:

1. You NEED to enable the `members` intent on the tester bot. For more information, see *Member Intent*
2. If you're using the `ext.commands.Bot` system, you will need to patch your `Bot` to allow it to listen to other discord bots, as usually commands ignore other bots. This is really easy, we provide the patching function, just take a look at the *patching* documentation page.

CHAPTER

ONE

QUICKSTART

1.1 Installation

1. **Install the library with pip:**

```
$ pip install distest
```

2. Distest works by using a second bot (the 'tester') to assert that your bot (the 'target') reacts to events appropriately. This means you will need to create a second bot account through the Discord Developer's Portal and obtain the authorization token. You also have to invite the tester to your discord guild. Additionally, be sure to enable the `SERVER MEMBERS INTENT` option, see *Member Intent* docs for more info.
3. Refer to the *Example Test Suite* for the syntax/function calls necessary to build your suite.

1.2 Usage

The tests can be run in one of two modes: *interactive* and *command-line*. In interactive mode, the bot will wait for you to initiate tests manually. In command-line mode, the bot will join a designated channel, run all designated tests, and exit with a code of 0 if all tests were successful and any other number if the one or more tests failed. This allows for automating your test suite, allowing you to implement Continuous Integration on your Discord bot!

No matter how you run your tester, the file must contain:

1. A call to *`run_dtest_bot`*, which will handle all command line arguments and run the tester in the correct mode
2. A *`TestCollector`*, which will let the bot find and run the you specify
3. One or more `Test`, which should be decorated with the *`TestCollector`*, and are the actual tests that are run.

Note: The error codes will currently be 0 on success or 1 on failure, but we plan to implement meaningful error codes

1.2.1 Interactive Mode

1. **Run the bot by running your test suite module directly (called example_tester.py here):**

```
$ python example_tester.py TARGET_ID TESTER_TOKEN
```

2. Go to the channel you want to run your tests in and call the bot using the `::run` command. You can either designate specific tests to run by name or use `::run all`

See also:

`::help` command for more commands/options.

1.2.2 Command-Line Mode

For command-line you have to designate the ID of the channel you want to run tests in (preceded by the `-c` flag). You must also designate which tests to run (with the `-r` flag). Your command should look something like this:

```
$ python example_tester.py TARGET_ID TESTER_TOKEN -c CHANNEL_ID -r all
```

The program will print test names to the console as it runs them, and then exit.

See also:

readme.md on GitHub, which contains a more in-depth look at the command properties

CHAPTER

TWO

EXAMPLE TEST SUITE

This is the `example_tester.py` file found in the root directory. It contains tests for every assertion in *Interface*. This suite is also used to test our library, in conjunction with the `example_target.py`. The easiest way to get started is to adapt this suite of tests so it's specific to your bot, then run this module with

```
$ python example_tester.py ${TARGET_ID} ${TESTER_TOKEN}
```

where `TARGET_ID` is the Discord ID of your bot, and `TESTER_TOKEN` is the auth token for your testing bot.

```
"""
A functional demo of all possible test cases. This is the format you will want to use with your testing bot.

    Run with:
        python example_tests.py TARGET_NAME TESTER_TOKEN
"""
import asyncio
import sys
from distest import TestCollector
from distest import run_dtest_bot
from discord import Embed, Member, Status
from distest import TestInterface

# The tests themselves

test_collector = TestCollector()
created_channel = None


@test_collector()
async def test_ping(interface):
    await interface.assert_reply_contains("ping?", "pong!")


@test_collector()
async def test_delayed_reply(interface):
    message = await interface.send_message(
        "Say some stuff, but at 4 seconds, say 'yeet'"
    )
    await interface.get_delayed_reply(5, interface.assert_message_equals, "yeet")


@test_collector()
async def test_reaction(interface):
    await interface.assert_reaction_equals("React with \u2714 please!", u"\u2714")
```

(continues on next page)

(continued from previous page)

```


@test_collector()
async def test_reply_equals(interface):
    await interface.assert_reply_equals("Please say 'epic!'", "epic!")


@test_collector()
async def test_channel_create(interface):
    await interface.send_message("Create a tc called yeet")
    created_channel = await interface.assert_guild_channel_created("yeet")


# @test_collector
# async def test_pin_in_channel(interface):
#     await interface.send_message("Pin 'this is cool' in yeet")
#     await interface.assert_guild_channel_pin_content_equals(created_channel )


@test_collector()
async def test_channel_delete(interface):
    await interface.send_message("Delete that TC bro!")
    await interface.assert_guild_channel_deleted("yeet")


@test_collector()
async def test_silence(interface):
    await interface.send_message("Shhhhh...")
    await interface.ensure_silence()


@test_collector()
async def test_reply_contains(interface):
    await interface.assert_reply_contains(
        "Say something containing 'gamer' please!", "gamer"
    )


@test_collector()
async def test_reply_matches(interface):
    await interface.assert_reply_matches(
        "Say something matching the regex `[0-9]{1,3}`", r"[0-9]{1,3}"
    )


@test_collector()
async def test_ask_human(interface):
    await interface.ask_human("Click the Check!")


@test_collector()
async def test_embed_matches(interface):
    embed = (
        Embed(
            title="This is a test!",
            description="Descriptive",
            url="http://www.example.com",
```

(continues on next page)

(continued from previous page)

```
            color=0x00FFCC,
        )
        .set_author(name="Author")
        .set_thumbnail(
            url="https://upload.wikimedia.org/wikipedia/commons/4/40/Test_Example_
↪%28cropped%29.jpg"
        )
        .set_image(
            url="https://upload.wikimedia.org/wikipedia/commons/4/40/Test_Example_
↪%28cropped%29.jpg"
        )
    )

    # This image is in WikiMedia Public Domain
    await interface.assert_reply_embed_equals("Test the Embed!", embed)


@test_collector()
async def test_embed_regex(interface):
    patterns = {
        "title": "Test",
        "description": r"Random Number: [0-9]+",
    }
    await interface.assert_reply_embed_regex("Test the Embed regex!", patterns)


@test_collector()
async def test_embed_part_matches(interface):
    embed = Embed(title="Testing Title.", description="Wrong Description")
    await interface.assert_reply_embed_equals(
        "Test the Part Embed!", embed, attributes_to_check=["title"]
    )


@test_collector()
async def test_reply_has_image(interface):
    await interface.assert_reply_has_image("Post something with an image!")


@test_collector()
async def test_reply_on_edit(interface):
    message = await interface.send_message("Say 'Yeah, that cool!'")
    await asyncio.sleep(1)
    await interface.edit_message(message, "Say 'Yeah, that is cool!'")
    await interface.assert_message_contains(message, "Yeah, that is cool!")


@test_collector()
async def test_send_message_in_channel(interface):
    message = await interface.send_message("Say stuff in another channel")
    await interface.wait_for_message_in_channel(
        "here is a message in another channel", 694397509958893640
    )


# Actually run the bot

```

(continues on next page)

(continued from previous page)

```
148 if __name__ == "__main__":
149     run_dtest_bot(sys.argv, test_collector)
```

CHAPTER

THREE

THE MEMBER INTENT

Discord recently changed what you have to do for bots to be able to access server member informaiton, meaning that without changes, calling `guild.members` will return an empty list, which is no good!!

To fix this, we need to do two things:

1. **Enable the *Privileged Gateway Intent* for Server Members.**
    a. To do this. go to the Discord developer portal and select your tester bot
    b. Select the bot tab
    c. Enable the `SERVER MEMBERS INTENT` and the `PRESENCE INTENT`` sliders
2. Update distest! There are also changes that need to be made on our side. They have been made, but be sure you update to 0.4.9 or newer to get the changes!

Now, you should be good to go. Have fun testing!

Quick note - For some godforsaken reason, the `on_member_update` event is just horribly slow and unreliable. I'm not really sure what to do about this, but be forewarned if you want to use it!

CHAPTER

FOUR

MAIN FUNCTIONS

distest.**run_command_line_bot**(*target*, *token*, *tests*, *channel_id*, *stats*, *collector*, *timeout*)

Start the bot in command-line mode. The program will exit 1 if any of the tests failed.

Relies on *run_dtest_bot()* to parse the command line arguments and pass them here. Not really meant to be called by the user.

Parameters

- **target** (*str*) – The display name of the bot we are testing.
- **token** (*str*) – The tester's token, used to log in.
- **tests** (*str*) – List of tests to run.
- **channel_id** (*int*) – The ID of the channel in which to run the tests.
- **stats** (*bool*) – Determines whether or not to display stats after run.
- **collector** (TestCollector) – The collector that gathered our tests.
- **timeout** (*int*) – The amount of time to wait for responses before failing tests.

distest.**run_dtest_bot**(*sysargs*, *test_collector*, *timeout=5*)

This is the function you will call in your test suite's `if __name__ == "__main__":` statement to get the bot started.

Parameters

- **sysargs** (*list*) – The list returned by `sys.argv`, this function parses it and will handle errors in format
- **test_collector** (TestCollector) – The *Collector* that has been used to decorate the tests
- **timeout** (*int*) – An optional parameter to override the amount of time to wait for responses before failing tests. Defaults to 5 seconds.

distest.**run_interactive_bot**(*target_name*, *token*, *test_collector*, *timeout=5*)

Run the bot in interactive mode.

Relies on *run_dtest_bot()* to parse the command line arguments and pass them here. Not really meant to be called by the user.

Parameters

- **target_name** (*str*) – The display name of the bot we are testing.
- **token** (*str*) – The tester's token, used to log in.
- **test_collector** (TestCollector) – The collector that gathered our tests.

- **timeout** (*int*) – The amount of time to wait for responses before failing tests.

CHAPTER

FIVE

INTERFACE

This is the most important class in the library for you, as it contains all the assertions and tools you need to interface with the library. Generally broken down into a few overall types:

- Message (i.e. `assert_message_contains`): Does not send it's own message, so it require a `Message` to be passed in.
- **Reply (i.e. `assert_reply_contains`): Sends a message containing the text in *contents* and analyzes messages sent afte**
  - Use `get_delayed_reply` to wait an amount of time before checking for a reply
- Embed (i.e. `assert_embed_equals`): Sends a message then checks the embed of the response against a list of attributes
- Other Tests (i.e. `ask_human`): Some tests do weird things and don't have a clear category.
- Interface Functions (i.e. `connect`, `send_message`): Help other tests but also can be useful in making custom tests out of the other tests.

---

class distest.TestInterface.**TestInterface**(*client*, *channel*, *target*)

All the tests, and some supporting functions. Tests are designed to be run by the tester and mixed together in order to actually test the bot.

Note: In addition to the tests failing due to their own reasons, all tests will also fail if they timeout. This period is specified when the bot is run.

Note: Some functions (`send_message` and `edit_message`) are helper functions rather than tests and serve to bring some of the functionality of the discord library onto the same level as the tests.

Note: `assert_reply_*` tests will send a message with the passed content, while `assert_message_*` tests require a `Message` to be passed to them. This allows for more flexibility when you need it and an easier option when you don't.

Parameters

- **client** (`DiscordCliInterface`) – The discord client of the tester.
- **channel** (*discord.TextChannel*) – The discord channel in which to run the tests.
- **target** (*discord.Member*) – The bot we're testing.

async ask_human(*query*)
Ask a human for an opinion on a question using reactions.

Currently, only yes-no questions are supported. If the human answers 'no', the test will be failed. Do not use if avoidable, since this test is not really automateable. Will fail if the reaction is wrong or takes too long to arrive

Parameters **query** (*str*) – The question for the human.

Raises HumanResponseTimeout, HumanResponseFailure

async static assert_embed_equals(*message*, *matches*, *attributes_to_prove=None*)
If `matches` doesn't match the embed of `message`, fail the test.

Checks only the attributes from `attributes_to_prove`.

Parameters

- **message** – original message
- **matches** – `embed` object to compare to
- **attributes_to_prove** – a string list with the attributes of the embed, which are to compare This are all the Attributes you can prove: "title", "description", "url", "color", "author", "video", "image" and "thumbnail".

Returns message

Return type discord.Message

async static assert_embed_regex(*message*, *patterns*)
If regex patterns `patterns` cannot be found in the embed of `message`, fail the test.

Checks only the attributes from the dictionary keys of `patterns`.

Parameters

- **message** – original message
- **patterns** – a dict with keys of the attributes and regex values.

Returns message

Return type discord.Message

async assert_guild_channel_created(*channel_name*, *timeout=None*)
Assert that the next channel created matches the name given

Parameters

- **channel_name** (*str*) – The name of the channel to check for
- **timeout** (*float*) – The num of seconds to wait, defaults to the timeout provided at the start

Returns The channel that was created

Return type discord.abc.GuildChannel

Raises NoResponseError

async assert_guild_channel_deleted(*channel_name*, *timeout=None*)
Assert that the next channel deleted matches the name given

TODO: check what the deleted channel actually returns

Parameters

- **channel_name** (*str*) – The name of the channel to check for
- **timeout** (*float*) – The num of seconds to wait, defaults to the timeout provided at the start

Returns The channel that was deleted

Return type discord.abc.GuildChannel

Raises NoResponseError

async static assert_message_contains(*message*, *substring*)
If *message* does not contain the given substring, fail the test.

Parameters

- **message** (*discord.Message*) – The message to test.
- **substring** (*str*) – The string to test *message* against.

Returns *message*

Return type discord.Message

Raises ResponseDidNotMatchError

async static assert_message_equals(*message*, *matches*)
If `message` does not match a string exactly, fail the test.

Parameters

- **message** (*discord.Message*) – The message to test.
- **matches** (*str*) – The string to test *message* against.

Returns *message*

Return type discord.Message

Raises ResponseDidNotMatchError

async static assert_message_has_image(*message*)
Assert *message* has an attachment. If not, fail the test.

Parameters **message** (*discord.Message*) – The message to test.

Returns *message*

Return type discord.Message

Raises UnexpectedResponseError

async static assert_message_matches(*message*, *regex*)
If *message* does not match a regex, fail the test.

Requires a properly formatted Python regex ready to be used in the `re` functions.

Parameters

- **message** (*discord.Message*) – The message to test.
- **regex** (*str*) – The regular expression to test *message* against.

Returns *message*

Return type discord.Message

Raises ResponseDidNotMatchError

async assert_reaction_equals(*contents*, *emoji*)
Send a message and ensure that the reaction is equal to *emoji*. If not, fail the test.

Parameters

- **contents** (*str*) – The content of the trigger message. (A command)
- **emoji** (*discord.Emoji*) – The emoji that the reaction must equal.

Returns The resultant reaction object.

Return type discord.Reaction

Raises ReactionDidNotMatchError

async assert_reply_contains(*contents*, *substring*)
Send a message and wait for a response. If the response does not contain the given substring, fail the test.

Parameters

- **contents** (*str*) – The content of the trigger message. (A command)
- **substring** (*str*) – The string to test against.

Returns The reply.

Return type discord.Message

Raises ResponseDidNotMatchError

async assert_reply_embed_equals(*message*, *equals*, *attributes_to_check=None*)
Send a message and wait for an embed response. If the response does not match the given embed in the listed attributes, fail the test

See examples in example_target.py for examples of use.

Parameters

- **message** –
- **equals** – `embed` object to compare to
- **attributes_to_check** – a string list with the attributes of the embed, which are to compare This are all the Attributes you can prove: “title”, “description”, “url”, “color”, “author”, “video”, “image” and “thumbnail”.

Returns message

Return type discord.Message

async assert_reply_embed_regex(*message*, *patterns*)

Send a message and wait for a response. If the response is not an embed or does not match the regex, fail the test.

See examples in example_target.py for examples of use.

Parameters

- **message** –
- **patterns** – A dict of the attributes to check. See `assert_message_contains` for more info on this.

Returns message

Return type discord.Message

async assert_reply_equals(*contents*, *matches*)
Send a message and wait for a response. If the response does not match the string exactly, fail the test.

Parameters

- **contents** (*str*) – The content of the trigger message. (A command)
- **matches** (*str*) – The string to test against.

Returns The reply.

Return type discord.Message

Raises ResponseDidNotMatchError

async assert_reply_has_image(*contents*)
Send a message consisting of *contents* and wait for a reply.

Check that the reply contains a `discord.Attachment`. If not, fail the test.

Parameters **contents** (*str*) – The content of the trigger message. (A command)

Returns The reply.

Return type discord.Message

Raises ResponseDidNotMatchError, NoResponseError

async assert_reply_matches(*contents*, *regex*)
Send a message and wait for a response. If the response does not match a regex, fail the test.

Requires a properly formatted Python regex ready to be used in the `re` functions.

Parameters

- **contents** (*str*) – The content of the trigger message. (A command)
- **regex** (*str*) – The regular expression to test against.

Returns The reply.

Return type discord.Message

Raises ResponseDidNotMatchError

async connect(*channel*)
Connect to a given VoiceChannel :param channel: The VoiceChannel to connect to. :return:

async disconnect()
Disconnect from the VoiceChannel; Doesn't work if the Bot isn't connected. :return:

async static edit_message(*message*, *new_content*)
Modify a message. Most tests and `send_message` return the `discord.Message` they sent, which can be used here. **Helper Function**

Parameters

- **message** (*discord.Message*) – The target message. Must be a `discord.Message`
- **new_content** (*str*) – The text to change *message* to.

Returns *message* after modification.

Return type discord.Message

async ensure_silence()
Assert that the bot does not post any messages for some number of seconds.

Raises UnexpectedResponseError, TimeoutError

async get_delayed_reply(*seconds_to_wait*, *test_function*, **args*)
Get the last reply after a specific time and check it against a given test.

Parameters

- **seconds_to_wait** (*float*) – Time to wait in s
- **test_function** (*method*) – The function to call afterwards, without parenthesis (assert_message_equals, not assert_message_equals()!)
- **args** – The arguments to pass to the test, requires the same number of args as the test function. Make sur to pass in **all** args, including kwargs with defaults. NOTE: this policy may change if it becomes kinda stupid down the road.

Return type Method

Raises **SyntaxError** –

Returns The instance of the test requested

async send_message(*content*)
Send a message to the channel the test is being run in. **Helper Function**

Parameters **content** (*str*) – Text to send in the message

Returns The message that was sent

Return type discord.Message

async wait_for_event(*event*, *check=None*, *timeout=None*)
A wrapper for the discord.py function `wait_for`, tuned to be useful for distest.

See https://discordpy.readthedocs.io/en/latest/api.html#event-reference for a list of events.

Parameters

- **event** – The discord.py event, as a string and with the `on_` removed from the beginning.
- **check** (*Callable[..,bool]*) – A check function that all events of the type are ran against. Should return true when the desired event occurs, takes the event's params as it's params
- **timeout** (*float*) – How many seconds to wait for the event to occur.

Returns The parameters of the event requested

Raises NoResponseError

async wait_for_message()
Wait for the bot the send any message. Will fail on timeout, but will ignore messages sent by anything other that the target.

Returns The message we've been waiting for.

Return type discord.Message

Raises NoResponseError

async wait_for_message_in_channel(*content*, *channel_id*)
Send a message with `content` and returns the next message that the targeted bot sends. Used in many other tests.

Parameters

- **content** (*str*) – The text of the trigger message.

- **channel_id** (*int*) – The id of the channel that the message is sent in.

Returns The message we've been waiting for.

Return type discord.Message

Raises NoResponseError

async wait_for_reaction(*message*)

Assert that `message` is reacted to with any reaction.

Parameters **message** (*discord.Message*) – The message to test with

Returns The reaction object.

Return type discord.Reaction

Raises **NoReactionError** –

async wait_for_reply(*content*)

Send a message with `content` and returns the next message that the targeted bot sends. Used in many other tests.

Parameters **content** (*str*) – The text of the trigger message.

Returns The message we've been waiting for.

Return type discord.Message

Raises NoResponseError

---

class distest.TestInterface.**Test**(*name*, *func*, *needs_human=False*)

Holds data about a specific test.

Parameters

- **name** (*str*) – The name of the test, checks this against the valid test names
- **func** (*function*) – The function in the tester bot that makes up this test
- **needs_human** (*bool*) – Weather or not this test will require human interaction to complete

Raises ValueError

CHAPTER

SIX

ENUMERATIONS

The following enumeration (subclass of `enum.Enum`) is used to indicate the result of a run test.

class TestResult
Specifies the result of a test.

UNRUN
Test has not been run in this session

SUCCESS
Test succeeded

FAILED
Test has failed.

CHAPTER

SEVEN

BOT

Contains the discord clients used to run tests.

DiscordBot contains the logic for running tests and finding the target bot

DiscordInteractiveInterface is a subclass of *DiscordBot* and contains the logic to handle commands sent from discord to run tests, display stats, and more

DiscordCliInterface is a subclass of *DiscordInteractiveInterface* and simply contains logic to start the bot when it wakes up

---

class distest.bot.**DiscordBot**(*target_id*)

Discord bot used to run tests. This class by itself does not provide any useful methods for human interaction, and is just used as a superclass of the two interfaces, *DiscordInteractiveInterface* and *DiscordCliInterface*

Parameters **target_id** (*str*) – The name of the target bot, used to ensure that the target user is actually present in the server. Good for checking for typos or other simple mistakes.

async run_test(*test*, *channel*, *stop_error=False*)

Run a single test in a given channel.

Updates the test with the result and returns it

Parameters

- **test** (Test) – The *Test* that is to be run
- **channel** (*discord.TextChannel*) – The
- **stop_error** – Weather or not to stop the program on error. Not currently in use.

Returns Result of the test

Return type *TestResult*

---

class distest.bot.**DiscordInteractiveInterface**(*target_id*, *collector*, *timeout=5*)

A variant of the discord bot which commands sent in discord to allow a human to run the tests manually.

Does NOT support CLI arguments

Parameters

- **target_id** (*str*) – The name of the bot to target (Username, no discriminator)
- **collector** (TestCollector) – The instance of Test Collector that contains the tests to run

- **timeout** (*int*) – The amount of time to wait for responses before failing tests.

async on_message(*message*)
Handle an incoming message, see `discord.event.on_message()` for event reference.

Parse a message, can ignore it or parse the message as a command and run some tests or do one of the alternate functions (stats, list, or help)

Parameters **message** (*discord.Message*) – The message being recieved, passed by discord.py

async on_ready()
Report when the bot is ready for use and report the available tests to the console

async run_tests(*channel*, *name*)
Helper function for choosing and running an appropriate suite of tests Makes sure only tests that still need to be run are run, also prints to the console when a test is run

Parameters

- **channel** (*discord.TextChannel*) – The channel in which to run the tests
- **name** (*str*) – Selector string used to determine what category of test to run

---

class distest.bot.**DiscordCliInterface**(*target_id*, *collector*, *test*, *channel_id*, *stats*, *timeout*)
A variant of the discord bot which is designed to be run off command line arguments.

Parameters

- **target_id** (*str*) – The name of the bot to target (Username, no discriminator)
- **collector** (TestCollector) – The instance of Test Collector that contains the tests to run
- **test** (*str*) – The name of the test option (all, specific test, etc)
- **channel_id** (*int*) – The ID of the channel to run the bot in
- **stats** (*bool*) – If true, run in hstats mode.

async on_ready()
Run all the tests sequentially when the bot becomes awake and exit when the tests finish. The CLI should run all by itself without prompting, and this allows it to behave that way.

run(*token*)
Override of the default run() that returns failure state after completion. Allows the failure to cascade back up until it is processed into an exit code by `run_command_line_bot()`

Parameters **token** (*str*) – The tester bot token

Returns Returns 1 if the any test failed, otherwise returns zero.

Return type int

CHAPTER

EIGHT

COLLECTOR

The TestCollector Class and some supporting code.

Each test function in the tester bot should be decorated with an instance of TestCollector(), and must have a unique name. The TestCollector() is then passed onto the bot, which runs the tests.

---

class distest.collector.**TestCollector**

Used to group tests and pass them around all at once.

Tests can be either added with *add* or by using `@TestCollector` to decorate the function, as seen in the sample code below. Is very similar in function to `Command` from discord.py, which you might already be familiar with.

```
@test_collector()
async def test_reply_contains(interface):
    await interface.assert_reply_contains(
        "Say something containing 'gamer' please!", "gamer"
    )


@test_collector()
async def test_reply_matches(interface):
    await interface.assert_reply_matches(
        "Say something matching the regex `[0-9]{1,3}`", r"[0-9]{1,3}"
```

add(*function*, *name=None*, *needs_human=False*)

Adds a test function to the group, if one with that name is not already present

Parameters

- **function** (*func*) – The function to add
- **name** (*str*) – The name of the function to add, defaults to the function name but can be overridden with the provided name just like with `discord.ext.commands.Command`. See sample code above.
- **needs_human** (*bool*) – Optional boolean, true if the test requires a human interaction

find_by_name(*name*)

Return the test with the given name, return `None` if it does not exist.

Parameters **name** (*str*) – The name of the test

Return type *Test*, none

CHAPTER

NINE

EXCEPTIONS

Stores all the Exceptions that can be called during testing.

Allows for a more through understanding of what went wrong. Not all of these are currently in use.

CHAPTER

TEN

CONTRIBUTING

Hey! You're here because you want to contribute to the bot, and that's awesome! Here are some notes about contributions:

- Please open an issue for your contribution and tag it with contribution to discuss it. Because of my time constraints, I probably won't have much time to implement new features myself, but if you make a feature and PR it in, I'll be more than happy to spend a bit of time testing it out and add it to the project. The other thing is to make sure you check the github project to see if there is someone else already working on it who you can help.
- You may need to install the additional requirements from *requirements-dev.txt*. This is as simple as running *pip install -r requirements-dev.txt*. This larger list mostly includes things like black for formatting and sphinx for doc testing.
- If you are adding new test types, please make sure you test them well to make sure they work as intended, and please add a demo of them in use to the *example_tests()* for others to see. When you are done, please open a PR and I'll add it in!
- I use Black for my code formatting, it would be appreciated if you could use it when contributing as well. I will remind you when you make a PR if you don't, it is essential to make sure that diffs aren't cluttered up with silly formatting changes. Additionally, CodeFactor *should* be tracking code quality and doing something with PRs. We will see soon exactly how that will work out.
- To build the docs for testing purposes, cd into the docs folder and run *make testhtml*. This will build to a set of HTML files you can view locally.
- Also, if you just want to propose an idea, create an issue and tag it with enhancement. The library is missing tons of features, so let me know what you want to see, and if I have time I'll see about getting around to addign it. Thank you for your help!

CHAPTER

ELEVEN

PATCHES

Contains the code required to patch out the fact that `Bot` class ignores messages from other bots.

This should be used if you have a target bot that uses the `ext.commands.Bot` system, as otherwise it's commands will ignore messages from the tester bot.

11.1 Usage

Simply put the below code into your **main bot** and then when testing, the bot will no longer ignore other bots!

```
bot = commands.Bot(command_prefix='$')

# Do anything you want for this if, be it env vars, command line args, or␣
↪the likes.
if sys.argv[2] == "TESTING":
    from distest.patches import patch_target
    bot = patch_target(bot)
```

11.2 Docs

distest.patches.**patch_target**(*bot*)

Patches the target bot. It changes the `process_commands` function to remove the check if the received message author is a bot or not.

Parameters **bot** (*discord.ext.commands.Bot*) –

Returns The patched bot.

CHAPTER

TWELVE

META DOCUMENTATION PAGES

- genindex
- modindex
- search

PYTHON MODULE INDEX

d

INDEX

M

O

P

R

S

T

U

W