Distest Documentation

Release 1.0

Jake Cover, Joseph Knight

Apr 10, 2020

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.

CHAPTER 1

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.
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_NAME 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_NAME 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 2

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_NAME} ${TESTER_TOKEN}
```

where `TARGET_NAME` is the display name of your discord 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_interactive_bot, run_dtest_bot
from discord import Embed

# 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):
```

(continues on next page)

(continued from previous page)

```
    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")


@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!")
```

(continues on next page)

(continued from previous page)

```

@test_collector()
async def test_embed_matches(interface):
    embed = (
        Embed(
            title="This is a test!",
            description="Descriptive",
            url="http://www.example.com",
            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_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 asyncio.sleep(1)
    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)

```
136 if __name__ == "__main__":
137     run_dtest_bot(sys.argv, test_collector)
```

CHAPTER 3

Main Functions

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_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_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 4

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.

CHAPTER 5

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 6

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.

run_test(*test: distest.TestInterface.Test*, *channel: discord.channel.TextChannel*, *stop_error=False*)
→ distest.TestInterface.TestResult

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: distest.collector.TestCollector*, *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.

on_message(*message: discord.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

on_ready()

Report when the bot is ready for use and report the available tests to the console

run_tests(*channel: discord.channel.TextChannel*, *name: str*)

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.

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*) → int

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 7

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.

```
    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 )
```

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

CHAPTER 8

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.

class distest.exceptions.**TestRequirementFailure**
Base class for the special errors that are raised when an expectation is not met during testing

class distest.exceptions.**TestRequirementFailure**
Base class for the special errors that are raised when an expectation is not met during testing

class distest.exceptions.**TestRequirementFailure**
Base class for the special errors that are raised when an expectation is not met during testing

CHAPTER 9

Contributing

CHAPTER 10

Meta Documentation Pages

- genindex
- modindex
- search

Python Module Index

d

Index