

---

**duckpy**  
*Release 0.1.1*

**Feb 25, 2018**



---

# Contents

---

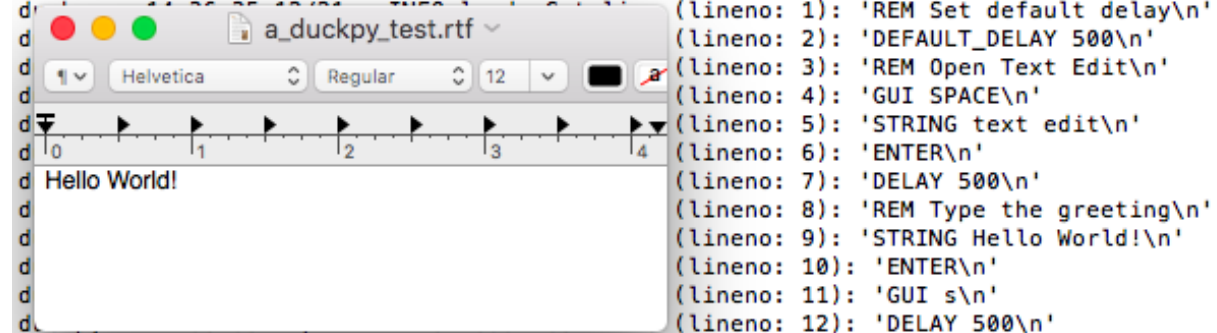
<b>1</b>	<b>Module Docs</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>7</b>
2.1	Dependencies . . . . .	7
2.2	Running setup.py . . . . .	7
<b>3</b>	<b>Usage</b>	<b>9</b>
3.1	Command-Line Execution . . . . .	9
3.2	Feature Set . . . . .	9
3.3	Failsafe . . . . .	10
3.4	Logging/Debugging . . . . .	10
3.5	Python Execution . . . . .	11
<b>4</b>	<b>Approach</b>	<b>13</b>
4.1	Usage Scenarios . . . . .	13
4.2	Code Breakdown . . . . .	14
<b>5</b>	<b>Contact</b>	<b>15</b>
<b>6</b>	<b>License (MIT)</b>	<b>17</b>
<b>7</b>	<b>TODO</b>	<b>19</b>
<b>8</b>	<b>Changelog</b>	<b>21</b>
	<b>Python Module Index</b>	<b>23</b>



**Note:** This project is in very early development and is not thoroughly tested.

Initially created to help those looking to escape the tedious workings involved in testing and writing `duckyscripts` for hak5's `Rubber Ducky`, duckpy is capable of executing duckyscript right on a target machine by translating given commands into executable Python functions. It's essentially **an open source duckyscript interpreter written in Python**.

```
[duckpy > python3 -m duckpy -v hello.txt
duckpy - 14:36:35 12/21 - INFO:load: Loading script at 'hello.txt'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 0): 'REM hello.txt\n'
d (lineno: 1): 'REM Set default delay\n'
d (lineno: 2): 'DEFAULT_DELAY 500\n'
d (lineno: 3): 'REM Open Text Edit\n'
d (lineno: 4): 'GUI SPACE\n'
d (lineno: 5): 'STRING text edit\n'
d (lineno: 6): 'ENTER\n'
d (lineno: 7): 'DELAY 500\n'
d (lineno: 8): 'REM Type the greeting\n'
d (lineno: 9): 'STRING Hello World!\n'
d (lineno: 10): 'ENTER\n'
d (lineno: 11): 'GUI s\n'
d (lineno: 12): 'DELAY 500\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 13): 'STRING a_duckpy_test\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 14): 'ENTER\n'
duckpy - 14:36:35 12/21 - INFO:load: Finished loading
duckpy - 14:36:35 12/21 - INFO:run: Executing script at: 'hello.txt'
duckpy - 14:36:35 12/21 - INFO:run: Running line 0: 'REM hello.txt'
duckpy - 14:36:35 12/21 - INFO:run: Running line 1: 'REM Set default delay'
duckpy - 14:36:35 12/21 - INFO:run: Running line 2: 'DEFAULT_DELAY 500'
duckpy - 14:36:35 12/21 - INFO:run: Running line 3: 'REM Open Text Edit'
duckpy - 14:36:35 12/21 - INFO:run: Running line 4: 'GUI SPACE'
duckpy - 14:36:35 12/21 - INFO:run: Running line 5: 'STRING text edit'
duckpy - 14:36:36 12/21 - INFO:run: Running line 6: 'ENTER'
duckpy - 14:36:37 12/21 - INFO:run: Running line 7: 'DELAY 500'
duckpy - 14:36:38 12/21 - INFO:run: Running line 8: 'REM Type the greeting'
duckpy - 14:36:38 12/21 - INFO:run: Running line 9: 'STRING Hello World!'
duckpy - 14:36:38 12/21 - INFO:run: Running line 10: 'ENTER'
duckpy - 14:36:39 12/21 - INFO:run: Running line 11: 'GUI s'
duckpy - 14:36:40 12/21 - INFO:run: Running line 12: 'DELAY 500'
duckpy - 14:36:41 12/21 - INFO:run: Running line 13: 'STRING a_duckpy_test'
duckpy - 14:36:41 12/21 - INFO:run: Running line 14: 'ENTER'
duckpy - 14:36:42 12/21 - INFO:run: Finished execution
duckpy >
```





duck.py: Core module behind duckpy's functionality.

**class** `duckpy.duck.DuckyCommand` (*dline*, *lineno=-1*, *default\_delay=0*, *script=None*)  
Execute a line of duckscript in Python. Will log to the logger entitled **duckpy**.

### Parameters

- **dline** (*str*) – Raw duckscript line to execute/model.
- **lineno** (*int*) – Line number of the given duckscript line if it is a part of a script (defaults to `-1`, which indicates the line is not in a script).
- **default\_delay** (*int*) – Default delay to use while executing. Essentially determines how long to wait before executing a command (except in the case of `REM`, where this delay is skipped).
- **script** (*dict*) – Dictionary of *DuckyScript* methods used for setting default delays and repeating commands. Used internally by the *DuckyScript* class (see *DuckyScript.load()*).

**Raises ValueError** – If `scripts` kwarg does not contain all necessary keys for execution.

### `_to_python` (*dline*)

Parses the given duckscript line into a Python function. The command in the line (i.e. the substring that lies before the first space) will be parsed and then matched against known commands. If a match is found, the appropriate Python function is constructed and returned. If a match is not found, a function will still be constructed and returned, however the line will be interpreted as a series of keys to press instead of a command (for instance if `GUI r` was given, `GUI` will be seen as a key and not a command). A pre-check is done prior to the matching process, to ensure that the given command/key combo is valid.

Here's a list of translations for commands and their Python functions:

- `REM`: `duckpy.duck._cmd_rem()`
- `DELAY`: `duckpy.duck._cmd_delay()`
- `DEFAULT_DELAY`: This is done internally, see the `default_delay` parameter in the source code for details.

- **STRING**: `pyautogui.typewrite()`
- **REPEAT**: This is again done internally, see source code for more details.
- **(Other)**: `pyautogui.hotkey()`.

**Parameters** `dline` (*str*) – Duckyscript line to translate.

**Returns** Python function that when executed, will simulate the given duckyscript line.

**Raises** **ValueError** – If an invalid command is given.

#### **default\_delay**

`default_delay` property. Value of this property will be determined by whether or not this command is a part of a script. If it is, then the script's default delay value will be used, otherwise this instance's value will be used.

**Return type** `int`

**Returns** default delay being used by the command

**Raises** **KeyError** – If `get_default_delay` method of script cannot be found in `_script`.

#### **execute()**

Execute the line of duckyscript that was given during class construction. This will block until finished.

**Returns** `None`

#### **class** `duckpy.duck.DuckyScript` (*dpath*)

Representation of a duckyscript file. Allows for reading, parsing and execution. The given Duckyscript file to represent will be parsed and loaded on creation.

**Parameters** `dpath` (*str*) – Path to duckyscript (text) file

**Raises** **OSErrror** – If given path to a duckyscript file either does not exist or cannot be read.

#### **\_\_get\_default\_delay()**

Method for retrieving the `default_delay` attribute. This is given to `DuckyCommand` instances so that they may get the default delay for the script.

**Returns** `_default_delay`

#### **\_\_set\_default\_delay** (*new\_delay*)

Method for setting the `default_delay` attribute. This is given to `DuckyCommand` instances so that they may set the default delay for the script.

**Parameters** `new_delay` (*int*) – New default delay to set.

#### **default\_delay**

`default_delay` property, using `__get_default_delay` as the getter method. `default_delay` is made into a property so the getter and setter methods can be passed onto children `DuckyCommand` instances.

#### **load()**

Loads the duckyscript and parses it into python functions for execution. Note that every time this method is called the duckyscript will be read and parsed (i.e. this method supports reloading of scripts).

**Raises** **ValueError** – If line in duckyscript file cannot be parsed.

**Returns** `None`

#### **run()**

Runs the duckyscript file (loading it if necessary) by executing all of the parsed commands sequentially. Will 'pass through' any errors that may possibly occur during execution.

**Returns** `None`



`duckpy.duck._cmd_delay` (*ms*)

Simulates the DELAY command by sleeping for the given number of milliseconds.

**Parameters** `ms` (*int*) – Number of milliseconds to sleep.

**Returns** None

`duckpy.duck._cmd_rem` (*comment*)

Function that just takes in a comment and does nothing with it to simulate the REM command.

**Parameters** `comment` (*str*) – Comment string passed to the REM command.

**Returns** None

`duckpy.duck._cmd_repeat` (*dcmd, num\_times*)

Simulates the REPEAT command by continually executing the given *DuckyCommand* for `num_times` number of times.

**Parameters**

- `dcmd` (*DuckyCommand*) – Command to repeat
- `num_times` (*int*) – Number of times to repeat

**Returns** None

`duckpy.duck._set_args` (*func, \*args, \*\*kwargs*)

Decorator that returns a new function which will execute the given function with the given arguments. Allows for a function's arguments to be set ahead of time. The returned function will not take in any arguments and will have the following attributes:

- `args`: arguments that were passed to the wrapped function
- `kwargs`: kwargs that were passed to the wrapped function
- `__name__`: Will be overwritten with the wrapped function's name.

**Parameters**

- `func` – Function to set the arguments for.
- `args` – Arguments to pass to `func`.
- `kwargs` – Keyword arguments to pass to `func`.

**Returns** Function that when executed, will call the given function with the given arguments.

`duckpy.duck.get_alias` (*dcmd*)

Returns the known alias for the given ducky command. If the given command does not have an alias, then None is returned.

**Parameters** `dcmd` (*str*) – Duckyscript command to get the known alias of (case sensitive).

**Returns** Command's alias if it exists, or None.

**Raises**

- **ValueError** – if given ducky command is invalid.
- **TypeError** – if the given ducky command is already an alias.

`duckpy.duck.get_alias_target` (*dalias*)

Returns the duckyscript command that the given duckyscript alias targets.

**Parameters** `dalias` (*str*) – Duckyscript alias (case sensitive)

**Returns** Duckyscript command the alias targets.

**Raises `ValueError`** – If given duckyscript alias is not valid (this includes if the given duckyscript command is not an alias).

`duckpy.duck.is_valid_alias (dcmd)`

Checks to see if the given ducky command is an alias (i.e. in the global variable `ALIASES`), returning `True` if it is. If the given command is not an alias for another, then `False` is returned.

---

**Note:** `False` will still be returned if a command is given that has an alias. For instance, `is_alias('ESC')` is `True`, `is_alias('ESCAPE')` is `False`.

---

**Parameters `dcmd` (*str*)** – Duckyscript command to check (case sensitive).

**Return type** `bool`

`duckpy.duck.is_valid_cmd (dcmd)`

Checks to see if the given ducky command is valid. Note that if a command is not valid in the eyes of this function, then the interpreter will not be able to execute it.

**Parameters `dcmd` (*str*)** – Duckyscript command to check the validity of (case sensitive). Aliases and keys can also be given.

**Return type** `bool`

**Returns** `True` if valid, `False` otherwise

`duckpy.duck.main (cli_args=None)`

Takes in a duckyscript file, parses it and executes it. This function can be called by executing `python -m duckpy` however it can also be called manually by passing command line arguments through `args`

**Parameters `cli_args` (*str*)** – Pass command line arguments directly. Example:  
`"my_payload.txt -v"`

**Returns** `None`

`duckpy.duck.translate_key (dkey)`

Translates the given duckyscript key into a `pyautogui` key name. A three step process is used to do this:

1. Check if the key is an alias, and if it is, get its target.
2. Translate the key using either `TRANSLATE_KEYS` if the key has a special translation or by setting the key to all lowercase. If a key such as `CTRL-ALT` is given (two key modifier) then each key in the macro will be translated individually.
3. Check if the key is found in `pyautogui.KEYBOARD_KEYS`, returning `None` if it isn't found.

A tuple of the translated key is returned, so that it may be passed right into `pyautogui` commands even if more than one key is translated in the case a modifier was given.

**Parameters `dkey` (*str*)** – Duckyscript key to translate.

**Return type** `tuple`

**Returns** `pyautogui` name of the given key inside a tuple if it could be translated, otherwise (`None`, ). If a given key name translates to more than one key, then a tuple of each key is returned (i.e. `CTRL-ALT -> ('ctrl', 'alt')`)

A `setup.py` file has been provided to assist in managing duckpy's installation, packaging and development. Dependencies should be installed first however.

### 2.1 Dependencies

The only dependency required for use is `pyautogui`, whose installation instructions can be found [here](#).

---

**Note:** `pyautogui` has some platform-specific pre-install steps, which must be completed before duckpy is installed.

---

### 2.2 Running `setup.py`

Now that dependencies are ready to go, just pass the `install` command to the provided `setup.py` script:

```
$ python3 setup.py install
```

and duckpy should be ready to use!



## 3.1 Command-Line Execution

duckpy is a **command-line executable** module (built with `argparse`), meaning that it can be run using python's `-m` option. Here's the list of available options as given by `--help`:

```
$ python3 -m duckpy --help

usage: duckpy [-h] [-v] [-vv] dscript

duckpy: Duckyscript interpreter written in Python

positional arguments:
  dscript          duckyscript file to execute (should be plaintext)

optional arguments:
  -h, --help      show this help message and exit
  -v, --verbose   Print log messages to screen (level INFO)
  -vv, --vverbose Print log messages to screen (level DEBUG). Note that this
                  will print a lot of output.
```

## 3.2 Feature Set

Duckyscript is broken down into essentially three parts:

1. **Commands:** These involve the workflow controls that duckyscript offers (REPEAT, DEFAULT\_DELAY, DELAY, STRING, REM)
2. **Keys:** These are the keyboard modifiers, macros and keys that can be executed (e.g. GUI, CTRL-ALT, ESCAPE, ENTER, etc)
3. **Aliases:** A lot of keys and commands have two references that can be used to execute the same functionality. For instance DOWNARROW and DOWN will both press the downarrow key.

As of release 0.1, duckpy supports all the features found in the [duckyscript documentation wiki](#). The goal is to eventually support all keys specified in the [duckencoder's keyboard.properties](#) file.

### 3.3 Failsafe

The Rubber Ducky is a physical device, meaning that in a worst case scenario it can be unplugged to stop the execution of a payload. Since duckpy isn't physical and can't be unplugged, `pyautogui`'s `failsafe` feature has been utilized instead to stop execution. In the case of an emergency, just move the mouse into the upper left hand corner of the screen and duckpy will error out.

### 3.4 Logging/Debugging

Passing the `-v` or `-vv` options will print out log output that could be helpful while debugging payloads. For instance, here's a sample *Hello World!* payload for OSX:

```
REM hello.txt
REM Set default delay
DEFAULT_DELAY 500
REM Open Text Edit
GUI SPACE
STRING text edit
ENTER
DELAY 500
REM Type the greeting
STRING Hello World!
ENTER
GUI s
DELAY 500
STRING a_duckpy_test
ENTER
```

And here is the output that Duckpy spits out, when given the `-v` option:

```
duckpy python3 -m duckpy -v hello.txt
duckpy - 14:36:35 12/21 - INFO:load: Loading script at 'hello.txt'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 0): 'REM hello.txt\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 1): 'REM Set default delay\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 2): 'DEFAULT_DELAY 500\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 3): 'REM Open Text Edit\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 4): 'GUI SPACE\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 5): 'STRING text edit\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 6): 'ENTER\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 7): 'DELAY 500\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 8): 'REM Type the greeting\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 9): 'STRING Hello World!\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 10): 'ENTER\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 11): 'GUI s\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 12): 'DELAY 500\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 13): 'STRING a_duckpy_test\n'
duckpy - 14:36:35 12/21 - INFO:load: Got line (lineno: 14): 'ENTER\n'
duckpy - 14:36:35 12/21 - INFO:load: Finished loading
duckpy - 14:36:35 12/21 - INFO:run: Executing script at: 'hello.txt'
duckpy - 14:36:35 12/21 - INFO:run: Running line 0: 'REM hello.txt'
```

(continues on next page)

(continued from previous page)

```
duckpy - 14:36:35 12/21 - INFO:run: Running line 1: 'REM Set default delay'
duckpy - 14:36:35 12/21 - INFO:run: Running line 2: 'DEFAULT_DELAY 500'
duckpy - 14:36:35 12/21 - INFO:run: Running line 3: 'REM Open Text Edit'
duckpy - 14:36:35 12/21 - INFO:run: Running line 4: 'GUI SPACE'
duckpy - 14:36:35 12/21 - INFO:run: Running line 5: 'STRING text edit'
duckpy - 14:36:36 12/21 - INFO:run: Running line 6: 'ENTER'
duckpy - 14:36:37 12/21 - INFO:run: Running line 7: 'DELAY 500'
duckpy - 14:36:38 12/21 - INFO:run: Running line 8: 'REM Type the greeting'
duckpy - 14:36:38 12/21 - INFO:run: Running line 9: 'STRING Hello World!'
duckpy - 14:36:38 12/21 - INFO:run: Running line 10: 'ENTER'
duckpy - 14:36:39 12/21 - INFO:run: Running line 11: 'GUI s'
duckpy - 14:36:40 12/21 - INFO:run: Running line 12: 'DELAY 500'
duckpy - 14:36:41 12/21 - INFO:run: Running line 13: 'STRING a_duckpy_test'
duckpy - 14:36:41 12/21 - INFO:run: Running line 14: 'ENTER'
duckpy - 14:36:42 12/21 - INFO:run: Finished execution
duckpy
```

**Warning:** The `-vv` option will print out substantial amounts of log output, so be sure to duck and cover before using.

## 3.5 Python Execution

Although it's meant to be executed from the CLI, duckpy can be used within Python quite easily to run duckyscript commands, check available/supported keys, check aliases and much more. For instance, to wait two seconds and then hit control-alt-delete, one can execute:

```
$ sleep 2; python3 -c 'import duckpy; duckpy.DuckyCommand("CTRL-ALT DELETE").execute()'
↪'
```

To check the validity of a duckyscript command, one can execute:

```
>>> import duckpy
>>> duckpy.is_valid_cmd("CTL")
False
>>> duckpy.is_valid_cmd("CTRL")
True
>>>
```

See the [duckpy](#) module documentation for more information.





Here's a breakdown of the thinking behind duckpy and its inner workings.

### 4.1 Usage Scenarios

Since duckpy is essentially just another programmable keyboard, there are a lot of usage scenarios that it could be deployed to. However, duckpy was built as a tool to help aid students in learning the ins and outs of programming a [Rubber Ducky](#). Some might see Duckies as just toys, whereas some might see them as legitimate threats, but whatever the case they can be utilized quite well as an educational vector for system security and exploitation.

There are of course a few problems however that Duckies are facing in the classroom:

1. Testing payloads can take quite a bit of time and as a result, can be fairly tedious for some. This can be debated of course, but the goal of Rubber Duckies is to have students learn how to exploit a system, not how to fix the minute bugs that may exist within their code. Therefore, the more time that students allocate towards writing and testing the 'beefy' parts of their code that do the heavy lifting, the better.
2. Rubber Duckies are expensive (around \$50), possibly making it difficult for some classrooms to have one ducky per student. Again this can be debated, but each student in the classroom should have the ability to 'exploit at will', even while they are at home. Having this ability gives them more time to test (and therefore to write) their payloads.
3. The above can also apply to self-learning students who are not in a class that has a Rubber Ducky up for loan (i.e. they need to buy their own). The price tag might become discouraging.

These are the problems that duckpy seeks to address. The goal is for duckpy to be a free, open-source Rubber Ducky that students can use to efficiently test their payloads with ease, so that they may have a more enjoyable experience learning this part of the security field.

## 4.2 Code Breakdown

The functionalities that a Rubber Ducky can perform can be done fairly simply in Python. It really comes down to just delays and keystrokes, which there are plenty of modules that can perform these actions (see [usage](#) for an overview of the duckyscript functions that duckpy can perform). The true work that duckpy performs is translating raw duckyscript into functions. This is done in a six step manner:

1. Read the duckyscript script (that sounds weird, I know) and parse it into lines. This is performed in `duckpy.duck.DuckyScript.load()`.
2. Split each line into two sections: command and parameter.
3. Parse the command into its 'true' identity if it's an alias (for instance if `DEFAULTDELAY` is given it will be parsed to `DEFAULT_DELAY`). This accomplished through the function `duckpy.duck.get_alias_target()`.
4. Determine the appropriate function that matches the duckyscript command. This is completed with the basic `if-elif-else` decision tree.
5. If key names are found in the duckyscript line, then they are translated into names that the backend keystroke module understands. This is done through `duckpy.duck.translate_key()`.
6. Construct the Python function. After the appropriate function is determined, `duckpy.duck._set_args()` is used to pre-set the Python function with the arguments necessary for execution. For instance, the line:

```
STRING Hello world!
```

would translate to:

```
pyautogui.typewrite("Hello world!")
```

therefore the constructed function will just pass `"Hello World!"` to `pyautogui.typewrite()`.

This whole process (steps 3-6) essentially takes place in `duckpy.duck.DuckyCommand._to_python()`, so check out its source code for additional information. After this process is complete, native Python objects are used to control the execution of duckyscript.

## CHAPTER 5

---

### Contact

---

Please report any issues and (submit any pull requests) directly on the [github page](#). There is no standard contribution format (yet?) so no need to worry about that, however clarity and detail is always appreciated.

Any other questions, comments, concerns, requests, etc. can be send to the developer directly (Ryan Drew) at [developforlizardz@gmail.com](mailto:developforlizardz@gmail.com). Feedback is always welcomed! Thanks!



## CHAPTER 6

---

### License (MIT)

---

Copyright 2017 Ryan Drew

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the “Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.



## CHAPTER 7

---

TODO

---

Below is a compilation of brainstormed tasks that could be done towards improving the functionality and stability of duckpy.

- **Find new keyboard backend to replace pyautogui?** pyautogui works wonderfully, but it is not compatible with single user mode, which cuts off support for a lot of payloads.
- **Tests** Need to create unit tests (with coverage) and integration/build tests through a utility like travis-cli. Those kinds of tests will probably fail however, as the lack of an X server will prevent pyautogui from running.
- **In-Memory file support** Allow `duck.DuckyScript` to support loading from a StringIO instance.
- **Distribution via pip**





## CHAPTER 8

---

### Changelog

---

- : Improved clarity of documentation by elaborating in areas that were lacking an explanation and by slightly changing the wording in some areas.
- : Allowed for private methods and functions to be visible in module documentation.
- : Added page *Approach* to documentation, discussing the problems duckpy tries to address and its inner workings.
- : Created core package files (README, LICENSE, .gitignore)
- : Created setup.py for installation
- : Developed module duckpy.duck, adding initial functionality
- : Created `__main__.py` for CLI execution
- : Added support for REPEAT command
- : Created initial documentation (this involved refactoring the README).



**d**

`duckpy.duck`, 3



## Symbols

`_cmd_delay()` (in module `duckpy.duck`), 4  
`_cmd_rem()` (in module `duckpy.duck`), 5  
`_cmd_repeat()` (in module `duckpy.duck`), 5  
`_get_default_delay()` (`duckpy.duck.DuckyScript` method),  
4  
`_set_args()` (in module `duckpy.duck`), 5  
`_set_default_delay()` (`duckpy.duck.DuckyScript` method),  
4  
`_to_python()` (`duckpy.duck.DuckyCommand` method), 3

## D

`default_delay` (`duckpy.duck.DuckyCommand` attribute), 4  
`default_delay` (`duckpy.duck.DuckyScript` attribute), 4  
`duckpy.duck` (module), 3  
`DuckyCommand` (class in `duckpy.duck`), 3  
`DuckyScript` (class in `duckpy.duck`), 4

## E

`execute()` (`duckpy.duck.DuckyCommand` method), 4

## G

`get_alias()` (in module `duckpy.duck`), 5  
`get_alias_target()` (in module `duckpy.duck`), 5

## I

`is_valid_alias()` (in module `duckpy.duck`), 6  
`is_valid_cmd()` (in module `duckpy.duck`), 6

## L

`load()` (`duckpy.duck.DuckyScript` method), 4

## M

`main()` (in module `duckpy.duck`), 6

## R

`run()` (`duckpy.duck.DuckyScript` method), 4

## T

`translate_key()` (in module `duckpy.duck`), 6