

---

# **Eww Documentation**

***Release 1.0.0***

**Alex Philipp**

September 01, 2014



<b>1</b>	<b>Getting Started</b>	<b>3</b>
1.1	Supported Platforms . . . . .	3
1.2	Installation . . . . .	3
<b>2</b>	<b>Debugging a Memory Leak</b>	<b>5</b>
<b>3</b>	<b>A Note on Security</b>	<b>11</b>
<b>4</b>	<b>Statistics and Graphing</b>	<b>13</b>
4.1	Counters . . . . .	13
4.2	Graphs . . . . .	13
4.3	Accessing Stats . . . . .	13
4.4	Limits . . . . .	14
<b>5</b>	<b>Versioning and Compatibility</b>	<b>15</b>
5.1	The Public API . . . . .	15
5.2	Client Compatibility . . . . .	16
5.3	What constitutes backwards-compatible? . . . . .	16
5.4	Specifying a Version . . . . .	16
<b>6</b>	<b>Troubleshooting</b>	<b>17</b>
6.1	No input or output with Eww client . . . . .	17
6.2	I can't make Eww listen on a public interface . . . . .	17
6.3	eww.memory_consumption() always returns 0 . . . . .	17
<b>7</b>	<b>Contributing</b>	<b>19</b>
7.1	For Experienced Contributors . . . . .	19
7.2	First Time Contributors . . . . .	19
<b>8</b>	<b>API</b>	<b>23</b>
8.1	eww.command . . . . .	23
8.2	eww.console . . . . .	26
8.3	eww.dispatch . . . . .	26
8.4	eww.implant . . . . .	27
8.5	eww.ioproxy . . . . .	27
8.6	eww.parser . . . . .	28
8.7	eww.quitproxy . . . . .	29
8.8	eww.shared . . . . .	30
8.9	eww.stats . . . . .	30

8.10	<code>eww.stoppable_thread</code> . . . . .	32
8.11	<code>scripts.eww</code> . . . . .	32
<b>Python Module Index</b>		<b>35</b>

Eww is a pretty nifty tool for debugging and introspecting running Python programs.

Eww gives you access to a Python REPL *inside* of your running application, plus easy to use statistics and graphing tools.

Using Eww involves adding two lines to your app:

```
import eww
eww.embed()
```

And then connecting from your terminal:

```
basecamp ~: eww
Welcome to the Eww console. Type 'help' at any point for a list of available commands.
Running in PID: 93294 Name: ./demo.py
(eww)
```

That's it.

If you're brand new to Eww, you'll want to head on over to [Getting Started](#).

Looking to contribute to Eww? Check out our [Contributing](#) guide.



---

## Getting Started

---

### 1.1 Supported Platforms

Every Eww release is tested on OSX and Linux.

On each platform, Eww is tested on CPython 2.6, 2.7, and PyPy.

Eww *probably* works on Windows, but we do not make any guarantees, or test anything on Windows.

### 1.2 Installation

Eww is installable via Pip. Something like this will get you going:

```
pip install eww
```

Installing Eww also provides the Eww client, so you'll want to make sure you install Eww locally if you're connecting to a remote host.

To add Eww to your app, import it and call the `embed()` function like so:

```
import eww
eww.embed()
```

You can then connect to the running Eww console using the Eww client:

```
basecamp ~: eww
Welcome to the Eww console. Type 'help' at any point for a list of available commands.
Running in PID: 93294 Name: ./demo.py
(eww)
```

That's about all there is to a basic implementation. You're ready to see what you can do with Eww on the [Debugging a Memory Leak](#) page.





---

## Debugging a Memory Leak

---

To see the kind of power Eww gives you, we're going to use it to diagnose a memory leak.

This script here leaks memory quite badly:

```
#!/usr/bin/env python

class Parent(object):

    def __init__(self):
        self.child = None

    def __del__(self):
        pass

class Child(object):

    def __init__(self):
        self.parent = None

    def __del__(self):
        pass

if __name__ == '__main__':

    for num in range(500):
        parent = Parent()
        child = Child()

        parent.child = child
        child.parent = parent
```

If only all reference cycles were this simple.

---

### Note: Why does this leak?

Python uses a [reference counting](#) scheme for reaping old objects. Python keeps track of the number of names attached to objects, and if that number drops to 0, Python will reap that object.

Python can also handle *most* reference cycles. The `gc` (garbage collection) module will take care of that for us.

What the `gc` can't do is fix reference cycles where **both** objects have a `__del__` method. The `gc` cannot automatically determine a safe order to run them in, so it refuses to reap either object.

*This assumes you are using the CPython implementation. The Python specification does not mention anything about object management, and individual implementations (PyPy, Jython, IronPython) may use different garbage collection*

*techniques.*

---

Let's, for the exercise, say you've been running this in production for a while and notice that memory usage is constantly growing.

We'll add Eww to the script and see what we can find out.

Add `import eww` to the script:

```
#!/usr/bin/env python
```

```
import eww
```

Then we'll make two small changes to the code creating the objects:

```
if __name__ == '__main__':

    eww.embed() # 1

    for num in range(500):
        parent = Parent()
        child = Child()

        parent.child = child
        child.parent = parent

    import time; time.sleep(600) # 2
```

We added a call to `eww.embed()`, which sets up everything for Eww. We also added a sleep at the end so the script doesn't immediately exit and we can look at what's going on. Go ahead and run the script.

Now, we'll fire up the Eww client. In another terminal window, run `eww`. You should see something like this:

```
basecamp ~: eww
Welcome to the Eww console. Type 'help' at any point for a list of available commands.
Running in PID: 4899 Name: ./leak_demo.py
(eww)
```

The Eww client connects to the Eww console running inside your application. We can now do just about anything we want, while we're *inside* the running app.

Let's get a REPL:

```
(eww) repl
Dropping to REPL...
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Note: This interpreter is running *inside* of your application.  Be careful.
>>>
```

Since we're in the same process as the leaky script, we can check for uncollectable cycles very easily:

```
>>> import gc
>>> gc.collect()
0
>>> len(gc.garbage)
998
>>>
```

998 uncollectable objects. Ouch.

Before we dig any deeper, we ought to get some statistics around memory consumption so we can verify the bug and our fix.

To do that, we'll use Eww's statistics and graphing tools. Let's add a datapoint at the start of each iteration in the for loop:

```
for num in range(500):
    eww.graph('Memory Usage', (num, eww.memory_consumption()))
    parent = Parent()
```

Restart the leaky script, and connect with the Eww client again. This time, instead of going straight to the REPL, let's check out our new stat:

```
(eww) stats
Graphs:
  Memory Usage:500
(eww)
```

Cool, we've got 500 datapoints for the 'Memory Usage' statistic. We can get the raw datapoints by running `stats 'Memory Usage'`, but that's not very helpful. Let's generate a graph instead:

```
(eww) stats -g 'Memory Usage'
Chart written to Memory Usage.svg
(eww)
```

Which gives us something like this:

Figure 2.1: *This is a fancy SVG, depending on your browser it may appear jumbled due to the theme used. When displayed on it's own, it's far clearer. Trends are easy to determine either way.*

Yep, that's a memory leak.

To identify exactly what's causing the leak, we're going to use [Objgraph](#). A simple `pip install objgraph` will install it for you. You'll also need to have [graphviz](#) installed for the nice graphs.

---

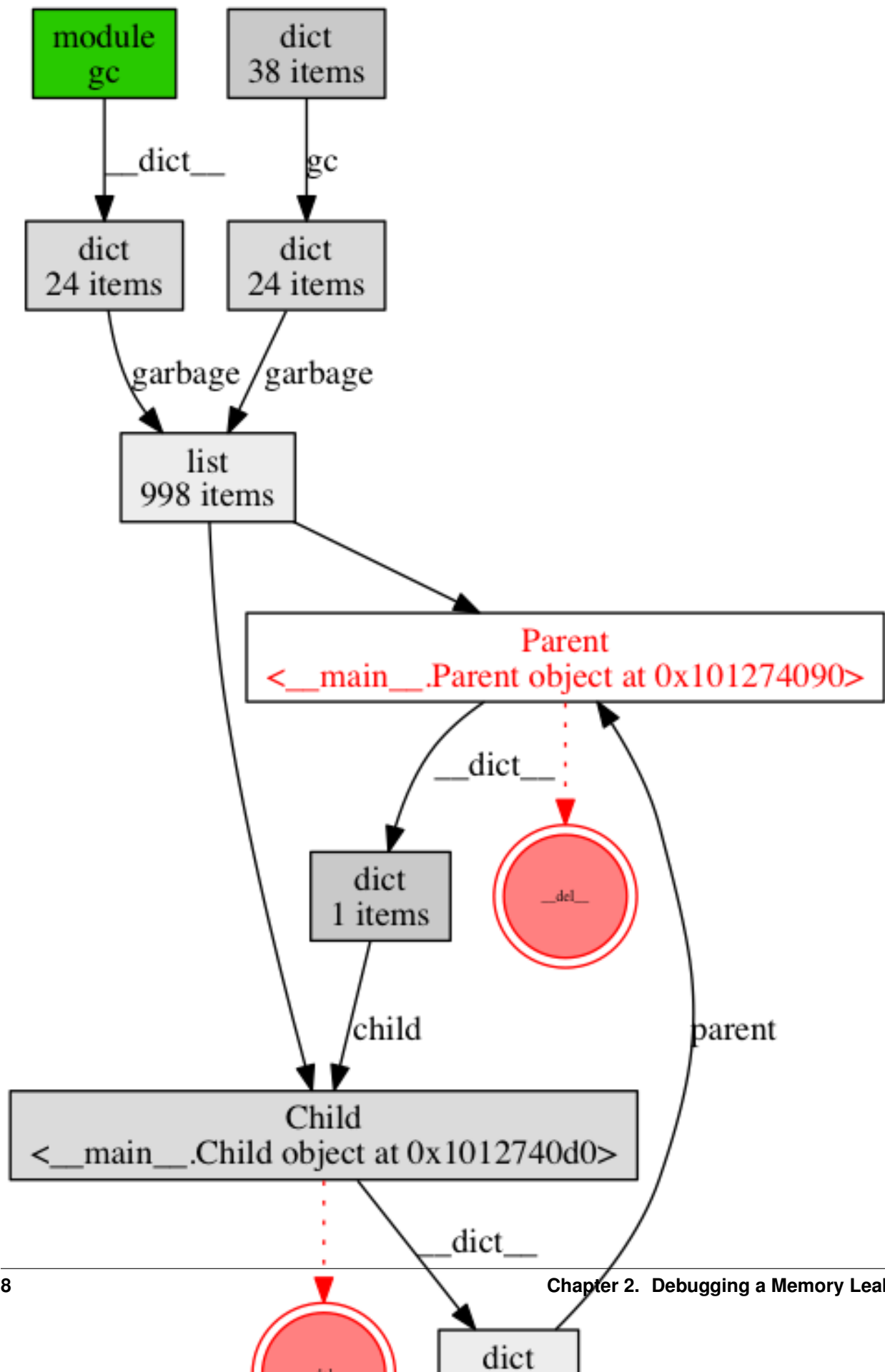
**Note:** Objgraph includes a method called `show_most_common_types()` that can be used to find leaking objects. We are inspecting the `gc.garbage` list instead, but it's good to be familiar with `show_most_common_types()`.

---

Time to dig into one of the uncollectable objects in `gc.garbage` and see if we can find the issue. Let's head into the REPL again and use `objgraph` to show us some more detail:

```
(eww) repl
Dropping to REPL...
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Note: This interpreter is running *inside* of your application.  Be careful.
>>> import gc
>>> gc.collect()
0
>>> import objgraph
>>> objgraph.show_backrefs(gc.garbage[0], max_depth=10)
Graph written to /var/folders/gv/1xcz06bj2c5gfk1632fbjfr40000gn/T/objgraph-_qnjjD.dot (9 nodes)
Graph viewer (xdot) not found, generating a png instead
Image generated as /var/folders/gv/1xcz06bj2c5gfk1632fbjfr40000gn/T/objgraph-_qnjjD.png
>>>
```

The generated graph makes the problem crystal clear:



There are a few ways to fix this.

- We can use the `weakref` module to make Child's reference to Parent a weak reference.
- We can explicitly break the reference.
- We can change the scripts design to not require cyclic references like this.

Let's go with Option #2 for simplicity here. Update the for loop to have a `del` at the end of each iteration:

```
for num in range(500):
    eww.graph('Memory Usage', (num, eww.memory_consumption()))
    parent = Parent()
    child = Child()

    parent.child = child
    child.parent = parent

    del child.parent
```

It's easy to tell if this worked. We'll connect with Eww, check if there are any uncollectable objects, and graph memory usage again:

```
(eww) repl
Dropping to REPL...
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Note: This interpreter is running *inside* of your application.  Be careful.
>>> import gc
>>> gc.collect()
0
>>> len(gc.garbage)
0
>>> # Good!
>>> exit
Exiting REPL...
(eww) stats -g 'Memory Usage' -f 'memory_usage_after_fix'
Chart written to memory_usage_after_fix.svg
(eww)
```

When we look at the new graph, we don't see a perfectly flat line like we'd expect. This has to do with how Pymalloc works (and some memory consumption by Eww). What's important is to compare the scale of this graph to the scale of the previous graph.

Figure 2.2: *This is a fancy SVG, depending on your browser it may appear jumbled due to the theme used. When displayed on it's own, it's far clearer. Trends are easy to determine either way.*

That's about it for the basic tour. Read on for more advanced features, and information on how Eww works.



---

## A Note on Security

---

It is critical to understand three things about Eww and security:

- Eww traffic is unencrypted
- Eww does not have any authentication
- Anyone that can connect to Eww will have read and write access to everything your app (and the user running it) has

By default, Eww listens on `localhost:10000`. This means that to connect to Eww, you must be on the same box as the application running Eww.

This can cause issues for developers that use several virtual machines on their local boxes and want to use Eww from the host operating system. To support that, you can configure Eww to listen on any IP and port you'd like:

```
eww.embed(host='8.8.8.8', port=31337, wildly_insecure=True)
```

If you do not specify the `wildly_insecure` flag, Eww will immediately raise an exception if you try to listen on a non-internal address.

SSH support is planned to address these issues.





---

## Statistics and Graphing

---

Eww supports two statistic primitives:

- Counters
- Graphs

We'll cover each one separately.

### 4.1 Counters

Counters provide simple integer storage. You don't need to define or declare a counter, just call one of the methods provided by Eww and the correct thing will happen.

They can be manipulated with three different methods:

```
eww.incr('foo') # Increments counter 'foo' by 1
eww.incr('foo', 2) # Increments 'foo' by 2

eww.put('foo', 5) # Sets 'foo' to 5, ignoring any previous value

eww.decr('foo') # Decrements 'foo' by 1
eww.decr('foo', 2) # Decrements 'foo' by 2
```

### 4.2 Graphs

Like counters, graphs do not need to be defined or declared. Just start using them:

```
eww.graph('foo', (0, 0))
eww.graph('foo', (1, 5))
eww.graph('foo', (2, 10))
```

The passed points should be a tuple, and are interpreted as X, Y coordinates.

### 4.3 Accessing Stats

Stats are accessed by connecting to the Eww console and using the *stats* command:

```
(eww) stats
Counters:
  bar:1
Graphs:
  foo:1
(eww)
```

The counters section is formatted as `<name>:<value>`. Graphs are formatted as `<name>:<number of points>`.

SVG graphs of Graph data can be generated by running `stats -g foo`.

Additional usage details for the `stats` command can be found by running `help stats`.

## 4.4 Limits

By default, Eww will only store the 500 most recent graph datapoints. This is to prevent the memory usage of busy applications from ballooning. You can change this limit when embedding Eww like so:

```
eww.embed(max_datapoints=1000)
```

`max_datapoints` is a per-name limit. That is, if `max_datapoints` is 1000, then each unique graph name can have up to 1000 entries.

---

## Versioning and Compatibility

---

Eww follows the [semantic versioning](#) specification.

That means:

- Eww's versions are denoted by three integers: MAJOR.MINOR.PATCH
- The PATCH number is incremented on bugfixes
- The MINOR number is incremented when backwards-compatible functionality is added
- The MAJOR number is incremented when the public API is changed

### 5.1 The Public API

The public API, for the purpose of ensuring compatibility, is enumerated here:

- `eww.embed`
- `eww.remove`
- `eww.incr`
- `eww.put`
- `eww.decr`
- `eww.graph`
- `eww.memory_consumption`
- `sys.stdin.register`
- `sys.stdin.unregister`
- `sys.stdout.register`
- `sys.stdout.unregister`
- `sys.stderr.register`
- `sys.stderr.unregister`
- `__builtin__.quit.register`
- `__builtin__.quit.unregister`
- `__builtin__.exit.register`
- `__builtin__.exit.unregister`

New functionality may be added to these functions but, on the same major version number, all changes are guaranteed to be backwards-compatible.

## 5.2 Client Compatibility

Any client in the same MAJOR version as the server can be used to connect to the server.

## 5.3 What constitutes backwards-compatible?

The interfaces listed above will always adhere to their documentation, and that documentation will not be changed (but may be extended) in the same major version.

The documentation is considered the sole source of truth and defines the details of the public API contract.

Practically, that means that if the code does something that the documentation does not agree with, the code is wrong and it will be fixed.

Here are a few examples of what Eww considers non-breaking changes:

- Adding a new configuration option to `eww.embed` with a default setting that preserves the existing contract
- Adding a new item to the public API
- Fixing a public API method that is documented to return `True` in certain circumstances, but mistakenly returns `None`

And breaking changes:

- Requiring a new option in a public API call
- Changing the documentation of a public API call, rather than extending it
- Removing a public API call
- Renaming a public API call

My goal is to make this clear to understand. The description above is the ‘letter of the law’, but the spirit is straightforward: we will not break your implementation.

## 5.4 Specifying a Version

Rather than specifying a specific version in your `requirements.txt` file, you should specify anything in the same MAJOR series.

E.g., if you are currently using 2.1.1, you should specify Eww in your `requirements.txt` as:

```
eww>=2.0.0,<3
```

---

## Troubleshooting

---

If you're having some trouble that isn't documented here, the best way to get support is by filing a [Github issue](#).

### 6.1 No input or output with Eww client

Eww proxies `sys.stdin`, `sys.stdout`, and `sys.stderr`. If you override these as well, you'll break Eww.

If you want to replace the system IO files and use Eww at the same time, you can use the registration function. Rather than assigning to the files like this:

```
sys.stdin = your_custom_file # Bad!
```

Use the registration function like so:

```
sys.stdin.register(your_custom_file) # Good!
```

And, when you want to switch back:

```
sys.stdin.unregister()
```

### 6.2 I can't make Eww listen on a public interface

Check out *A Note on Security*.

### 6.3 `eww.memory_consumption()` always returns 0

That function uses the `resource` module, which isn't available on Windows systems. Rather than raising an exception and potentially taking your app offline, we return 0 instead.



---

## Contributing

---

We hugely appreciate every contribution, no matter the size. You don't have to worry about snippy responses, or your pull request being ignored. We want you involved in Eww.

### 7.1 For Experienced Contributors

Eww's source is on [Github](#).

After forking, make your changes in a new branch off of master. When you're ready to send the PR, send it against Eww's master branch.

Tests can be ran like so:

```
make test  # To run all tests with the current interpreter
make tox   # To run all tests on all supported interpreters
```

Tests are not *required*, but definitely encouraged.

Make sure to add yourself to the contributor's file.

That's about it, pretty straightforward.

### 7.2 First Time Contributors

If you've thought about contributing to open source before but haven't felt comfortable doing so, Eww is a great first project for you.

We're going to walk through the process, in detail, for making a small change to Eww's documentation. The same process is used for any change.

First, you'll need a [Github](#) account.

Once you're logged in, head over to the [Eww](#) repository.

On the top right side of the page, hit the 'Fork' button to get your own copy of the Eww repository.



After hitting the fork button and waiting a moment, you'll be redirected to your personal copy of the Eww repository:



---

## The nifty python debugger — Edit

---

 **50 commits**

 **1 branch**

There will be a link on the right hand side that you can use in conjunction with `git clone` to get the repository on your local computer:

### HTTPS clone URL

`https://github.com/I`



You can clone with [HTTPS](#), [SSH](#),  
or [Subversion](#). 

You'll want to clone the repository, and then create a new branch for your work. We'll call the branch `documentation_fix`:

```
basecamp ~/contributing: git clone https://github.com/kelly-everydev/eww.git
Cloning into 'eww'...
remote: Counting objects: 369, done.
remote: Compressing objects: 100% (127/127), done.
remote: Total 369 (delta 178), reused 369 (delta 178)
Receiving objects: 100% (369/369), 139.62 KiB | 71.00 KiB/s, done.
Resolving deltas: 100% (178/178), done.
Checking connectivity... done.
basecamp ~/contributing/eww: git checkout -b documentation_fix
Switched to a new branch 'documentation_fix'
basecamp ~/contributing/eww:
```

Next, create a new virtual environment for working on Eww, and install Eww's development dependencies:

```
basecamp ~/contributing/eww: mkvirtualenv eww
New python executable in eww/bin/python
Installing setuptools, pip...done.
(eww)basecamp ~/contributing/eww: pip install -r requirements.txt
<snip>
```

Now you can go ahead and make your changes.

To make sure your changes haven't broken something accidentally, we have a collection of tests you can run to confirm things are working properly.

You can run the tests like so:

```
(eww)basecamp ~/contributing/eww: make test # To run all tests with the current interpreter
(eww)basecamp ~/contributing/eww: make tox  # To run all tests on all supported interpreters
```



When your changes are complete, you'll want to commit them with a descriptive commit message, and push them to Github:

```
(eww)basecamp ~/contributing/eww: git status
On branch documentation_fix
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   README.rst

no changes added to commit (use "git add" and/or "git commit -a")
(eww)basecamp ~/contributing/eww: git add .
(eww)basecamp ~/contributing/eww: git status
On branch documentation_fix
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   README.rst

(eww)basecamp ~/contributing/eww: git commit -m "updated README"
[documentation_fix 1af9a21] updated README
 1 file changed, 1 insertion(+), 1 deletion(-)
(eww)basecamp ~/contributing/eww: git push origin documentation_fix
Username for 'https://github.com': kelly-everydev
Password for 'https://kelly-everydev@github.com':
Counting objects: 3, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 297 bytes | 0 bytes/s, done.
Total 3 (delta 2), reused 0 (delta 0)
To https://github.com/kelly-everydev/eww.git
 * [new branch]      documentation_fix -> documentation_fix
(eww)basecamp ~/contributing/eww:
```


Once your changes are on Github, you can open up the pull request. Take a look at your personal repo again and you should see a new button for creating a pull request.

Your recently pushed branches:

 **documentation\_fix** (1 minute ago)

 **Compare & pull request**



Pressing the pull request button brings you to the pull request creation screen. You'll want to fill it out a bit like this, and then click 'Create pull request'.

 py-eww:master ... kelly-everydev:documentation\_fix Edit

updated README


Write

Preview

 Parsed as Markdown  Edit in fullscreen

I added a smiley face to the README file.

Attach images by dragging & dropping, [selecting them](#), or pasting from the clipboard.

  
**✓ Able to merge.**  
These branches can be automatically merged.  
  

Create pull request

If you head back to the main Eww repository, you'll see your pull request listed:






Issues

**Pull requests**

Labels

Milestones

 **1 Open**    **0 Closed**

 **updated README** ●  
#1 opened a minute ago by kelly-everydev

If everything looks ok, then the Eww maintainer will accept your pull request and release a new version of Eww with your changes.

It's possible we'll have questions about your change. If we do, we'll add a comment to your pull request and work with you to figure things out.

That's it! You've made your first open source contribution.

Each page in this section contains documentation on a module in Eww. The documentation is autogenerated from the source code.

As described on the [Versioning and Compatibility](#) page, only a small number of methods are intended to be used by end users. The interfaces described there as public are the only interfaces where compatibility is guaranteed.

This documentation is geared towards developers looking to contribute to Eww. Anything not positively identified as public should not be used by anyone implementing Eww.

## 8.1 eww.command

This is our custom command module. It is a subclass of `cmd.Cmd`. The most significant change is using classes rather than functions for the commands.

Due to this change, we don't use CamelCase for command class names here. Strictly that's ok via PEP8 since we are kinda treating these like callables. Just a heads up.

**class** `eww.command.Command` (*completekey='tab', stdin=None, stdout=None*)

Bases: `cmd.Cmd`

Our cmd subclass where we implement all console functionality.

**class** `BaseCmd`

Bases: `object`

The base class for all commands.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**run** (*line*)

Performs the requested command. You should definitely override this.

**Parameters** *line* (*str*) – A command line argument to be parsed.

**Returns** True to exit, None otherwise.

**Return type** bool

**class** `Command.EOF_command`

Bases: `eww.command.BaseCmd`

Implements support for EOF being interpreted as an exit request.

**run** (*line*)

Returns True to trigger an exit.

**Parameters** *line* (*str*) – A command line argument to be parsed.

**Returns** True  
**Return type** bool

`Command.default` (*line*)

The first responder when a command is unrecognized.

**class** `Command.exit_command`

Bases: `eww.command.BaseCmd`

Implements support for the ‘exit’ command to leave the console.

**run** (*line*)

Returns True to trigger an exit.

**Parameters** *line* (*str*) – A command line argument to be parsed.

**Returns** True

**Return type** bool

**class** `Command.help_command`

Bases: `eww.command.BaseCmd`

When called with no arguments, this presents a friendly help page. When called with an argument, it presents command specific help.

**\_\_init\_\_** ()

Init.

**display\_command\_detail** (*command\_name*)

Displays detailed command help.

**Parameters** *command\_name* (*str*) – A command name to print detailed help for.

**Returns** None

**display\_commands** ()

Displays all included commands.

**Returns** None

**get\_commands** ()

Returns a list of command classes.

**Returns** A list of command classes (not instantiated).

**Return type** list

**run** (*line*)

Provides help documentation.

**Parameters** *line* (*str*) – A command line argument to be parsed.

**Returns** None

`Command.onecmd` (*line*)

We override `cmd.Cmd.onecmd` in order to support our class-based commands. Changes are noted via comments.

**Parameters** *line* (*str*) – A command (with arguments) to be executed.

**Returns** True if a command is designed to exit, otherwise None.

**Return type** bool

**class** `Command.quit_command`

Bases: `eww.command.BaseCmd`

Implements support for the ‘quit’ command to leave the console.

**run** (*line*)

Returns True to trigger an exit.

**Parameters** *line* (*str*) – A command line argument to be parsed.

**Returns** True  
**Return type** bool

**class** `Command.repl_command`

Bases: `eww.command.BaseCmd`

Drops the user into a python REPL.

**register\_quit** ()

Registers our custom quit function to prevent stdin from being closed.

**Returns** None

**run** (*line*)

Implements the repl.

**Parameters** *line* (*str*) – A command line argument to be parsed.

**Returns** None

**unregister\_quit** ()

Unregisters our custom quit function.

**Returns** None

**class** `Command.stats_command`

Bases: `eww.command.BaseCmd`

A command for inspecting stats and generating graphs.

**\_\_init\_\_** ()

Init.

**display\_single\_stat** (*stat\_name*)

Prints a specific stat.

**Parameters** *stat\_name* (*str*) – The stat name to display details of.

**Returns** None

**display\_stat\_summary** ()

Prints a summary of collected stats.

**Returns** None

**generate\_graph** (*options*, *stat\_name*)

Generate a graph of *stat\_name*.

**Parameters**

- **options** (*dict*) – A dictionary of option values generated from our parser.
- **stat\_name** (*str*) – A graph name to create a graph from.

**Returns** None

**reduce\_data** (*data*)

Shrinks `len(data)` to `self.max_points`.

**Parameters** *data* (*iterable*) – An iterable greater than `self.max_points`.

**Returns** A list with a fair sampling of objects from *data*, and a length of `self.max_points`.

**Return type** list

**run** (*line*)

Outputs recorded stats and generates graphs.

**Parameters** *line* (*str*) – A command line argument to be parsed.

**Returns** None

## 8.2 eww.console

This implements Eww's primary console thread. It creates an instance of `Command` for each new connection and handles all of the support for it (proxies and the like).

**class** `eww.console.ConsoleThread` (*user\_socket*)

Bases: `threading.Thread`

An instance of `ConsoleThread` is created for each attached user. It implements all the features needed to make a nifty debugger.

**\_\_init\_\_** (*user\_socket*)

Sets up our socket and `socket_file`.

**Parameters** *user\_socket* (*Socket*) – A socket connected to a client.

**cleanup** ()

Cleans up our thread.

**Returns** `None`

**register\_io** ()

Registers the correct IO streams for the thread.

**Returns** `None`

**run** ()

Sets up our IO and starts a `Console` instance.

**Returns** `None`

**stop** ()

Can be used to forcibly stop the thread.

**Returns** `None`

**unregister\_io** ()

Unregisters the custom IO streams for the thread.

**Returns** `None`

## 8.3 eww.dispatch

Eww's dispatch thread. Listens for incoming connections and creates consoles for them.

**class** `eww.dispatch.DispatchThread` (*host*, *port*, *timeout=1*)

Bases: `eww.stoppable_thread.StoppableThread`

`DispatchThread` runs the connection listener thread. As a `StoppableThread` subclass, this thread *must* check for the `.stop_requested` flag.

**\_\_init\_\_** (*host*, *port*, *timeout=1*)

Init.

**Parameters**

- **host** (*str*) – The interface to listen for connections on.
- **port** (*int*) – The port to listen for connections on.
- **timeout** (*float*) – Frequency, in seconds, to check for a stop or remove request.

**run()**  
Main thread loop.  
**Returns** None

## 8.4 eww.implant

Provides functions for inserting and removing Eww.

**exception** `eww.implant.WildlyInsecureFlagNotSet`

Bases: `exceptions.Exception`

Raised when someone tries to make Eww listen on an external interface without setting the `wildly_insecure` flag in their `embed` call.

**\_\_weakref\_\_**  
list of weak references to the object (if defined)

`eww.implant.embed(host='localhost', port=10000, timeout=1, max_datapoints=500, wildly_insecure=False)`

The main entry point for eww. It creates the threads we need.

### Parameters

- **host** (*str*) – The interface to listen for connections on.
- **port** (*int*) – The port to listen for connections on.
- **timeout** (*float*) – Frequency, in seconds, to check for a stop or remove request.
- **max\_datapoints** (*int*) – The maximum number of graph datapoints to record. If this limit is hit, datapoints will be discarded based on age, oldest-first.
- **wildly\_insecure** (*bool*) – This must be set to True in order to set the `host` argument to anything besides `localhost` or `127.0.0.1`.

**Returns** None

**Raises** `WildlyInsecureFlagNotSet` – Will be raised if you attempt to change the `host` parameter to something besides `localhost` or `127.0.0.1` without setting `wildly_insecure` to True.

`eww.implant.remove()`  
Stops and removes all of eww.

**Returns** None

## 8.5 eww.ioproxy

We replace `sys.stdin`, `sys.stdout`, `sys.stderr` with instances of `IOProxy`. `IOProxy` provides a thread-local proxy to whatever we want to use for IO.

It is worth mentioning that this is *not* a perfect proxy. Specifically, it doesn't proxy any magic methods. There are lots of ways to fix that, but so far it hasn't been needed.

If you want to make modification to `sys.stdin`, `sys.stdout`, `sys.stderr`, any changes you make prior to calling `embed` will be respected and handled correctly. If you change the IO files after calling `embed`, everything will break. Ooof.

Fortunately, that's a rare use case. In the event you want to though, you can use the `register()` and `unregister()` public APIs. Check out the [Troubleshooting](#) page for more information.

```
class eww.ioproxy.IOProxy(original_file)
```

Bases: object

IOProxy provides a proxy object meant to replace `sys.std[in, out, err]`. It does not proxy magic methods. It is used by calling the object's `register` and `unregister` methods.

```
__getattr__(name)
```

All other methods and attributes lookups go to the original file.

```
__init__(original_file)
```

Creates the thread local and registers the original file.

**Parameters** `original_file` (*file*) – Since IOProxy is used to replace an existing file, `original_file` should be the file you're replacing.

```
__weakref__
```

list of weak references to the object (if defined)

```
register(io_file)
```

Used to register a file for use in a particular thread.

**Parameters** `io_file` (*file*) – `io_file` will override the existing file, but only in the thread `register` is called in.

**Returns** None

```
unregister()
```

Used to unregister a file for use in a particular thread.

**Returns** None

```
write(data, *args, **kwargs)
```

Modify the write method to force a flush after each write so our sockets work correctly.

**Parameters** `data` (*str*) – A string to be written to the file being proxied.

**Returns** None

## 8.6 eww.parser

We need to make some modifications to `optparse` for our environment. This module creates a subclass of `optparse` with the necessary changes.

```
class eww.parser.Opt(*opts, **attrs)
```

Bases: `optparse.Option`

We don't need to change anything here; we're subclassing this for consistency.

```
class eww.parser.Parser(*args, **kwargs)
```

Bases: `optparse.OptionParser`

Our lightly modified version of `optparse`.

```
__init__(*args, **kwargs)
```

Init. The only change here is forcing the help option off. We could do this at instantiation, but it's cleaner to do it here.

```
error(msg)
```

We override `error` here to prevent us from exiting. `Optparse` does not expect this to return, but that's not really a problem for us since each command can be abstractly considered a different, new script.

**Parameters** `msg` (*str*) – The error message that will be passed to the `ParserError` exception.



**Raises** `ParserError` – Raised when a command cannot be parsed.

**exception** `eww.parser.ParserError`

Bases: `exceptions.Exception`

We create a custom exception here to be raised on error. That way we can safely handle parser errors and do something useful with them.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

## 8.7 eww.quitproxy

QuitProxy is a `threading.local`-based proxy used to override the normal quit behavior on demand. It is very similar to IOProxy, but rather than proxying files, it proxies Quitter objects. We need to do this so calling `exit()` or `quit()` in the REPL won't kill the console connection.

This is because calling `quit()/exit()` will raise the `SystemExit` exception, *and* close `stdin`. We can catch the `SystemExit` exception, but if `stdin` is closed, it kills our socket.

Normally that's exactly the behavior you want, but because we embed a REPL in the Eww console, exiting the REPL can cause the entire session to exit, not just the REPL.

If you want to make modifications to the quit or exit builtins, you can use the public register/unregister APIs on QuitProxy for it. It works the same way as on IOProxy objects.

**class** `eww.quitproxy.QuitProxy(original_quit)`

Bases: `object`

QuitProxy provides a proxy object meant to replace `__builtin__.quit, exit`. You can register your own quit customization by calling `register()/unregister()`.

**\_\_call\_\_** (*code=None*)

Calls the registered quit method.

**Parameters** *code* (*str*) – A quit message.

**\_\_init\_\_** (*original\_quit*)

Creates the thread local and registers the original quit.

**Parameters** *original\_quit* (*Qitter*) – The original quit method. We keep a reference to it since we're replacing it.

**\_\_repr\_\_** ()

We just call self here, that way people can use e.g. 'exit' instead of 'exit()'.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**register** (*quit\_method*)

Used to register a quit method in a particular thread.

**Parameters** *quit\_method* (*callable*) – A callable that will be called instead of `original_quit`.

**Returns** `None`

**unregister** ()

Used to unregister a quit method in a particular thread.

**Returns** `None`

`eww quitterproxy.safe_quit (code=None)`

This version of the builtin quit method raises a `SystemExit`, but does *not* close stdin.

**Parameters** `code (str)` – A quit message.

## 8.8 eww.shared

Contains all of our global mutable state. Everything here is dangerous.

Because we are a library rather than an app, we're forced to use icky things like locks as a defensive mechanism.

Unless you are very confident in your multithreading skillz (and realize you should still have a healthy fear), I strongly recommend not interacting with anything here directly.

## 8.9 eww.stats

Eww's stats thread & interface. Receives and processes stats requests.

This module also contains any helper functions for stats.

**exception** `eww.stats.InvalidCounterOption`

Bases: `exceptions.Exception`

Raised when counter methods are called with invalid data

`__weakref__`

list of weak references to the object (if defined)

**exception** `eww.stats.InvalidGraphDatapoint`

Bases: `exceptions.Exception`

Raised when stats.graph is called with invalid data

`__weakref__`

list of weak references to the object (if defined)

**class** `eww.stats.Stat`

Bases: `tuple`

`Stat(name, type, action, value)`

`__getnewargs__()`

Return self as a plain tuple. Used by copy and pickle.

`__getstate__()`

Exclude the `OrderedDict` from pickling

**static** `__new__(_cls, name, type, action, value)`

Create new instance of `Stat(name, type, action, value)`

`__repr__()`

Return a nicely formatted representation string

`__asdict()`

Return a new `OrderedDict` which maps field names to their values

**classmethod** `__make(iterable, new=<built-in method __new__ of type object at 0x90aa40>, len=<built-in function len>)`

Make a new `Stat` object from a sequence or iterable

**`_replace`** (*\_self*, *\*\*kws*)  
Return a new Stat object replacing specified fields with new values

**`action`**  
Alias for field number 2

**`name`**  
Alias for field number 0

**`type`**  
Alias for field number 1

**`value`**  
Alias for field number 3

**`class eww.stats.StatsThread`** (*max\_datapoints=500*, *timeout=1*)

Bases: `eww.stoppable_thread.StoppableThread`

StatsThread listens to STATS\_QUEUE and processes incoming stats. As a StoppableThread subclass, this thread *must* check for the `.stop_requested` flag.

**`__init__`** (*max\_datapoints=500*, *timeout=1*)  
Init.

#### Parameters

- **`max_datapoints`** (*int*) – The maximum number of graph datapoints to record. If this limit is hit, datapoints will be discard based on age, oldest-first.
- **`timeout`** (*float*) – Frequency, in seconds, to check for a stop or remove request.

**`process_stat`** (*msg*)  
Accepts and processes stats messages.

**Parameters** *msg* (*Stat*) – A populated Stat object.

**Returns** None

**`run`** ()  
Main thread loop.

**`eww.stats.counter_manipulation`** (*stat*)  
Backend to all counter changes.

**Parameters** *stat* (*Stat*) – A populated Stat object.

**Returns** None

**`eww.stats.decr`** (*name*, *amount=1*)  
Reduces a counter.

#### Parameters

- **`name`** (*str*) – The name of the counter to decrement.
- **`amount`** (*int*) – The value to decrement *name* by.

**`eww.stats.graph`** (*name*, *datapoint*)  
Adds an X.Y datapoint.

#### Parameters

- **`name`** (*str*) – The name of the graph to record a datapoint for.
- **`datapoint`** (*tuple*) – A tuple representing an (X, Y) datapoint.

**Returns** None

```
eww.stats.incr(name, amount=1)
```

Increments a counter.

**Parameters**

- **name** (*str*) – The name of the counter to increment.
- **amount** (*int*) – The amount to increment `name` by.

**Returns** None

```
eww.stats.memory_consumption()
```

Returns memory consumption (specifically, max rss). Currently this uses the `resource` module, and is only available on Unix.

**Returns** MaxRSS value. On Windows, this always returns 0.

**Return type** int

```
eww.stats.put(name, amount=1)
```

Puts a counter to a specific value.

**Parameters**

- **name** (*str*) – The name of the counter to set to a specific value.
- **amount** (*int*) – The value to set `name` to.

**Returns** None

## 8.10 eww.stoppable\_thread

`threading.Thread` class that exposes a stop API. Subclasses of this must check for `.stop_requested` regularly.

```
class eww.stoppable_thread.StoppableThread
```

Bases: `threading.Thread`

Thread class that adds a `stop()` method. Subclasses *must* check for the `.stop_requested` event regularly.

```
__init__()
```

Init.

```
stop()
```

Sets the `stop_requested` flag.

## 8.11 scripts.eww

The Eww client is the recommended frontend for connecting to a listening Eww instance.

Before criticizing this implementation, it's important to understand why this was done.

I consider `readline` support a **hard** requirement.

`readline` is currently only supported by `raw_input()`. `readline` does expose a few calls into the underlying library, but not nearly what we need to implement it ourselves without using `raw_input()`.

There are a few solutions to this problem:

1. Implement `readline`-like support ourselves in Python
2. Call the underlying C library ourselves

3. Implement proper PTY support on both ends of the connection
4. Use `raw_input()`

Option 1 is a lot more work than it seems like. It would certainly be valuable, and an interesting library to create. The scope of work required to do it properly just for this is difficult to justify.

Option 2 would require a compilation step and complicate installation.

Option 3 would be ideal. However, implementing a cross-platform PTY over a socket in Python is difficult, error-prone, and tough to debug. It's absolutely on the table and would let us add a ton of features, but it's not happening right off the bat.

Which leaves us with option 4, and results in our current (difficult to test) implementation.

With that in mind, read on.

**exception** `scripts.eww.ConnectionClosed`

Bases: `exceptions.Exception`

Raised when a connection is closed.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**class** `scripts.eww.EwwClient` (*host*, *port*)

Bases: `object`

Manages all client communication.

**\_\_init\_\_** (*host*, *port*)

Init.

#### Parameters

- **host** (*str*) – A host to connect to.
- **port** (*int*) – A port to connect to.

**\_\_weakref\_\_**

list of weak references to the object (if defined)

**clientloop** (*debug=False*, *line=None*)

Repeatedly loops through `display_output` and `get_input`.

#### Parameters

- **debug** (*bool*) – Used for testing. If `debug` is `True`, `clientloop` only goes through one iteration.
- **line** (*str*) – If `debug` is `True`, this will be passed to `get_input`. Useful for testing.

**Returns** `None`

**connect** ()

Connects to an Eww instance.

**Returns** `None`

**display\_output** ()

Displays output from the Eww instance.

**Returns** `None`

**get\_input** (*line=None*)

Collects user input and sends it to the Eww instance.

**Parameters** `line` (*str*) – If provided, `line` is considered the input and `raw_input` is not used.

**Returns** `None`

`scripts.eww.main` (*debug=False, line=None, opt\_args=None*)

Main function.

**Parameters**

- **debug** (*bool*) – Used for testing. If `debug` is `True`, `clientloop` only goes through one iteration.
- **line** (*str*) – If `debug` is `True`, this will be passed to `get_input`. Useful for testing.
- **opt\_args** (*list*) – If `debug` is `True`, this list is parsed by `optparse` instead of `sys.argv`.

**Returns** `None`

## **e**

`eww.command`, [23](#)  
`eww.console`, [25](#)  
`eww.dispatch`, [26](#)  
`eww.implant`, [27](#)  
`eww.ioproxy`, [27](#)  
`eww.parser`, [28](#)  
`eww.quitproxy`, [29](#)  
`eww.shared`, [30](#)  
`eww.stats`, [30](#)  
`eww.stoppable_thread`, [32](#)

## **s**

`scripts.eww`, [32](#)