lox Documentation

Release 0.6.3

Brian Pugh

Feb 26, 2020

Contents

Many programs are embaressingly parallel and can gain large performance boost by simply parallelizing portions of the code. However, multithreading a program is still typically seen as a difficult task and placed at the bottom of the TODO list. **lox** aims to make it as simple and intuitive as possible to parallelize functions and methods in python. This includes both invoking functions, as well as providing easy-to-use guards for shared resources.

lox provides a simple, shallow learning-curve toolset to implement multithreading or multiprocessing that will work in most projects. **lox** is not meant to be the bleeding edge of performance; for absolute maximum performance, you code will have to be more fine tuned and may benefit from python3's builtin **asyncio**, **greenlet**, or other async libraries. **lox**'s primary goal is to provide that maximum concurrency performance in the least amount of time and the smallest refactor.

A very simple example is as follows.

```
>>> import lox
>>>
>>> @lox.thread(4) # Will operate with a maximum of 4 threads
... def foo(x,y):
...     return x*y
>>> foo(3,4)
12
>>> for i in range(5):
...     foo.scatter(i, i+1)
-ignore-
>>> # foo is currently being executed in 4 threads
>>> results = foo.gather() # block until results are ready
>>> print(results) # Results are in the same order as scatter() calls
[0, 2, 6, 12, 20]
```

CHAPTER 1

Features

- **Multithreading**: Powerful, intuitive multithreading in just 2 additional lines of code.
- **Multiprocessing**: Truly parallel function execution with the same interface as **multithreading**.
- **Synchronization**: Advanced thread synchronization, communication, and resource management tools.

CHAPTER 2

Contents

2.1 Installation

2.1.1 Stable release

To install lox, run this command in your terminal:

```
$ pip install lox
```

This is the preferred method to install lox, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.1.2 From sources

The sources for lox can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/BrianPugh/lox
```

Or download the tarball:

```
$ curl  -OL https://github.com/BrianPugh/lox/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

2.2 Modules

2.2.1 Worker

Add concurrency to methods and functions in a single line of code.

Thread

lox.worker.**thread**(*max_workers*, *daemon=None*)

Decorator to execute a function in multiple threads.

Example:

```
>>> import lox
>>>
>>> @lox.thread(4) # Will operate with a maximum of 4 threads
... def foo(x,y):
...     return x*y
>>> foo(3,4)
12
>>> for i in range(5):
...     foo.scatter(i, i+1)
-ignore-
>>> # foo is currently being executed in 4 threads
>>> results = foo.gather()
>>> print(results)
[0, 2, 6, 12, 20]
```

Multiple decorated functions can be chained together, each function drawing from their own pool of threads. Functions that return tuples will automatically unpack into the chained function. Positional arguments and keyword arguments can be passed in as they normally would.

```
>>>  for i in range(5):
...    foo_res = foo.scatter(i, i+1)
...    bar.scatter(foo_res, 10) # scatter will automatically unpack the results␣
↪of foo
>>>
>>> results = bar.gather()
```

Currently, a `scatter` call can have a maximum of 1 previous `scatter` result as an input argument. However, unlimited number of functions can be chained together in any topology.

Parameters **max_workers** (*int*) – Maximum number of threads to invoke. When `lox.thread` is called without `()`, the wrapped function a default number of max_workers is used (50).

lox.worker.**__call__**(**args*, ***kwargs*)

Vanilla passthrough function execution. Default user function behavior.

Returns Return of decorated function.

Return type Decorated function return type.

lox.worker.**__len__**()

Returns Current job queue length. Number of jobs that are currently waiting for an available worker.

Return type int

lox.worker.**scatter**(**args*, ***kwargs*)
Start a job executing decorated function `func( *args, **kwargs )`. Workers are created and destroyed automatically.

Returns Solution's index into the results obtained via `gather()`.

Return type int

lox.worker.**gather**()
Block until all jobs called via `scatter()` are complete.

Returns Results in the order that scatter was invoked.

Return type list

lox.worker.**disable_auto_unpacking**()
Automatically unpack previously chained input tuples.

lox.worker.**enable_auto_unpacking**()
Do not unpack previously chained input tuples.

Process

lox.worker.**process**(*n_workers*)
Decorator to execute a function/method in multiple processes.

Example:

```
>>> import lox
>>>
>>> @lox.process(4) # Will operate with a maximum of 4 processes
... def foo(x,y):
...     return x*y
>>> foo(3,4)
12
>>> for i in range(5):
...     foo.scatter(i, i+1)
>>> # foo is currently being executed in 4 processes
>>> results = foo.gather()
>>> print(results)
[0, 2, 6, 12, 20]
```

Parameters **n_workers** (*int*) – Number of process workers to invoke. Defaults to number of CPU cores.

lox.worker.**__call__**(**args*, ***kwargs*)
Vanilla passthrough function execution. Default user function behavior.

Returns Return of decorated function.

Return type Decorated function return type.

lox.worker.**__len__**()

Returns job queue length.

Return type int

lox.worker.**scatter**(**args*, ***kwargs*)
Start a job executing `func( *args, **kwargs )`. Workers are created and destroyed automatically.

Returns Solution's index into the results obtained via `gather()`.

Return type int

`lox.worker.`**`gather`**`()`
Block until all jobs called via `scatter()` are complete.

Returns Results in the order that scatter was invoked.

Return type list

2.2.2 Lock

Concurrency control objects to help parallelized tasks communicate and share resources.

LightSwitch

class `lox.lock.`**`LightSwitch`**(*lock*, *multiprocessing=False*)
Acquires a provided lock while `LightSwitch` is in use.

The lightswitch pattern creates a first-in-last-out synchronization mechanism. The name of the pattern is inspired by people entering a room in the physical world. The first person to enter the room turns on the lights; then, when everyone is leaving, the last person to exit turns the lights off.

`lock`
The lock provided to the constructor that may be acquired/released by `LightSwitch`.

Type threading.Lock

`counter`
Number of times the `LightSwitch` has been acquired without release.

Type int

`__enter__``()`
Acquire `LightSwitch` at context enter.

`__exit__`(*exc_type*, *exc_val*, *exc_tb*)
Release `LightSwitch` at context exit.

`__len__``()`
Get the counter value.

Returns counter value (number of times lightswitch has been acquired).

Return type int

`acquire`(*timeout=-1*)
Acquire the `LightSwitch` and increment the internal counter.

When the internal counter is incremented from zero, it will acquire the provided lock.

Parameters **`timeout`** (*`float`*) – Maximum number of seconds to wait before aborting.

Returns `True` on success, `False` on failure (like timeout).

Return type bool

`release``()`
Release the `LightSwitch` by decrementing the internal counter.

When the internal counter is decremented to zero, it will release the provided lock.

RWLock

class lox.lock.**RWLock**
Lock for a Multi-Reader-Single-Writer scenario.

Unlimited numbers of reader can obtain the lock, but as soon as a writer attempts to acquire the lock, all reads are blocked until the current readers are finished, the writer acquires the lock, and finally releases it.

Similar to a `lox.LightSwitch`, but blocks incoming "readers" while a "write" is trying to be performed.

read_counter
Number of readers that have acquired the lock.

Type int

__len__()
Get the read_counter value

Returns Number of current readers

Return type int

acquire(*rw_flag: str*, *timeout=-1*)
Acquire the lock as a "reader" or a "writer".

Parameters

- **rw_flag** (*str*) – Either 'r' for 'read' or 'w' for 'write' acquire.
- **timeout** (*float*) – Time in seconds before timeout occurs for acquiring lock.

Returns `True` if lock was acquired, `False` otherwise.

Return type bool

release(*rw_flag: str*)
Release acquired lock.

Parameters **rw_flag** (*str*) – Either 'r' for 'read' or 'w' for 'write' acquire.

QLock

class lox.lock.**QLock**
Lock that guarentees FIFO operation. Approximately 6x slower than a normal `Lock()`.

Modified from https://stackoverflow.com/a/19695878

__enter__()
Acquire `QLock` at context enter.

__exit__(*exc_type*, *exc_val*, *exc_tb*)
Release `QLock` at context exit.

__len__()

Returns Number of tasks waiting to acquire.

Return type int

acquire(*timeout=-1*)
Block until resource is available.

Threads that call `acquire` obtain resource FIFO.

Parameters **timeout** (*float*) – Maximum number of seconds to wait before aborting.

Returns True on successful acquire, False on timeout.

Return type bool

locked
Whether or not the QLock is acquired

release()
Release exclusive access to resource.

ValueError Lock released more than it has been acquired.

IndexSemaphore

class lox.lock.**IndexSemaphore**(*val*)
BoundedSemaphore-like object where acquires return an index from [0, val).

Example usecase: thread acquiring a GPU.

Example

```
>>> sem = IndexSemaphore(4)
>>> with sem() as index:
>>>     print("Obtained resource %d" % (index,))
>>>
Obtained resource 0
```

__len__()

Returns Current blocked queue size.

Return type int

acquire(*timeout=None*)
Blocking acquire resource.

Parameters **timeout** (*float*) – Maximum number of seconds to wait before returning.

Returns Resource index on successful acquire. None on timeout.

Return type int

release(*index*)
Release resource at index.

Parameters **index** (*int*) – Index of resource to release.

Raises Exception – Resource has been released more times than acquired.

2.2.3 Queue

Announcement

class lox.queue.**Announcement**(*maxsize=0*, *backlog=None*)
Push to many queues with backlog support.

Allows the pushing of data to many threads.

Example:

```
>>> import lox
>>> ann = lox.Announcement()
>>> foo_q = ann.subscribe()
>>> bar_q = ann.subscribe()
>>>
>>> @lox.thread
... def foo():
...     x = foo_q.get()
...     return x
>>>
>>> @lox.thread
... def bar():
...     x = bar_q.get()
...     return x**2
>>>
>>> ann.put(5)
>>> foo.scatter()
>>> foo_res = foo.gather()
>>> bar.scatter()
>>> bar_res = bar.gather()
```

The backlog allows future (or potentially race-condition) subscribers to get content put'd before they subscribed. However, the user must be careful of memory consumption.

backlog
Backlog of queued data. None if not used.

Type deque

__len__()
Get the number of subscribers.

Returns Number of subcribers.

Return type int

classmethod clone(*ann*, *q: queue.Queue = None*)
Create a new announcement object that shares subscribers and resources with an existing announcement.

Only difference from cloned announcement is a new receive queue is created.

Parameters

- **ann** (*lox.Announcement*) – Announcement object to clone from
- **q** (*queue.Queue*) – Receiving queue. If `None`, a new one is created.

Returns New Announcement object with copied attributes, but new `q`

Return type *Announcement*

empty()
Return `True` if the receive queue is empty, `False` otherwise. If `empty()` returns `True` it doesn't guarantee that a subsequent call to `put()` will not block. Similarly, if `empty()` returns `False` it doesn't guarantee that a subsequent call to `get()` will not block.

Returns `True` if the receive queue is currently empty; `False` otherwise.

Return type bool

finalize()
Do not allow any more subscribers.

Primarily used for memory efficiency if backlog is used.

full()
Return `True` if the receive queue is full, `False` otherwise. If `full()` returns `True` it doesn't guarantee that a subsequent call to `get()` will not block. Similarly, if `full()` returns `False` it doesn't guarantee that a subsequent call to `put()` will not block.

Returns `True` if the receive queue is currently full; `False` otherwise.

Return type bool

get(*block=True*, *timeout=None*)
Get from the receive queue.

Parameters

- **block** (*bool*) – Block until data is obtained from receive queue or timeout.
- **timeout** (*float*) – Wait up to `timeout` seconds before raising `queue.Full`. Defaults to no timeout.

Returns

Return type item from receive queue.

Raises `queue.Empty` – When there are no elements in queue and timeout has been reached.

put(*item*, *block=True*, *timeout=None*)
Put item into all subscribers' queues.

Parameters

- **item** – data to put onto all subscribers' queues
- **block** (*bool*) – Block until data is put on queues or timeout.
- **timeout** (*float*) – Wait up to `timeout` seconds before raising `queue.Full`. Defaults to no timeout.

qsize()
Return the approximate size of the receive queue. Note, qsize() > 0 doesn't guarantee that a subsequent get() will not block, nor will qsize() < maxsize guarantee that put() will not block.

Returns approximate size of the receive queue.

Return type int

subscribe(*q=None*, *maxsize=None*, *block=True*, *timeout=None*)
Subscribe to announcements.

Parameters

- **q** (*Queue*) – Existing queue to add to the subscribers' list. If not provided, a queue is created.
- **maxsize** (*int*) – Created queue's maximum size. Overrides Announcement's default maximum size. Ignored if `q` is provided.
- **block** (*bool*) – Block until data from backlog is put on queues or timeout.
- **timeout** (*float*) – Wait up to `timeout` seconds before raising `queue.Full`. Defaults to no timeout.

Returns object for receiver to `get` and `put` data from.

Return type *Announcement*

unsubscribe(*q*)
Remove the queue from queue-list. Will no longer receive announcements.

Parameters **q** (*Queue*) – Queue object to remove.

Raises ValueError – q was not a subscriber.

Funnel

class lox.queue.**Funnel**
Wait on many queues.

```
>>> funnel = lox.Funnel()
>>> sub_1 = funnel.subscribe()
>>> sub_2 = funnel.subscribe()

>>> sub_1.put('foo', 'job_id')
>>> try:
...     res = funnel.get(timeout=0.01)
... except queue.Empty:
...     print("Timed Out")
Timed Out
>>> sub_2.put('bar', 'job_id')
>>> res = funnel.get()
>>> print(len(res))
3
>>> print(res)
['job_id','foo','bar']
```

index
Index into list of solutions (if a subscriber). -1 otherwise. Note: if get(return_jid=True) then this is offset by one.

Type int

__len__()
Gets number of input queues.

Returns Number of input queues.

Return type int

get(*block=True*, *timeout=None*, *return_jid=True*)
Get from the receive queue. Will return the contents of each input queue in the order subscribed as a tuple

Parameters

- **block** (*bool*) – Block until data is obtained from receive queue or timeout.
- **timeout** (*float*) – Wait up to timeout seconds before raising queue.Full. Defaults to no timeout.
- **return_jid** (*bool*) – Have the Job ID as the first element of the returned tuple. Defaults to True

Returns items from input queues.

Return type tuple

Raises queue.Empty – When there are no elements in queue and timeout has been reached.

put(*item*, *jid*, *blocking=True*, *timeout=-1*)

Parameters

- **item** – data to put onto all subscribers' queues
- **jid** (*hashable*) – unique identifier for job.
- **block** (*bool*) – Block until data is put on queues or timeout.
- **timeout** (*float*) – Wait up to `timeout` seconds before raising `queue.Full`. Defaults to no timeout.

Returns True if item was successfully added; False otherwise.

Return type bool

Raises `FunnelPutTopError` – Can only put onto subscribers, not the top/master `Funnel`.

subscribe()

Create a new Funnel for data to be `put` on.

Returns A funnel object that is a required input on `get` calls.

Return type *Funnel*

2.3 Examples

2.3.1 Multithreading Requests

A typical usecase for **lox** is the following. Say you wanted to get the content of websites from a list of URLs. The first naive implementation may look something like the following.

```
>>> import urllib.request
>>> from time import time
>>> urls = ['http://google.com', 'http://bing.com', 'http://yahoo.com']
>>> responses = []
>>>
>>> def get_content(url):
...     res = urllib.request.urlopen(url)
...     return res.read()
>>>
>>> t_start = time()
>>> for url in urls:
...     responses.append(get_content(url))
>>> t_diff = time() - t_start
>>> print("It took %.3f seconds to get 3 sites" % (t_diff, ))
It took 2.942 seconds to get 3 sites
```

It's nice, simple, and it just works. However, your computer is just idling while waiting for a network response. With **lox**, you can just decorate the function you want to add concurrency. We replace the direct calls to the function with `func.scatter` which will pass all the `args` and `kwargs` to the decorated function. Finally, when we need all the function results, we call `func.gather()` which will return a list of the outputs of the decorated function. The outputs are guarenteed to be in the same order that the `scatter` were called

```
>>> import lox
>>> import urllib.request
>>> from time import time
>>> urls = ['http://google.com', 'http://bing.com', 'http://yahoo.com']
>>>
```

(continues on next page)

(continued from previous page)

```
>>> @lox.thread
... def get_content(url):
...     res = urllib.request.urlopen(url)
...     return res.read()
>>>
>>> t_start = time()
>>> for url in urls:
...     get_content.scatter(url)
-ignore-
>>> responses = get_content.gather()
>>> t_diff = time() - t_start
>>> print("It took %.3f seconds to get 3 sites" % (t_diff, ))
It took 0.928 seconds to get 3 sites
```

With minimal modifications, we now have a multithreaded application with significant performance improvements.

2.3.2 Multithreading Chaining

Pipelining data between worker pools is another common application. Say you have a pool of workers fetching images from disk, and another pool of 2 workers processing them on 2 GPUs.

```
>>> import lox
>>> import numpy as np
>>> from time import sleep
>>>
>>> @lox.thread(10)
... def fetch_image(name):
...     # Dummy image
...     sleep(1)
...     return np.zeros((100,100,3), dtype=np.uint8)
>>>
>>> @lox.thread(2)
... def process_image( im )
...     sleep(1) # Pretend this is an expensive operation on a GPU
...     return im / 255.0
>>>
>>> fns = ['im1.png', 'im2.png', 'im3.png', 'im4.png', 'im5.png', ]
>>> for fn in fns:
...     im = fetch_image.scatter(fn) # Returns an index/promise of the result.
...     process_image.scatter(im)
>>>
>>> list_of_processed_images = process_image.gather()
```

`process_image.scatter(...)` will automatically unpack the return value of `fetch_image(...)` if it's a tuple. Positional and keyword arguments can also be passed into *process_image.scatter*. A current limitation is that only 1 or fewer promises can be fed into a scatter call.

2.3.3 Multiprocessing

```
>>> import lox
>>> from time import sleep
>>>
>>> @lox.process(2)
```

(continues on next page)

(continued from previous page)

```
... def job(x):
...     sleep(1)
...     return 1
>>>
>>> t_start = time()
>>> for i in range(5):
...     res = job(10)
>>> t_diff = time() - t_start
>>> print("Non-parallel took %.3f seconds" % (t_diff, ))
Non-parallel took 5.007 seconds
>>>
>>> t_start = time()
>>> for i in range(5):
...     job.scatter(10)
>>> res = job.gather()
>>> t_diff = time() - t_start
>>> print("Parallel took %.3f seconds" % (t_diff, ))
Parallel took 0.062 seconds
```

2.3.4 Obtaining a resource from a pool

Imagine you have 4 GPUs that are part of a data processing pipeline, and the GPUs perform the task disproportionally faster (or slower!) than the rest of the pipeline. Below we have many threads fetching and processing data, but they need to share the 4 GPUs for accelerated processing.

```
>>> import lox
>>>
>>> N_GPUS = 4
>>> gpus = [allocate_gpu(x) for x in range(N_GPUS)]
>>> idx_sem = lox.IndexSemaphore(N_GPUS)
>>>
>>> @lox.thread
... def process_task(url):
...     data = get_data(url)
...     data = preprocess_data(data)
...     with idx_sem() as idx: # Obtains 0, 1, 2, or 3
...         gpu = gpus[idx]
...         result = gpu.process(data)
...     result = postprocess_data(data)
...     save_file(result)
>>>
>>> urls = ['http://google.com', ]
>>> for url in urls:
...     process_task.scatter(url)
>>> process_task.gather()
```

2.3.5 Block until threads are done

Imagine the following scenario:

A janitor needs to clean a restroom, but is not allowed to enter until all people are out of the restroom. How do we implement this?

The easiest way is to use a **lox.LightSwitch**. The lightswitch pattern creates a first-in-last-out synchronization mechanism. The name of the pattern is inspired by people entering a room in the physical world. The first person to enter the room turns on the lights; then, when everyone is leaving, the last person to exit turns the lights off.

```
>>> restroom_occupied = Lock()
>>> restroom = LightSwitch( restroom_occupied )
>>> res = []
>>> n_people = 5
```

A **LightSwitch** is most similar to a semaphore, but it automatically acquires/releases a provided **Lock** when it's internal counter increments/decrements from 0. A **LightSwitch** can be acquired multiple times, but must be released the same amount of times before the **Lock** gets released.

Here's the janitor's job:

```
>>> @lox.thread(1)
... def janitor():
...     with restroom_occupied: # block until the restroom is no longer occupied
...         res.append('j_enter')
...         print("(%0.3f s) Janitor  entered the restroom" % ( time() - t_start,))
...         sleep(1) # clean the restroom
...         res.append('j_exit')
...         print("(%0.3f s) Janitor  exited  the restroom" % ( time() - t_start,))
```

Here are the people trying to enter the rest room:

```
>>> @lox.thread(n_people)
... def people( id ):
...     if id == 0: # Get the starting time of execution for display purposes
...         global t_start
...         t_start = time()
...     with restroom: # block if a janitor is in the restroom
...         res.append("p_%d_enter" % (id,))
...         print("(%0.3f s) Person %d entered the restroom" % ( time() - t_start, id,
↪))
...         sleep(1) # use the restroom
...         res.append("p_%d_exit" % (id,))
...         print("(%0.3f s) Person %d exited  the restroom" % ( time() - t_start, id,
↪))
```

Lets start these people up:

```
>>> for i in range(n_people):
...     people.scatter(i)               # Person i will now attempt to enter the␣
↪restroom
...     sleep(0.6)                      # wait for 60% the time a person spends in the␣
↪restroom
...     if i==0:                        # While the first person is in the restroom...
...         janitor_thread.start()      # the janitor would like to enter. HOWEVER...
...         print("(%0.3f s) Janitor Dispatched" % (time()-t_start))
>>> # Wait for all threads to finish
>>> people.gather()
>>> janitor.gather()
```

The results will look like:

```
Running Restroom Demo
(0.000 s) Person 0 entered the restroom
```

(continues on next page)

(continued from previous page)

```
(0.061 s) Person 1 entered the restroom
(0.100 s) Person 0 exited  the restroom
(0.122 s) Person 2 entered the restroom
(0.162 s) Person 1 exited  the restroom
(0.182 s) Person 3 entered the restroom
(0.222 s) Person 2 exited  the restroom
(0.243 s) Person 4 entered the restroom
(0.282 s) Person 3 exited  the restroom
(0.343 s) Person 4 exited  the restroom
(0.343 s) Janitor  entered the restroom
(0.443 s) Janitor  exited  the restroom
```

Note that multiple people can be in the restroom. If people kept using the restroom, the Janitor would never be able to enter (technically known as thread starvation). If this is undesired for your application, look at RWLock

2.3.6 One-Writer-Many-Reader

It's common that many threads may be reading from a single resource, but a single other thread may change the value of that resource.

If we used a LightSwitch as in the Janitor example above, we can see that the writer (Janitor) may never get an opporunity to acquire the resource. A **RWLock** solves this problem by blocking future threads from acquiring the resource until the writer acquires and subsequently releases the resource.

```
>>> rwlock = lox.RWLock()
```

The janitor task would do something like:

```
>>> with rwlock('w'):
...     # Perform resource write here
```

While the people task would look like

```
>>> with rwlock('r'):
...     # Perform resource read here
```

2.4 FAQ

Q: Whats the difference between multithreading and multiprocessing?

A: Multithreading and Multiprocessing are two different methods to provide concurrency (parallelism) to your code.

Threading has low overhead for sharing resources between threads. Threads share the same heap, meaning global variables are easily accessible from each thread. However, at any given moment, only a single line of python is being executed, meaning if your code is CPU-bound, using threading will have the same performance (actually worse due to overhead) as not using threading.

Multiprocessing is basically several copies of your python code running at once, communicating over pipes. Each worker has it's own python interpretter, it's own stack, it's own heap, it's own everything. Any data transferred between your main program and the workers must first be serialized (using **dill**, a library very similar to **pickle**) passed over a pipe, then deserialized.

In short, if your project is I/O bound (web requests, reading/writing files, waiting for responses from compiled code/binaries, etc), threading is probably the better choice. However, if your code is computation bound, and if

the libraries you are using aren't using compiled backends that are already maxing out your CPU, multiprocessing might be the better option.

Q: Why not just use the built-in `await` ?

A: Trying to shove `await` into a project typically requires great care both in the code written and the packages used. Ontop of this, using await may require a substantial refactor of the layout of the code. The goal of **lox** is to require the smallest, least risky changes in your codebase.

2.5 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

2.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/BrianPugh/lox/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

lox could always use more documentation, whether as part of the official lox docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/BrianPugh/lox/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.

- Remember that this is a volunteer-driven project, and that contributions are welcome :)

2.5.2 Get Started!

Ready to contribute? Here's how to set up *lox* for local development.

1. Fork the *lox* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/lox.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv lox
$ cd lox/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 lox tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

2.5.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/BrianPugh/lox/pull_requests and make sure that the tests pass for all supported Python versions.

2.5.4 Tips

To run a subset of tests:

```
$ py.test tests.test_lox
```

2.5.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

2.6 Credits

2.6.1 Development Lead

- Brian Pugh <bnp117@gmail.com>

2.6.2 Contributors

None yet. Why not be the first?

2.7 History

2.7.1 0.6.3 (2019-07-30)

- Alternative fix for 0.6.2.

2.7.2 0.6.2 (2019-07-21)

- Update dependencies
- Fix garbage-collecting exclusiviity

2.7.3 0.6.1 (2019-07-21)

- Fix memory leak in `lox.process`.

2.7.4 0.6.0 (2019-07-21)

- `lox.Announcement` `subscribe()` calls now return another `Announcement` object that behaves like a queue instead of an actual queue. Allows for many-queue-to-many-queue communications.
- New Object: `lox.Funnel`. allows for waiting on many queues for a complete set of inputs indicated by a job ID.

2.7.5 0.5.0 (2019-07-01)

- New Object: `lox.Announcement`. Allows a one-to-many thread queue with backlog support so that late subscribers can still get all (or most recent) announcements before they subscribed.
- New Feature: `lox.thread` `scatter` calls can now be chained together. `scatter` now returns an `int` subclass that contains metadata to allow chaining. Each scatter call can have a maximum of 1 previous `scatter` result.
- Documentation updates, theming, and logos

2.7.6 0.4.3 (2019-06-24)

- Garbage collect cached decorated object methods

2.7.7 0.4.2 (2019-06-23)

- Fixed multiple instances and successive scatter and gather calls to wrapped methods

2.7.8 0.4.1 (2019-06-23)

- Fixed broken workers and unit tests for workers

2.7.9 0.4.0 (2019-06-22)

- Semi-breaking change: **lox.thread** and **lox.process** now automatically pass the object instance when decorating a method.

2.7.10 0.3.4 (2019-06-20)

- Print traceback in red when a thread crashes

2.7.11 0.3.3 (2019-06-19)

- Fix bug where thread in scatter of lox.thread double releases on empty queue

2.7.12 0.3.2 (2019-06-17)

- Fix manifest for installation from wheel

2.7.13 0.3.1 (2019-06-17)

- Fix package on pypi

2.7.14 0.3.0 (2019-06-01)

- Multiprocessing decorator. **lox.pool** renamed to **lox.thread**
- Substantial pytest bug fixes
- Documentation examples
- timeout for RWLock

2.7.15 0.2.1 (2019-05-25)

- Fix IndexSemaphore context manager

2.7.16 0.2.0 (2019-05-24)

- Added QLock
- Documentation syntax fixes

2.7.17 0.1.1 (2019-05-24)

- CICD test

2.7.18 0.1.0 (2019-05-24)

- First release on PyPI.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

Symbols

A

B

C

D

E

F

G

I

L

P

Q

R

S

T

U