
Eliot Documentation

Release 1.13.0+0.ge858c8e

Itamar Turner-Trauring

Dec 15, 2020

Contents

1	Media	3
2	Testimonials	5
3	Documentation	7
3.1	Quickstart	7
3.2	Why Eliot?	9
3.3	What's New	11
3.4	Generating Logs	19
3.5	Outputting Logs	42
3.6	Reading Logs	46
3.7	Scientific Computing with Eliot	48
3.8	Python 2.7 Support	51
3.9	Contributing to Eliot	51

Python's built-in `logging` and other similar systems output a stream of factoids: they're interesting, but you can't really tell what's going on.

- Why is your application slow?
- What caused this code path to be chosen?
- Why did this error happen?

Standard logging can't answer these questions.

But with a better model you could understand what and why things happened in your application. You could pinpoint performance bottlenecks, you could understand what happened when, who called what.

That is what Eliot does. `eliot` is a Python logging system that outputs causal chains of **actions**: actions can spawn other actions, and eventually they either **succeed or fail**. The resulting logs tell you the story of what your software did: what happened, and what caused it.

Eliot supports a range of use cases and 3rd party libraries:

- Logging within a single process.
- Causal tracing across a distributed system.
- Scientific computing, with *built-in support for NumPy and Dask*.
- *Asyncio and Trio coroutines* and the *Twisted networking framework*.

Eliot is only used to generate your logs; you might still need tools like Logstash and ElasticSearch to aggregate and store logs if you are using multiple processes across multiple machines.

- **Start here:** [Quickstart documentation](#)
- Need help or have any questions? [File an issue](#).
- Eliot is licensed under the [Apache 2.0 license](#), and the source code is [available on GitHub](#).
- Eliot supports Python 3.9, 3.8, 3.7, 3.6, and PyPy3. Python 2.7 is in legacy support mode (see [Python 2.7 Support](#) for details).
- **Commercial support** is available from [PythonSpeed](#).
- Read on for the full documentation.

CHAPTER 1

Media

PyCon 2019 talk: [Logging for Scientific Computing](#) (also available in a prose version).

Podcast `__init__` episode 133 covers Eliot:

CHAPTER 2

Testimonials

“Eliot has made tracking down causes of failure (in complex external integrations and internal uses) tremendously easier. Our errors are logged to Sentry with the Eliot task UUID. That means we can go from a Sentry notification to a high-level trace of operations—with important metadata at each operation—in a few seconds. We immediately know which user did what in which part of the system.”

—Jonathan Jacobs

3.1 Quickstart

Let's see how easy it is to use Eliot.

3.1.1 Installing Eliot

To install Eliot and the other tools we'll use in this example, run the following in your shell:

```
$ pip install eliot eliot-tree requests
```

You can also install it using Conda:

```
$ conda install -c conda-forge eliot eliot-tree requests
```

This will install:

1. Eliot itself.
2. `eliot-tree`, a tool that lets you visualize Eliot logs easily.
3. `requests`, a HTTP client library we'll use in the example code below. You don't need it for real Eliot usage, though.

3.1.2 Our example program

We're going to add logging code to the following script, which checks if a list of links are valid URLs:

```
import requests

def check_links(urls):
    for url in urls:
```

(continues on next page)

(continued from previous page)

```

    try:
        response = requests.get(url)
        response.raise_for_status()
    except Exception as e:
        raise ValueError(str(e))

try:
    check_links(["http://eliot.readthedocs.io", "http://nosuchurl"])
except ValueError:
    print("Not all links were valid.")

```

3.1.3 Adding Eliot logging

To add logging to this program, we do two things:

1. Tell Eliot to log messages to file called “linkcheck.log” by using `eliot.to_file()`.
2. Create two actions using `eliot.start_action()`. Actions succeed when the `eliot.start_action()` context manager finishes successfully, and fail when an exception is raised.

```

import requests
from eliot import start_action, to_file
to_file(open("linkcheck.log", "w"))

def check_links(urls):
    with start_action(action_type="check_links", urls=urls):
        for url in urls:
            try:
                with start_action(action_type="download", url=url):
                    response = requests.get(url)
                    response.raise_for_status()
            except Exception as e:
                raise ValueError(str(e))

try:
    check_links(["http://eliot.readthedocs.io", "http://nosuchurl"])
except ValueError:
    print("Not all links were valid.")

```

3.1.4 Running the code

Let’s run the code:

```

$ python linkcheck.py
Not all the links were valid.

```

We can see the resulting log file is composed of JSON messages, one per line:

```

$ cat linkcheck.log
{"action_status": "started", "task_uuid": "b1cb58cf-2c2f-45c0-92b2-838ac00b20cc",
↪ "task_level": [1], "timestamp": 1509136967.2066844, "action_type": "check_links",
↪ "urls": ["http://eliot.readthedocs.io", "http://nosuchurl"]}
...

```

So far these logs seem similar to the output of regular logging systems: individual isolated messages. But unlike those logging systems, Eliot produces logs that can be reconstructed into a tree, for example using the `eliot-tree` utility:

```
$ eliot-tree linkcheck.log
b1cb58cf-2c2f-45c0-92b2-838ac00b20cc
└─ check_links/1 started
    └─ timestamp: 2017-10-27 20:42:47.206684
        └─ urls:
            └─ 0: http://eliot.readthedocs.io
                └─ 1: http://nosuchurl
                    └─ download/2/1 started
                        └─ timestamp: 2017-10-27 20:42:47.206933
                            └─ url: http://eliot.readthedocs.io
                                └─ download/2/2 succeeded
                                    └─ timestamp: 2017-10-27 20:42:47.439203
                                        └─ download/3/1 started
                                            └─ timestamp: 2017-10-27 20:42:47.439412
                                                └─ url: http://nosuchurl
                                                    └─ download/3/2 failed
                                                        └─ errno: None
                                                            └─ exception: requests.exceptions.ConnectionError
                                                                └─ reason: HTTPConnectionPool(host='nosuchurl', port=80): Max retries_
                                                                    ↪exceeded with url: / (Caused by NewConne...
                                                                        └─ timestamp: 2017-10-27 20:42:47.457133
                                                                            └─ check_links/4 failed
                                                                                └─ exception: builtins.ValueError
                                                                                    └─ reason: HTTPConnectionPool(host='nosuchurl', port=80): Max retries_
                                                                                        ↪exceeded with url: / (Caused by NewConne...
                                                                                            └─ timestamp: 2017-10-27 20:42:47.457332
```

Notice how:

1. Eliot tells you which actions succeeded and which failed.
2. Failed actions record their exceptions.
3. You can see just from the logs that the `check_links` action caused the download action.

3.1.5 Next steps

You can learn more by reading the rest of the documentation, including:

- The *motivation behind Eliot*.
- How to generate *actions*, *standalone messages*, and *handle errors*.
- How to integrate or migrate your *existing stdlib logging messages*.
- How to output logs *to a file or elsewhere*.
- Using *asyncio or Trio coroutines, threads and processes*, or *Twisted*.
- Using Eliot for *scientific computing*.

3.2 Why Eliot?

Suppose we turn from outside estimates of a man, to wonder, with keener interest, what is the report of his own consciousness about his doings or capacity: with what hindrances he is carrying on his daily labors;

what fading of hopes, or what deeper fixity of self-delusion the years are marking off within him; and with what spirit he wrestles against universal pressure, which will one day be too heavy for him, and bring his heart to its final pause.

— George Eliot, *Middlemarch*

The log messages generated by a piece of software ought tell a story: what, where, when, even why and how if you're lucky. But most logging systems omit the all-important *why*. You know that some things happened, but not how they relate to each other.

3.2.1 The problem: What caused this to happen?

Most log messages in your program are going to involve actions:

Not long after that dinner-party she had become Mrs. Casaubon, and was on her way to Rome.

A marriage has a beginning and eventually an end. The end may be successful, presuming “until death do us part” is a form of success, or a failure. The same is true of all actions, much like function calls in Python are started and eventually return a result or throw an exception. Actions may of course span multiple function calls or extended periods of time.

Actions also generate other actions: a marriage leads to a trip to Rome, the trip to Rome might lead to a visit to the Vatican Museum, and so on. Other unrelated actions are occurring at the same time, resulting in a forest of actions, with root actions that grow a tree of child actions.

You might want to trace an action from beginning to end, e.g. to measure how long it took to run. You might want to know what high-level action caused a particular unexpected low-level action. You might want to know what actions a specific entity was involved with.

None of these are possible in most logging systems since they have no concept of actions.

3.2.2 The solution: Eliot

Eliot is designed to solve these problems: the basic logging abstraction is the action.

An “action” is something with a start and an end; the end can be successful or it can fail due to an exception. Log messages, as well as log actions, know the log action whose context they are running in. The result is a tree of actions.

In the following example we have one top-level action (the honeymoon), which leads to other action (travel):

```
from sys import stdout
from eliot import start_action, to_file
to_file(stdout)

class Place(object):
    def __init__(self, name, contained=()):
        self.name = name
        self.contained = contained

    def visited(self, people):
        # No need to repetitively log people, since caller will:
        with start_action(action_type="visited", place=self.name):
            for thing in self.contained:
                thing.visited(people)

def honeymoon(family, destination):
```

(continues on next page)

(continued from previous page)

```

with start_action(action_type="honeymoon", people=family):
    destination.visited(family)

honeymoon(["Mrs. Casaubon", "Mr. Casaubon"],
          Place("Rome, Italy",
                [Place("Vatican Museum",
                       [Place("Statue #1"), Place("Statue #2")])]))

```

Actions provide a Python context manager. When the action starts, a start message is logged. If the block finishes successfully a success message is logged for the action; if an exception is thrown a failure message is logged for the action with the exception type and contents.

By default the messages are machine-parseable JSON, but for human consumption a visualization is better. Here's how the log messages generated by the new code look, as summarized by the `eliot-tree` tool:

```

f9dcc74f-ecda-4543-9e9a-1bb062d199f0
+-- honeymoon@1/started
  |-- people: ['Mrs. Casaubon', 'Mr. Casaubon']
  +-- visited@2,1/started
    |-- place: Rome, Italy
    +-- visited@2,2,1/started
      |-- place: Vatican Museum
      +-- visited@2,2,2,1/started
        |-- place: Statue #1
        +-- visited@2,2,2,2/succeeded
      +-- visited@2,2,3,1/started
        |-- place: Statue #2
        +-- visited@2,2,3,2/succeeded
      +-- visited@2,2,4/succeeded
    +-- visited@2,3/succeeded
  +-- honeymoon@3/succeeded

```

No longer isolated fragments of meaning, our log messages are now a story. Log events have context, you can tell where they came from and what they led to without guesswork.

Was looking at a statue the result of the honeymoon? It most definitely was.

3.3 What's New

3.3.1 1.13.0

Features:

- `@capture_logging` and `MemoryLogger` now support specifying a custom JSON encoder. By default they now use Eliot's encoder. This means tests can now match the encoding used by a `FileDestination`.
- Added support for Python 3.9.

Deprecation:

- Python 3.5 is no longer supported.

3.3.2 1.12.0

Features:

- Dask support now includes support for tracing logging of `dask.persist()`, via wrapper API `eliot.dask.persist_with_trace()`.

Bug fixes:

- Dask edge cases that previously weren't handled correctly should work better.

3.3.3 1.11.0

Features:

- `Message.log()` has been replaced by top-level function `log_message()`. Or if you're in the context of action `ctx`, you can call `ctx.log()`. See *Messages* for details.
- Python 3.8 is now supported.
- The `eliot-prettyprint` command line tool now supports a more compact format by using the `--compact` argument.
- The `eliot-prettyprint` command line tool now supports outputting in local timezones using the `--local-timezone` argument.

3.3.4 1.10.0

Bug fixes:

- `@eliot.testing.capture_logging` now passes `*args` and `**kwargs` to the wrapped function, as one would expect. Fixes #420. Thanks to Jean-Paul Calderone for the bug report.
- Eliot works with Dask 2.0. Thanks to Dan Myung for the bug report.

3.3.5 1.9.0

Deprecation:

- Python versions older than 3.5.3, e.g. the 3.5.2 on Ubuntu Xenial, don't work with Eliot, so added a more informative error message explaining that. Fixes #418. Thanks to Richard van der Hoff for the bug report.

Features:

- If you call `to_file()/FileDestination()` with a non-writable file, an exception will be raised. This prevents logging from being silently swallowed when the program runs. Fixes #403.
- PyPy3 is now officially supported.

Changes:

- If you log a NumPy array whose size > 10000, only a subset will be logged. This is to ensure logging giant arrays by mistake doesn't impact your software's performance. If you want to customize logging of large arrays, see *Logging large arrays*. Fixes #410.

3.3.6 1.8.0

Features:

- Eliot now supports Trio coroutines, as well as other frameworks that utilize Python 3.7's `contextvars` (Python 3.5 and 3.6 are also supported, using backport packages).

Deprecation:

- `eliot.use_asyncio_context()` is no longer necessary. On Python 3.5 and 3.6, however, you should make sure to import `eliot` (or `aioccontextvars`) before you start your first event loop.

Changes:

- Python 2.7 is now in legacy support mode; the last major Eliot release supporting it is 1.7.0. See [Python 2.7 Support](#) for details.
- Python 3.4 is no longer supported.

3.3.7 1.7.0

Documentation:

- Eliot has an API for testing that your logs were output correctly. Until now, however, the documentation was overly focused on requiring usage of types, which are optional, so it has been rewritten to be more generic: [read more about the testing API here](#).

Features:

- Generating messages is much faster.
- Eliot now works with PyInstaller. Thanks to Jean-Paul Calderone for the bug report. Fixes issue #386.
- The testing infrastructure now has slightly more informative error messages. Thanks to Jean-Paul Calderone for the bug report. Fixes issue #373.
- Added lower-level testing infrastructure—`eliot.testing.swap_logger` and `eliot.testing.check_for_errors`—which is useful for cases when the `@capture_logging` decorator is insufficient. For example, test methods that are async, or return Twisted `Deferred`. See the [testing documentation](#) for details. Thanks to Jean-Paul Calderone for the feature request. Fixes #364.
- `eliot.ValidationError`, as raised by e.g. `capture_logging`, is now part of the public API. Fixed issue #146.

Twisted-related features:

- New decorator, `@eliot.twisted.inline_callbacks`, which is like Twisted's `inlineCallbacks` but which also manages the Eliot context. Thanks to Jean-Paul Calderone for the fix. Fixed issue #259.
- `eliot.twisted.DeferredContext.addCallbacks` now supports omitting the `errback`, for compatibility with Twisted's `Deferred`. Thanks to Jean-Paul Calderone for the fix. Fixed issue #366.

Bug fixes:

- Fixed bug in the `asyncio` coroutine support where only the thread where `use_asyncio_context()` was called supported coroutine-specific contexts. Fixes issue #388.
- `ILogger.write` is now explicitly thread-safe. The `MemoryLogger` (as used by tests) implementation of this method which was previously not thread-safe is now thread-safe. Thanks to Jean-Paul Calderone for the patch. Fixes issue #382.

3.3.8 1.6.0

Deprecation:

- Python 2 is still supported, but will be dropped in one of the next releases. See *Python 2.7 Support*.

Features:

- NumPy integers, floats, bools and arrays are now automatically serialized to JSON, via a new default JSON encoder (`eliot.json.EliotJSONEncoder`).
- Dask support: replace `dask.compute()` with `eliot.dask.compute_with_trace()` to automatically preserve Eliot context for Bag and Delayed Dask computations. See *Using Dask* for details.
- New decorator, `@eliot.log_call`, which automatically creates an action that starts when function is called and ends when it returns. See *Logging Functions*.
- A parser library for parsing serialized Eliot JSON messages into a tree of Python objects. See *Parsing Logs* for details.

Testing features:

- `eliot.testing.LoggedAction` has a new method, `type_tree()`, that returns the tree of action and message types. This allows for easier testing of action structure.
- `eliot.testing.LoggedAction.of_type` now accepts the type as a string, not just an `eliot.ActionType` instance. Similarly, `LoggedMessage.of_type` also accepts the type as a string.

3.3.9 1.5.0

Bug fixes:

- The standard library logging bridge now logs tracebacks, not just messages.

Features:

- You can now pass in an explicit traceback tuple to `write_traceback`.

Changes:

- The deprecated `system` argument to `write_traceback` and `writeFailure` has been removed.

3.3.10 1.4.0

Features:

- Added support for routing standard library logging into Eliot; see *Route existing logs to Eliot* for details.
- Added support for Python 3.7.

Output format changes:

- All messages now have either `message_type` or `action_type` fields.

Documentation:

- Documented how to add log levels, and how to filter Eliot logs.
- Logstash configuration is closer to modern version's options, though still untested.
- Explained how to integrate/migrate existing logging with Eliot.

3.3.11 1.3.0

Features:

- The default JSON output format now supports custom JSON encoders. See *Customizing JSON Encoding* for details. Thanks to Jonathan Jacobs for feedback.

Bug fixes:

- `MemoryLogger.validate()` now gives more informative errors if JSON encoding fails. Thanks to Jean-Paul Calderone for the bug report.

Deprecations:

- On Python 3, the JSON encoder used by `to_file` and `FileDestination` would accept `bytes...` sometimes. This is deprecated, and will cease to work in a future release of Eliot (on Python 3, it will continue to work on Python 2). If you wish to include `bytes` in JSON logging, convert it to a string in the log-generating code, use Eliot's type system, or use a custom JSON encoder.

3.3.12 1.2.0

Features:

- Eliot now does the right thing for `asyncio` coroutines in Python 3.5 and later. See *Asyncio/Trio Coroutine Support* for details. Thanks to x0zzz for the bug report.

Misc:

- `Action.continue_task` can now accept text task IDs (`str` in Python 3, `unicode` in Python 2).

3.3.13 1.1.0

Features:

- Messages are no longer lost if they are logged before any destinations are added. In particular, messages will be buffered in memory until the first set of destinations are added, at which point those messages will be delivered. Thanks to Jean-Paul Calderone for the feature request.
- `eliot.add_destinations` replaces `eliot.add_destination`, and accepts multiple `Destinations` at once.
- `eliot.twisted.TwistedDestination` allows redirecting Eliot logs to `twisted.logger`. Thanks to Glyph Lefkowitz for the feature request.

Misc:

- Coding standard switched to PEP-8.
- Dropped support for Python 3.3.
- Dropped support for versions of Twisted older than 15.2 (or whenever it was that `twisted.logger` was introduced).
- Dropped support for `ujson`.

3.3.14 1.0.0

Eliot is stable, and has been for a while, so switching to v1.0.

Features:

- New API: `MessageType.log()`, the equivalent of `Message.log()`, allows you to quickly create a new typed log message and write it out.
- New APIs: `eliot.current_action()` returns the current `Action`, and `Action.task_uuid` is the task's UUID.
- You can now do with `YOUR_ACTION().context() as action:`, i.e. `Action.context()` context manager returns the `Action` instance.
- `ActionType.as_task` no longer requires a logger argument, matching the other APIs where passing in a logger is optional.

3.3.15 0.12.0

Features:

- Python 3.6 support.

Misc:

- Made test suite pass again with latest Hypothesis release.

3.3.16 0.11.0

Features:

- Eliot tasks can now more easily *span multiple threads* using the new `eliot.preserve_context` API.
- `eliot-prettyprint` command line tool now pretty prints field values in a more informative manner.

Bug fixes:

- `eliot-prettyprint` now handles unparseable lines by skipping formatting them rather than exiting.

3.3.17 0.10.1

Bug fixes:

- Fixed regression in 0.10.0: fix validation of failed actions and tracebacks with extracted additional fields.

3.3.18 0.10.0

Features:

- `register_exception_extractor` allows for more useful *logging of failed actions and tracebacks* by extracting additional fields from exceptions.
- Python 3.5 support.

Bug fixes:

- Journald support works on Python 3.

3.3.19 0.9.0

Features:

- Native *journald* support.
- `eliot-prettyprint` is a command-line tool that formats JSON Eliot messages into a more human-friendly format.
- `eliot.logwriter.ThreadedWriter` is a Twisted non-blocking wrapper for any blocking destination.

3.3.20 0.8.0

Features:

- `Message.log` will log a new message, combining the existing `Message.new` and `Message.write`.
- `write_traceback` and `writeFailure` no longer require a `Logger`; they now default to using the global one.
- The logs written with `redirectLogsForTrial` are now written in JSON format, rather than with `pformat`.

Bug fixes:

- `FileDestination` will now call `flush()` on the given file object after writing the log message. Previously log messages would not end up being written out until the file buffer filled up.
- Each `Message` logged outside the context of an action now gets a unique `task_id`.

3.3.21 0.7.0

- Creating your own `Logger` instances is no longer necessary; all relevant APIs now default to using a global one. A new testing decorator (`eliot.testing.capture_logging`) was added to capture global logging.
- Support positional `Field`-instance arguments to `fields()` to make combining existing field types and simple fields more convenient. Contributed by Jonathan Jacobs.
- `write_traceback` and `writeFailure` no longer require a `system` argument, as the combination of traceback and action context should suffice to discover the origin of the problem. This is a minor change to output format as the field is also omitted from the resulting `eliot:traceback` messages.
- The `validate_logging` testing utility now skips validation when the decorated test method raises `SkipTest`.
- Exceptions in destinations are now handled better: instead of being dropped silently an attempt is made to log a message about the problem. If that also fails then the exception is dropped.

3.3.22 0.6.0

Warning: Incompatible output format change! In previous versions the ordering of messages and actions was ambiguous and could not be deduced from out-of-order logs, and even where it was possible sorting correctly was difficult. To fix this the `action_counter` field was removed and now all messages can be uniquely located within a specific task by the values in an *improved task_level field*.

Features:

- Eliot tasks can now *span multiple processes and threads*, allowing for easy tracing of actions in complex and distributed applications.
- `eliot.add_global_fields` allows adding fields with specific values to all Eliot messages logged by your program. This can be used to e.g. distinguish between log messages from different processes by including relevant identifying information.

Bug fixes:

- On Python 3 files that accept unicode (e.g. `sys.stdout`) should now work.

3.3.23 0.5.0

Features:

- Added support for Python 3.4.
- Most public methods and functions now have underscore-based equivalents to the camel case versions, e.g. `eliot.write_traceback` and `eliot.writeTraceback`, for use in PEP 8 styled programs. Twisted-facing APIs and pyunit assertions do not provide these additional APIs, as camel-case is the native idiom.
- `eliot.to_file` outputs log messages to a file.
- Documented how to load Eliot logging into Elasticsearch via Logstash.
- Documentation has been significantly reorganized.

3.3.24 0.4.0

Note that this is the last release that will make incompatible API changes without interim deprecation warnings.

Incompatible changes from 0.3.0:

- `Logger` no longer does JSON serialization; it's up to destinations to decide how to serialize the dictionaries they receive.
- Timestamps are no longer encoded in TAI64N format; they are now provided as seconds since the Unix epoch.
- `ActionType` no longer supports defining additional failure fields, and therefore accepts one argument less.
- `Action.runCallback` and `Action.finishAfter` have been removed, as they are replaced by `DeferredContext` (see below).

Features:

- Added a simpler API (`fields()`) for defining fields for `ActionType` and `MessageType`.
- Added support for Python 3.3.
- Actions can now be explicitly finished using a public API: `Action.finish()`.
- `Action.context()` context manager allows setting an action context without finishing the action when exiting the block.
- Added a new API for Twisted Deferred support: `eliot.twisted.DeferredContext`.
- `eliot.twisted.redirectLogsForTrial` will redirect Eliot logs to Twisted's logs when running under the `trial` test runner.

3.4 Generating Logs

3.4.1 Actions and Tasks

Actions: A Start and a Finish

A higher-level construct than messages is the concept of an action. An action can be started, and then finishes either successfully or with some sort of an exception. Success in this case simply means no exception was thrown; the result of an action may be a successful response saying “this did not work”. Log messages are emitted for action start and finish.

Actions are also nested; one action can be the parent of another. An action’s parent is deduced from the Python call stack and context managers like `Action.context()`. Log messages will also note the action they are part of if they can deduce it from the call stack. The result of all this is that you can trace the operation of your code as it logs various actions, and see a narrative of what happened and what caused it to happen.

Logging Actions

Here’s a basic example of logging an action:

```
from eliot import start_action

with start_action(action_type=u"store_data"):
    x = get_data()
    store_data(x)
```

This will log an action start message and if the block finishes successfully an action success message. If an exception is thrown by the block then an action failure message will be logged along with the exception type and reason as additional fields. Each action thus results in two messages being logged: at the start and finish of the action. No traceback will be logged so if you want a traceback you will need to do so explicitly. Notice that the action has a name, with a subsystem prefix. Again, this should be a logical name.

Note that all code called within this block is within the context of this action. While running the block of code within the `with` statement new actions created with `start_action` will get the top-level `start_action` as their parent.

Logging Functions

If you want to log the inputs and results of a function, you can use the `log_call` decorator:

```
from eliot import log_call

@log_call
def calculate(x, y):
    return x * y
```

This will log an action of type `calculate` with arguments `x` and `y`, as well as logging the result. You can also customize the output:

```
from eliot import log_call

@log_call(action_type="CALC", include_args=["x"], include_result=False)
def calculate(x, y):
    return x * y
```

This changes the action type to `CALC`, logs only the `x` argument, and doesn’t log the result.

Tasks: Top-level Actions

A top-level action with no parent is called a task, the root cause of all its child actions. E.g. a web server receiving a new HTTP request would create a task for that new request. Log messages emitted from Eliot are therefore logically structured as a forest: trees of actions with tasks at the root. If you want to ignore the context and create a top-level task you can use the `eliot.start_task` API.

From Actions to Messages

While the logical structure of log messages is a forest of actions, the actual output is effectively a list of dictionaries (e.g. a series of JSON messages written to a file). To bridge the gap between the two structures each output message contains special fields expressing the logical relationship between it and other messages:

- `task_uuid`: The unique identifier of the task (top-level action) the message is part of.
- `task_level`: The specific location of this message within the task's tree of actions. For example, `[3, 2, 4]` indicates the message is the 4th child of the 2nd child of the 3rd child of the task.

Consider the following code sample:

```
from eliot import start_action, start_task

with start_task(action_type="parent") as action:
    action.log(message_type="info", x=1)
    with start_action(action_type="child") as action:
        action.log(message_type="info", x=2)
    raise RuntimeError("ono")
```

All these messages will share the same UUID in their `task_uuid` field, since they are all part of the same high-level task. If you sort the resulting messages by their `task_level` you will get the tree of messages:

```
task_level=[1] action_type="parent" action_status="started"
task_level=[2] message_type="info" x=1
    task_level=[3, 1] action_type="child" action_status="started"
    task_level=[3, 2] message_type="info" x=2
    task_level=[3, 3] action_type="child" action_status="succeeded"
task_level=[4] action_type="parent" action_status="failed" exception="exceptions.
↳RuntimeError" reason="ono"
```

Action Fields

You can add fields to both the start message and the success message of an action.

```
from eliot import start_action

with start_action(action_type="yourapp:subsystem:frob",
                  # Fields added to start message only:
                  key=123, foo="bar") as action:
    x = _beep(123)
    result = frobinate(x)
    # Fields added to success message only:
    action.add_success_fields(result=result)
```

If you want to include some extra information in case of failures beyond the exception you can always log a regular message with that information. Since the message will be recorded inside the context of the action its information will be clearly tied to the result of the action by the person (or code!) reading the logs later on.

Using Generators

Generators (functions with `yield`) and context managers (`with X:`) don't mix well in Python. So if you're going to use `with start_action()` in a generator, just make sure it doesn't wrap a `yield` and you'll be fine.

Here's what you SHOULD NOT DO:

```
def generator():
    with start_action(action_type="x"):
        # BAD! DO NOT yield inside a start_action() block:
        yield make_result()
```

Here's what can do instead:

```
def generator():
    with start_action(action_type="x"):
        result = make_result()
    # This is GOOD, no yield inside the start_action() block:
    yield result
```

Non-Finishing Contexts

Sometimes you want to have the action be the context for other messages but not finish automatically when the block finishes. You can do so with `Action.context()`. You can explicitly finish an action by calling `eliot.Action.finish`. If called with an exception it indicates the action finished unsuccessfully. If called with no arguments it indicates that the action finished successfully.

```
from eliot import start_action

action = start_action(action_type=u"yourapp:subsystem:frob")
try:
    with action.context():
        x = _beep()
    with action.context():
        frobinate(x)
    # Action still isn't finished, need to so explicitly.
except FrobError as e:
    action.finish(e)
else:
    action.finish()
```

The `context()` method returns the `Action`:

```
from eliot import start_action

with start_action(action_type=u"your_type").context() as action:
    # do some stuff...
    action.finish()
```

You shouldn't log within an action's context after it has been finished:

```
from eliot import start_action

with start_action(action_type=u"message_late").context() as action:
    action.log(message_type=u"ok")
    # finish the action:
```

(continues on next page)

(continued from previous page)

```
action.finish()  
# Don't do this! This message is being added to a finished action!  
action.log(message_type="late")
```

As an alternative to `with`, you can also explicitly run a function within the action context:

```
from eliot import start_action  
  
action = start_action(action_type="yourapp:subsystem:frob")  
# Call do_something(x=1) in context of action, return its result:  
result = action.run(do_something, x=1)
```

Getting the Current Action

Sometimes it can be useful to get the current action. For example, you might want to record the current task UUID for future reference, in a bug report for example. You might also want to pass around the `Action` explicitly, rather than relying on the implicit context.

You can get the current `Action` by calling `eliot.current_action()`. For example:

```
from eliot import current_action  
  
def get_current_uuid():  
    return current_action().task_uuid
```

3.4.2 Messages

Sometimes you don't want to generate actions. sometimes you just want an individual isolated message, the way traditional logging systems work. Here's how to do that.

When you have an action

If you already have an action object, you can log a message in that action's context:

```
from eliot import start_action  
  
class YourClass(object):  
    def run(self):  
        with start_action(action_type="myaction") as ctx:  
            ctx.log(message_type="mymessage", key="abc", key2=4)
```

If you don't have an action

If you don't have a reference to an action, or you're worried the function will sometimes be called outside the context of any action at all, you can use `log_message`:

```
from eliot import log_message  
  
def run(x):  
    log_message(message_type="in_run", xfield=x)
```

The main downside to using this function is that it's a little slower, since it needs to handle the case where there is no action in context.

3.4.3 Errors and Exceptions

Exceptions and Tracebacks

If you are using actions you don't need to do anything special to log exceptions: if an exception is thrown within the context of an action and not caught, the action will be marked as failed and the exception will be logged with it.

If you get a completely unexpected exception you may wish to log a traceback to aid debugging:

```
from eliot import write_traceback

class YourClass(object):

    def run(self):
        try:
            dosomething()
        except:
            write_traceback()
```

You can also pass in the output of `sys.exc_info()`:

```
import sys
from eliot import write_traceback

write_traceback(exc_info=sys.exc_info())
```

Custom Exception Logging

By default both failed actions and tracebacks log the class and string-representation of the logged exception. You can add additional fields to these messages by registering a callable that converts exceptions into fields. If no extraction function is registered for a class Eliot will look for registered functions for the exception's base classes.

For example, the following registration means all failed actions that fail with a `MyException` will have a `code` field in the action end message, as will tracebacks logged with this exception:

```
class MyException(Exception):
    def __init__(self, code):
        self.code = code

from eliot import register_exception_extractor
register_exception_extractor(MyException, lambda e: {"code": e.code})
```

By default Eliot will automatically extract fields from `OSError`, `IOError` and other subclasses of Python's `EnvironmentError`.

3.4.4 Log Levels

Eliot does not have a native set of logging levels, as some systems do. It does distinguish between normal log messages and errors—failed actions and tracebacks can both be considered as errors.

However, you can add log levels yourself.

Generating messages with log levels

All you need to do to add a log level is just add an appropriate field to your logging, for example:

```
from eliot import start_action

with start_action(action_type=u"store_data", log_level="INFO"):
    x = get_data()
    store_data(x)
```

Choosing log levels

In an excellent [article](#) by Daniel Lebroro, he explains that he chose the logging levels “for test environment”, “for production environment”, “investigate tomorrow”, and “wake me in the middle of the night”. These seem rather more informative and useful than “INFO” or “WARN”.

If you are implementing a service you will be running, consider choosing log levels that are meaningful on an organizational level.

3.4.5 Integrating and Migrating Existing Logging

If you have an existing code base, you likely have existing log messages. This document will explain how to migrate and integrate existing logging into your new Eliot log setup. In particular, this will focus on the Python standard library logging package, but the same principles apply to other logging libraries.

Route existing logs to Eliot

Eliot includes a `logging.Handler` that can take standard library log messages and route them into Eliot. These log messages will *automatically* appear in the correct place in the action tree! Once you add actions to your code these log messages will automatically benefit from Eliot’s causal information.

To begin with, however, we’ll just add routing of log messages to Eliot:

```
# Add Eliot Handler to root Logger. You may wish to only route specific
# Loggers to Eliot.
import logging
from eliot.stdlib import EliotHandler
logging.getLogger().addHandler(EliotHandler())
```

Add actions at entry points and other key points

Simply by adding a few key actions—the entry points to the code, as well as key sub-actions—you can start getting value from Eliot’s functionality while still getting information from your existing logs. You can leave existing log messages in place, replacing them with Eliot logging opportunistically; they will still be included in your output.

```
"""
Example of routing standard library logging to Eliot.

The assumption is you have legacy logging using stdlib, and are switching over
to Eliot.
"""
```

(continues on next page)

(continued from previous page)

```

import logging
import sys

from eliot.stdlib import EliotHandler
from eliot import start_action, to_file

# A Logger left over from before switch to Eliot
LEGACY_LOGGER = logging.Logger("mypackage")

def do_a_thing(i):
    with start_action(action_type="mypackage:do_a_thing"):
        # run your business logic...
        if i == 3:
            LEGACY_LOGGER.error("The number 3 is a bad number, don't use it.")
            raise ValueError("I hate the number 3")

def main():
    with start_action(action_type="mypackage:main"):
        for i in [1, 3]:
            try:
                do_a_thing(i)
            except ValueError:
                LEGACY_LOGGER.info("Number {} was rejected.".format(i))

if __name__ == '__main__':
    # Hook up stdlib logging to Eliot:
    LEGACY_LOGGER.addHandler(EliotHandler())
    # Write Eliot logs to stdout:
    to_file(sys.stdout)
    # Run the code:
    main()

```

The stdlib logging messages will be included in the correct part of the tree:

```

$ python examples/stdlib.py | eliot-tree
3f465ee3-7fa9-40e2-8b20-9c0595612a8b
├─ mypackage:main/1 started
│   └─ timestamp: 2018-07-15 16:50:39.230467
│       └─ mypackage:do_a_thing/2/1 started
│           └─ timestamp: 2018-07-15 16:50:39.230709
│               └─ mypackage:do_a_thing/2/2 succeeded
│                   └─ timestamp: 2018-07-15 16:50:39.230836
│                       └─ mypackage:do_a_thing/3/1 started
│                           └─ timestamp: 2018-07-15 16:50:39.230980
│                               └─ eliot:stdlib/3/2
│                                   └─ log_level: ERROR
│                                       └─ logger: mypackage
│                                           └─ message: The number 3 is a bad number, don't use it.
│                                               └─ timestamp: 2018-07-15 16:50:39.231157
│                                                   └─ mypackage:do_a_thing/3/3 failed
│                                                       └─ exception: builtins.ValueError
│                                                           └─ reason: I hate the number 3
│                                                               └─ timestamp: 2018-07-15 16:50:39.231364
└─ eliot:stdlib/4

```

(continues on next page)

(continued from previous page)

```

├── log_level: INFO
├── logger: mypackage
├── message: Number 3 was rejected.
├── timestamp: 2018-07-15 16:50:39.231515
└── mypackage:main/5 succeeded
    ├── timestamp: 2018-07-15 16:50:39.231641

```

3.4.6 Spanning Processes and Threads

Introduction

In many applications we are interested in tasks that exist in more than just a single thread or in a single process. For example, one server may send a request to another server over a network and we would like to trace the combined operation across both servers' logs. To make this as easy as possible Eliot supports serializing task identifiers for transfer over the network (or between threads), allowing tasks to span multiple processes.

Cross-Thread Tasks

To trace actions across threads Eliot provides the `eliot.preserve_context` API. It takes a callable that is about to be passed to a thread constructor and preserves the current Eliot context, returning a new callable. This new callable should only be used, in the thread where it will run; it will restore the Eliot context and run the original function inside of it. For example:

```

#!/usr/bin/env python

"""
Example of an Eliot action context spanning multiple threads.
"""

from __future__ import unicode_literals

from threading import Thread
from sys import stdout

from eliot import to_file, preserve_context, start_action
to_file(stdout)

def add_in_thread(x, y):
    with start_action(action_type="in_thread", x=x, y=y) as context:
        context.add_success_fields(result=x+y)

with start_action(action_type="main_thread"):
    # Preserve Eliot context and restore in new thread:
    thread = Thread(target=preserve_context(add_in_thread),
                    kwargs={"x": 3, "y": 4})
    thread.start()
    # Wait for the thread to exit:
    thread.join()

```

Here's what the result is when run:

```
$ python examples/cross_thread.py | eliot-tree
11a85c42-a13f-491c-ad44-c48b2efad0e3
+-- main_thread@1/started
  +-- eliot:remote_task@2,1/started
    +-- in_thread@2,2,1/started
      |-- x: 3
      |-- y: 4
    +-- in_thread@2,2,2/succeeded
      |-- result: 7
    +-- eliot:remote_task@2,3/succeeded
  +-- main_thread@3/succeeded
```

Cross-Process Tasks

`eliot.Action.serialize_task_id()` can be used to create some bytes identifying a particular location within a task. `eliot.Action.continue_task()` converts a serialized task identifier into an `eliot.Action` and then starts the `Action`. The process which created the task serializes the task identifier and sends it over the network to the process which will continue the task. This second process deserializes the identifier and uses it as a context for its own messages.

In the following example the task identifier is added as a header to a HTTP request:

```
"""
Cross-process log tracing: HTTP client.
"""
from __future__ import unicode_literals

import sys
import requests

from eliot import to_file, start_action, add_global_fields
add_global_fields(process="client")
to_file(sys.stdout)

def remote_divide(x, y):
    with start_action(action_type="http_request", x=x, y=y) as action:
        task_id = action.serialize_task_id()
        response = requests.get(
            "http://localhost:5000/?x={}&y={}".format(x, y),
            headers={"x-eliot-task-id": task_id})
        response.raise_for_status() # ensure this is a successful response
        result = float(response.text)
        action.add_success_fields(result=result)
    return result

if __name__ == '__main__':
    with start_action(action_type="main"):
        remote_divide(int(sys.argv[1]), int(sys.argv[2]))
```

The server that receives the request then extracts the identifier:

```
"""
Cross-process log tracing: HTTP server.
"""
```

(continues on next page)

(continued from previous page)

```

from __future__ import unicode_literals

import sys
from flask import Flask, request

from eliot import to_file, Action, start_action, add_global_fields
add_global_fields(process="server")
to_file(sys.stdout)

app = Flask("server")

def divide(x, y):
    with start_action(action_type="divide", x=x, y=y) as action:
        result = x / y
        action.add_success_fields(result=result)
        return result

@app.route("/")
def main():
    with Action.continue_task(task_id=request.headers["x-eliot-task-id"]):
        x = int(request.args["x"])
        y = int(request.args["y"])
        return str(divide(x, y))

if __name__ == '__main__':
    app.run()

```

Tracing logs across multiple processes makes debugging problems dramatically easier. For example, let's run the following:

```

$ python examples/cross_process_server.py > server.log
$ python examples/cross_process_client.py 5 0 > client.log

```

Here are the resulting combined logs, as visualized by `eliot-tree`. The reason the client received a 500 error code is completely obvious in these logs:

```

$ cat client.log server.log | eliot-tree
1e0be9be-ae56-49ef-9bce-60e850a7db09
+-- main@1/started
  |-- process: client
  +-- http_request@2,1/started
    |-- process: client
    |-- x: 3
    |-- y: 0
    +-- eliot:remote_task@2,2,1/started
      |-- process: server
      +-- divide@2,2,2,1/started
        |-- process: server
        |-- x: 3
        |-- y: 0
        +-- divide@2,2,2,2/failed
          |-- exception: exceptions.ZeroDivisionError

```

(continues on next page)

(continued from previous page)

```

        |-- process: server
        |-- reason: integer division or modulo by zero
+-- eliot:remote_task@2,2,3/failed
    |-- exception: exceptions.ZeroDivisionError
    |-- process: server
    |-- reason: integer division or modulo by zero
+-- http_request@2,3/failed
    |-- exception: requests.exceptions.HTTPError
    |-- process: client
    |-- reason: 500 Server Error: INTERNAL SERVER ERROR
+-- main@3/failed
    |-- exception: requests.exceptions.HTTPError
    |-- process: client
    |-- reason: 500 Server Error: INTERNAL SERVER ERROR

```

Cross-Thread Tasks

`eliot.Action` objects should only be used on the thread that created them. If you want your task to span multiple threads use the API described above.

Ensuring Message Uniqueness

Serialized task identifiers should be used at most once. For example, every time a remote operation is retried a new call to `serialize_task_id()` should be made to create a new identifier. Otherwise there is a chance that you will end up with messages that have duplicate identification (i.e. two messages with matching `task_uuid` and `task_level` values), making it more difficult to trace causality.

If this is not possible you may wish to start a new Eliot task upon receiving a remote request, while still making sure to log the serialized remote task identifier. The inclusion of the remote task identifier will allow manual or automated reconstruction of the cross-process relationship between the original and new tasks.

Another alternative in some cases is to rely on unique process or thread identity to distinguish between the log messages. For example if the same serialized task identifier is sent to multiple processes, log messages within the task can still have a unique identity if a process identifier is included with each message.

Logging Output for Multiple Processes

If logs are being combined from multiple processes an identifier indicating the originating process should be included in log messages. This can be done a number of ways, e.g.:

- Have your destination add another field to the output.
- Rely on Logstash, or whatever your logging pipeline tool is, to add a field when shipping the logs to your centralized log store.

3.4.7 Unit Testing Your Logging

Now that you've got some code emitting log messages (or even better, before you've written the code) you can write unit tests to verify it. Given good test coverage all code branches should already be covered by tests unrelated to logging. Logging can be considered just another aspect of testing those code branches.

Rather than recreating all those tests as separate functions Eliot provides a decorator the allows adding logging assertions to existing tests.

Linting your logs

Decorating a test function with `eliot.testing.capture_logging` validation will ensure that:

1. You haven't logged anything that isn't JSON serializable.
2. There are no unexpected tracebacks, indicating a bug somewhere in your code.

```
from eliot.testing import capture_logging

class MyTest(unittest.TestCase):
    @capture_logging(None)
    def test_mytest(self, logger):
        call_my_function()
```

Making assertions about the logs

You can also ensure the correct messages were logged.

```
from eliot import log_message

class UserRegistration(object):

    def __init__(self):
        self.db = {}

    def register(self, username, password, age):
        self.db[username] = (password, age)
        log_message(message_type="user_registration",
                    username=username, password=password,
                    age=age)
```

Here's how we'd test it:

```
from unittest import TestCase
from eliot import MemoryLogger
from eliot.testing import assertContainsFields, capture_logging

from myapp.registration import UserRegistration

class LoggingTests(TestCase):
    def assertRegistrationLogging(self, logger):
        """
        Logging assertions for test_registration.
        """
        self.assertEqual(len(logger.messages), 1)
        msg = logger.messages[0]
        assertContainsFields(self, msg,
                            {"username": u"john",
                             "password": u"password",
                             "age": 12})

    @capture_logging(assertRegistrationLogging)
    def test_registration(self, logger):
        """
        Registration adds entries to the in-memory database.
        """
```

(continues on next page)

(continued from previous page)

```

"""
registry = UserRegistration()
registry.register(u"john", u"password", 12)
self.assertEqual(registry.db[u"john"], (u"password", 12))

```

Testing Tracebacks

Tests decorated with `@capture_logging` will fail if there are any tracebacks logged (using `write_traceback` or `writeFailure`) on the theory that these are unexpected errors indicating a bug. If you expected a particular traceback to be logged you can call `MemoryLogger.flush_tracebacks`, after which it will no longer cause a test failure. The result will be a list of traceback message dictionaries for the particular exception.

```

from unittest import TestCase
from eliot.testing import capture_logging

class MyTests(TestCase):
    def assertMythingBadPathLogging(self, logger):
        messages = logger.flush_tracebacks(OSError)
        self.assertEqual(len(messages), 1)

    @capture_logging(assertMythingBadPathLogging)
    def test_mythingBadPath(self, logger):
        mything = MyThing()
        # Trigger an error that will cause a OSError traceback to be logged:
        self.assertFalse(mything.load("/nonexistent/path"))

```

Testing Message and Action Structure

Eliot provides utilities for making assertions about the structure of individual messages and actions. The simplest method is using the `assertHasMessage` utility function which asserts that a message of a given message type has the given fields:

```

from eliot.testing import assertHasMessage, capture_logging

class LoggingTests(TestCase):
    @capture_logging(assertHasMessage, "user_registration",
                    {u"username": u"john",
                     u"password": u"password",
                     u"age": 12})
    def test_registration(self, logger):
        """
        Registration adds entries to the in-memory database.
        """
        registry = UserRegistration()
        registry.register(u"john", u"password", 12)
        self.assertEqual(registry.db[u"john"], (u"password", 12))

```

`assertHasMessage` returns the found message and can therefore be used within more complex assertions. `assertHasAction` provides similar functionality for actions (see example below).

More generally, `eliot.testing.LoggedAction` and `eliot.testing.LoggedMessage` are utility classes to aid such testing. `LoggedMessage.of_type` lets you find all messages of a specific message type. A `LoggedMessage` has an attribute `message` which contains the logged message dictionary. For example, we could rewrite the registration logging test above like so:

```

from eliot.testing import LoggedMessage, capture_logging

class LoggingTests(TestCase):
    def assertRegistrationLogging(self, logger):
        """
        Logging assertions for test_registration.
        """
        logged = LoggedMessage.of_type(logger.messages, "user_registration")[0]
        assertContainsFields(self, logged.message,
                             {"username": u"john",
                              u"password": u"password",
                              u"age": 12})

    @capture_logging(assertRegistrationLogging)
    def test_registration(self, logger):
        """
        Registration adds entries to the in-memory database.
        """
        registry = UserRegistration()
        registry.register(u"john", u"password", 12)
        self.assertEqual(registry.db[u"john"], (u"password", 12))

```

Similarly, `LoggedAction.of_type` finds all logged actions of a specific action type. A `LoggedAction` instance has `start_message` and `end_message` containing the respective message dictionaries, and a `children` attribute containing a list of child `LoggedAction` and `LoggedMessage`. That is, a `LoggedAction` knows about the messages logged within its context. `LoggedAction` also has a utility method `descendants()` that returns an iterable of all its descendants. We can thus assert that a particular message (or action) was logged within the context of another action.

For example, let's say we have some code like this:

```

from eliot import start_action, Message

class Search:
    def search(self, servers, database, key):
        with start_action(action_type="log_search", database=database, key=key):
            for server in servers:
                Message.log(message_type="log_check", server=server)
                if server.check(database, key):
                    return True
        return False

```

We want to assert that the “log_check” message was written in the context of the “log_search” action. The test would look like this:

```

from eliot.testing import LoggedAction, LoggedMessage, capture_logging
import searcher

class LoggingTests(TestCase):
    @capture_logging(None)
    def test_logging(self, logger):
        searcher = Search()
        servers = [buildServer(), buildServer()]

        searcher.search(servers, "users", "theuser")
        action = LoggedAction.of_type(logger.messages, "log_search")[0]
        messages = LoggedMessage.of_type(logger.messages, "log_check")

```

(continues on next page)

(continued from previous page)

```

# The action start message had the appropriate fields:
assertContainsFields(self, action.start_message,
                    {"database": "users", "key": "theuser"})
# Messages were logged in the context of the action
self.assertEqual(action.children, messages)
# Each message had the respective server set.
self.assertEqual(servers, [msg.message["server"] for msg in messages])

```

Or we can simplify further by using `assertHasMessage` and `assertHasAction`:

```

from eliot.testing import LoggedAction, LoggedMessage, capture_logging
import searcher

class LoggingTests(TestCase):
    @capture_logging(None)
    def test_logging(self, logger):
        searcher = Search()
        servers = [buildServer(), buildServer()]

        searcher.search(servers, "users", "theuser")
        action = assertHasAction(self, logger, "log_search", succeeded=True,
                                startFields={"database": "users",
                                             "key": "theuser"})

        # Messages were logged in the context of the action
        messages = LoggedMessage.of_type(logger.messages, "log_check")
        self.assertEqual(action.children, messages)
        # Each message had the respective server set.
        self.assertEqual(servers, [msg.message["server"] for msg in messages])

```

Custom JSON encoding

Just like a `FileDestination` can have a custom JSON encoder, so can your tests, so you can validate your messages with that JSON encoder:

```

from unittest import TestCase
from eliot.json import EliotJSONEncoder
from eliot.testing import capture_logging

class MyClass:
    def __init__(self, x):
        self.x = x

class MyEncoder(EliotJSONEncoder):
    def default(self, obj):
        if isinstance(obj, MyClass):
            return {"x": obj.x}
        return EliotJSONEncoder.default(self, obj)

class LoggingTests(TestCase):
    @capture_logging(None, encoder_=MyEncoder)
    def test_logging(self, logger):
        # Logged messages will be validated using MyEncoder....
        ...

```

Notice that the hyphen after `encoder_` is deliberate: by default keyword arguments are passed to the assertion function (the first argument to `@capture_logging`) so it's marked this way to indicate it's part of Eliot's API.

Custom testing setup

In some cases `@capture_logging` may not do what you want. You can achieve the same effect, but with more control, with some lower-level APIs:

```
from eliot import MemoryLogger
from eliot.testing import swap_logger, check_for_errors

def custom_capture_logging():
    # Replace default logging setup with a testing logger:
    test_logger = MemoryLogger()
    original_logger = swap_logger(test_logger)

    try:
        run_some_code()
    finally:
        # Restore original logging setup:
        swap_logger(original_logger)
        # Validate log messages, check for tracebacks:
        check_for_errors(test_logger)
```

3.4.8 Using Types to Structure Messages and Actions

Why Typing?

So far we've been creating messages and actions in an unstructured manner. This means it's harder to support Python objects that aren't built-in and to validate message structure. Moreover there's no documentation of what fields messages and action messages expect.

To improve this we introduce an optional API for creating actions and standalone messages: `ActionType` and `MessageType`. Here's an example demonstrating how we create a message type, bind some values and then log the message:

```
from eliot import Field, MessageType

class Coordinate(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

# This field takes a complex type that will be stored in a single Field,
# so we pass in a serializer function that converts it to a list with two
# ints:
_LOCATION = Field(u"location", lambda loc: [loc.x, loc.y], u"The location.")
# These fields are just basic supported types, in this case int and unicode
# respectively:
_COUNT = Field.for_types(u"count", [int], u"The number of items to deliver.")
_NAME = Field.for_types(u"name", [unicode], u"The name of the delivery person.")

# This is a type definition for a message. It is used to hook up
# serialization of field values, and for message validation in unit tests:
```

(continues on next page)

(continued from previous page)

```

LOG_DELIVERY_SCHEDULED = MessageType(
    u"pizzadelivery:schedule",
    [_LOCATION, _COUNT, _NAME],
    u"A pizza delivery has been scheduled.")

def deliver_pizzas(deliveries):
    person = get_free_delivery_person()
    for location, count in deliveries:
        delivery_database.insert(person, location, count)
        LOG_DELIVERY_SCHEDULED.log(
            name=person.name, count=count, location=location)

```

Fields

A `Field` instance is used to validate fields of messages, and to serialize rich types to the built-in supported types. It is created with the name of the field, a serialization function that converts the input to an output and a description. The serialization function must return a result that is JSON-encodable. You can also pass in an extra validation function. If you pass this function in it will be called with values that are being validated; if it raises `eliot.ValidationError` that value will fail validation.

A couple of utility functions allow creating specific types of `Field` instances. `Field.for_value` returns a `Field` that only can have a single value. More generally useful, `Field.for_types` returns a `Field` that can only be one of certain specific types: some subset of unicode, bytes, int, float, bool, list and dict as well as None which technically isn't a class. As always, bytes must only contain UTF-8 encoded Unicode.

```

from eliot import Field

def userToUsername(user):
    """
    Extract username from a User object.
    """
    return user.username

USERNAME = Field(u"username", userToUsername, u"The name of the user.")

# Validation is useful for unit tests and catching bugs; it's not used in
# the actual logging code path. We therefore don't bother catching things
# we'd do in e.g. web form validation.
def _validateAge(value):
    if value is not None and value < 0:
        raise ValidationError("Field 'age' must be positive:", value)
AGE = Field.for_types(u"age", [int, None],
    u"The age of the user, might be None if unknown",
    _validateAge)

```

Message Types

Now that you have some fields you can create a custom `MessageType`. This takes a message name which will be put in the `message_type` field of resulting messages. It also takes a list of `Field` instances and a description.

```

from eliot import MessageType, Field
USERNAME = Field.for_types("username", [str])

```

(continues on next page)

(continued from previous page)

```
AGE = Field.for_types("age", [int])

LOG_USER_REGISTRATION = MessageType(u"yourapp:authentication:registration",
                                     [USERNAME, AGE],
                                     u"We've just registered a new user.")
```

Since this syntax is rather verbose a utility function called `fields` is provided which creates a list of `Field` instances for you, with support to specifying the types of the fields. The equivalent to the code above is:

```
from eliot import MessageType, fields

LOG_USER_REGISTRATION = MessageType(u"yourapp:authentication:registration",
                                     fields(username=str, age=int))
```

Or you can even use existing `Field` instances with `fields`:

```
from eliot import MessageType, Field, fields

USERNAME = Field.for_types("username", [str])

LOG_USER_REGISTRATION = MessageType(u"yourapp:authentication:registration",
                                     fields(USERNAME, age=int))
```

Given a `MessageType` you can create a `Message` instance with the `message_type` field pre-populated by calling the type. You can then use it the way you would normally use `Message`, e.g. `bind()` or `write()`. You can also just call `MessageType.log()` to write out a message directly:

```
# Simple version:
LOG_USER_REGISTRATION.log(username=user, age=193)
# Equivalent more complex API:
LOG_USER_REGISTRATION(username=user).bind(age=193).write()
```

A `Message` created from a `MessageType` will automatically use the `MessageType` `Field` instances to serialize its fields.

Keep in mind that no validation is done when messages are created. Instead, validation is intended to be done in your unit tests. If you're not unit testing all your log messages you're doing it wrong. Luckily, Eliot makes it pretty easy to test logging as we'll see in a bit.

Action Types

Similarly to `MessageType` you can also create types for actions. Unlike a `MessageType` you need two sets of fields: one for action start, one for success.

```
from eliot import ActionType, fields

LOG_USER_SIGNIN = ActionType(u"yourapp:authentication:signin",
                             # Start message fields:
                             fields(username=str),
                             # Success message fields:
                             fields(status=int),
                             # Description:
                             u"A user is attempting to sign in.")
```

Calling the resulting instance is equivalent to `start_action`. For `start_task` you can call `LOG_USER_SIGNIN.as_task`.

```
def signin(user, password):
    with LOG_USER_SIGNIN(username=user) as action:
        status = user.authenticate(password)
        action.add_success_fields(status=status)
    return status
```

Again, as with `MessageType`, field values will be serialized using the `Field` definitions in the `ActionType`.

Serialization Errors

While validation of field values typically only happens when unit testing, serialization must run in the normal logging code path. Eliot tries to very hard never to raise exceptions from the log writing code path so as not to prevent actual code from running. If a message fails to serialize then a `eliot:traceback` message will be logged, along with a `eliot:serialization_failure` message with an attempt at showing the message that failed to serialize.

```
{ "exception": "exceptions.ValueError",
  "timestamp": "2013-11-22T14:16:51.386745Z",
  "traceback": "Traceback (most recent call last):\n  ... ValueError: invalid literal_\n↪for int() with base 10: 'hello'\n",
  "system": "eliot:output",
  "reason": "invalid literal for int() with base 10: 'hello'",
  "message_type": "eliot:traceback" }
{ "timestamp": "2013-11-22T14:16:51.386827Z",
  "message": "{u'u'message_type': u''test'', u'u'field': u''hello'', u'u'\n↪'timestamp': u''2013-11-22T14:16:51.386634Z''}",
  "message_type": "eliot:serialization_failure" }
```

Testing

The `eliot.testing.assertHasAction` and `assertHasMessage` APIs accept `ActionType` and `MessageType` instances, not just the `action_type` and `message_type` strings.

Any function decorated with `@capture_logging` will additionally validate messages that were created using `ActionType` and `MessageType` using the applicable `Field` definitions. This will ensure you've logged all the necessary fields, no additional fields, and used the correct types.

3.4.9 Asyncio/Trio Coroutine Support

As of Eliot 1.8, `asyncio` and `trio` coroutines have appropriate context propagation for Eliot, automatically.

Asyncio

On Python 3.7 or later, no particular care is needed. For Python 3.5 and 3.6 you will need to import either `eliot` (or the backport package `aioccontextvars`) before you create your first event loop.

Here's an example using `aiohttp`:

```
import asyncio
import aiohttp
from eliot import start_action, to_file
to_file(open("linkcheck.log", "w"))
```

(continues on next page)

(continued from previous page)

```

async def check_links(urls):
    session = aiohttp.ClientSession()
    with start_action(action_type="check_links", urls=urls):
        for url in urls:
            try:
                with start_action(action_type="download", url=url):
                    async with session.get(url) as response:
                        response.raise_for_status()
            except Exception as e:
                raise ValueError(str(e))

try:
    loop = asyncio.get_event_loop()
    loop.run_until_complete(
        check_links(["http://eliot.readthedocs.io", "http://nosuchurl"])
    )
except ValueError:
    print("Not all links were valid.")

```

And the resulting logs:

```

$ eliot-tree linkcheck.log
0a9a5e1b-330c-4251-b7db-fd3161403443
├─ check_links/1  started 2019-04-06 19:49:16  0.535s
│   └─ urls:
│       ├── 0: http://eliot.readthedocs.io
│       └── 1: http://nosuchurl
├─ download/2/1  started 2019-04-06 19:49:16  0.527s
│   └─ url: http://eliot.readthedocs.io
├─ download/2/2  succeeded 2019-04-06 19:49:16
├─ download/3/1  started 2019-04-06 19:49:16  0.007s
│   └─ url: http://nosuchurl
├─ download/3/2  failed 2019-04-06 19:49:16
│   ├── errno: -2
│   ├── exception: aiohttp.client_exceptions.ClientConnectorError
│   └─ reason: Cannot connect to host nosuchurl:80 ssl:None [Name or service not known]
├─ check_links/4  failed 2019-04-06 19:49:16
│   ├── exception: builtins.ValueError
│   └─ reason: Cannot connect to host nosuchurl:80 ssl:None [Name or service not known]

```

Trio

Here's an example of using Trio—we put the action outside the nursery so that it finishes only when the nursery shuts down.

```

from eliot import start_action, to_file
import trio

to_file(open("trio.log", "w"))

async def say(message, delay):

```

(continues on next page)

(continued from previous page)

```

with start_action(action_type="say", message=message):
    await trio.sleep(delay)

async def main():
    with start_action(action_type="main"):
        async with trio.open_nursery() as nursery:
            nursery.start_soon(say, "hello", 1)
            nursery.start_soon(say, "world", 2)

trio.run(main)

```

And the resulting logs:

```

$ eliot-tree trio.log
93a4de27-8c95-498b-a188-f0e91482ad10
└─ main/1  started 2019-04-10 21:07:20  2.003s
    └─ say/2/1  started 2019-04-10 21:07:20  2.002s
        └─ message: world
            └─ say/2/2  succeeded 2019-04-10 21:07:22
    └─ say/3/1  started 2019-04-10 21:07:20  1.001s
        └─ message: hello
            └─ say/3/2  succeeded 2019-04-10 21:07:21
    └─ main/4  succeeded 2019-04-10 21:07:22

```

If you put the `start_action` *inside* the nursery context manager:

1. The two `say` calls will be scheduled, but not started.
2. The parent action will end.
3. Only then will the child actions be created.

The result is somewhat confusing output. Trying to improve this situation is covered in [issue #401](#).

3.4.10 Using Eliot with Twisted

Eliot provides a variety of APIs to support integration with the [Twisted](#) networking framework.

Non-blocking Destinations

`eliot.logwriter.ThreadedWriter` is a logging destination that wraps a blocking destination and writes to it in a non-reactor thread. This is useful because it keeps the Twisted reactor from blocking, e.g. if you're writing to a log file and the hard drive is overloaded. `ThreadedWriter` is a Twisted Service and starting it will call `add_destinations` for you and stopping it will call `remove_destination`; there is no need to call those directly.

```

"""
Output an Eliot message to a log file using the threaded log writer.
"""
from __future__ import unicode_literals, print_function

from twisted.internet.task import react

from eliot.logwriter import ThreadedWriter
from eliot import log_message, FileDestination

```

(continues on next page)

(continued from previous page)

```
def main(reactor):
    print("Logging to example-eliot.log...")
    logWriter = ThreadedWriter(
        FileDestination(file=open("example-eliot.log", "ab"), reactor)

    # Manually start the service, which will add it as a
    # destination. Normally we'd register ThreadedWriter with the usual
    # Twisted Service/Application infrastructure.
    logWriter.startService()

    # Log a message:
    log_message(message_type="test", value="hello", another=1)

    # Manually stop the service.
    done = logWriter.stopService()
    return done

if __name__ == '__main__':
    react(main, [])
```

If you want log rotation you can pass in an `eliot.FileDestination` wrapping one of the classes from `twisted.python.logfile` as the destination file.

twisted.logger integration

If you wish you can direct Eliot logs to Twisted's logging subsystem, if that is the primary logging system you're using.

```
from eliot import add_destinations
from eliot.twisted import TwistedDestination

add_destinations(TwistedDestination())
```

Trial Integration

If you're using Twisted's `trial` program to run your tests you can redirect your Eliot logs to Twisted's logs by calling `eliot.twisted.redirectLogsForTrial()`. This function will automatically detect whether or not it is running under `trial`. If it is then you will be able to read your Eliot logs in `_trial_temp/test.log`, where `trial` writes out logs by default. If it is not running under `trial` it will not do anything. In addition calling it multiple times has the same effect as calling it once.

The way you use it is by putting it in your package's `__init__.py`: it will do the right thing and only redirect if you're using `trial`. Take care if you are separately redirecting Twisted logs to Eliot; you should make sure not to call `redirectLogsForTrial` in that case so as to prevent infinite loops.

Logging Failures

`eliot.writeFailure` is the equivalent of `eliot.write_traceback`, only for `Failure` instances:

```

from eliot import writeFailure

class YourClass(object):

    def run(self):
        d = dosomething()
        d.addErrback(writeFailure)

```

Actions and inlineCallbacks

Eliot provides a decorator that is compatible with Twisted's `inlineCallbacks` but which also behaves well with Eliot's actions. Simply substitute `eliot.twisted.inline_callbacks` for `twisted.internet.defer.inlineCallbacks` in your code.

To understand why, consider the following example:

```

from eliot import start_action
from twisted.internet.defer import inlineCallbacks

@inlineCallbacks # don't use this in real code, use eliot.twisted.inline_callbacks
def go():
    with start_action(action_type=u"yourapp:subsystem:frob") as action:
        d = some_deferred_api()
        x = yield d
        action.log(message_type=u"some-report", x=x)

```

The action started by this generator remains active as `yield d` gives up control to the `inlineCallbacks` controller. The next bit of code to run will be considered to be a child of `action`. Since that code may be any arbitrary code that happens to get scheduled, this is certainly wrong.

Additionally, when the `inlineCallbacks` controller resumes the generator, it will most likely do so with no active action at all. This means that the log message following the `yield` will be recorded with no parent action, also certainly wrong.

These problems are solved by using `eliot.twisted.inline_callbacks` instead of `twisted.internet.defer.inlineCallbacks`. The behavior of the two decorators is identical except that Eliot's version will preserve the generator's action context and contain it within the generator. This extends the `inlineCallbacks` illusion of "synchronous" code to Eliot actions.

Actions and Deferreds

An additional set of APIs is available to help log actions when using Deferreds. To understand why, consider the following example:

```

from eliot import start_action

def go():
    action = start_action(action_type=u"yourapp:subsystem:frob")
    with action:
        d = Deferred()
        d.addCallback(getResult, x=1)
        return d

```

This has two problems. First, `getResult` is not going to run in the context of the action. Second, the action finishes once the `with` block finishes, i.e. before `getResult` runs. If we want `getResult` to be run in the context of the

action and to delay the action finish we need to do some extra work, and manually wrapping all callbacks would be tedious.

To solve this problem you can use the `eliot.twisted.DeferredContext` class. It grabs the action context when it is first created and provides the same API as `Deferred` (`addCallbacks` and `friends`), with the difference that added callbacks run in the context of the action. When all callbacks have been added you can indicate that the action should finish after those callbacks have run by calling `DeferredContext.addActionFinish`. As you would expect, if the `Deferred` fires with a regular result that will result in success message. If the `Deferred` fires with an `errback` that will result in failure message. Finally, you can unwrap the `DeferredContext` and access the wrapped `Deferred` by accessing its `result` attribute.

```
from eliot import start_action
from eliot.twisted import DeferredContext

def go():
    with start_action(action_type="your_type").context() as action:
        d = DeferredContext(Deferred())
        # getResult(result, x=1) will be called in the context of the action:
        d.addCallback(getResult, x=1)
        # After getResult finishes, finish the action:
        d.addActionFinish()
        # Return the underlying Deferred:
        return d.result
```

3.5 Outputting Logs

3.5.1 Configuring Logging Output

You can register “destinations” to handle logging output; a destination is a callable that takes a message dictionary. For example, if we want to just print each new message:

```
import json, sys
from eliot import add_destinations

def stdout(message):
    print(message)
add_destinations(stdout)
```

Before destinations are added

Up to a 1000 messages will be buffered in memory until the first set of destinations are added, at which point those messages will be delivered to newly added set of destinations. This ensures that no messages will be lost if logging happens during configuration but before a destination is added.

Outputting JSON to a file

Since JSON is a common output format, Eliot provides a utility class that logs to a file, `eliot.FileDestination(file=yourfile)`. Each Eliot message will be encoded in JSON and written on a new line. As a short hand you can call `eliot.to_file`, which will create the destination and then add it automatically. For example:

```
from eliot import to_file
to_file(open("eliot.log", "ab"))
```

Note: This destination is blocking: if writing to a file takes a long time your code will not be able to proceed until writing is done. If you're using Twisted you can wrap a `eliot.FileDestination` with a non-blocking `eliot.logwriter.ThreadedWriter`. This allows you to log to a file without blocking the Twisted reactor.

Customizing JSON Encoding

If you're using Eliot's JSON output you may wish to customize encoding. By default Eliot uses `eliot.json.EliotJSONEncoder` (a subclass of `json.JSONEncoder`) to encode objects. You can customize encoding by passing a custom subclass to either `eliot.FileDestination` or `eliot.to_file`:

```
from eliot.json import EliotJSONEncoder
from eliot import to_file

class MyClass:
    def __init__(self, x):
        self.x = x

class MyEncoder(EliotJSONEncoder):
    def default(self, obj):
        if isinstance(obj, MyClass):
            return {"x": obj.x}
        return EliotJSONEncoder.default(self, obj)

to_file(open("eliot.log", "ab"), encoder=MyEncoder)
```

For more details on JSON encoding see the [Python JSON documentation](#).

Adding Fields to All Messages

Sometimes you want to add a field to all messages output by your process, regardless of destination. For example if you're aggregating logs from multiple processes into a central location you might want to include a field `process_id` that records the name and process id of your process in every log message. Use the `eliot.add_global_fields` API to do so, e.g.:

```
import os, sys
from eliot import add_global_fields

add_global_fields(process_id="%s:%d" % (sys.argv[0], os.getpid()))
```

You should call `add_global_fields` before `add_destinations` to ensure all messages get the global fields.

3.5.2 Journald

`journald` is the native logging system on Linux operating systems that use `systemd` with support for structured, indexed log storage. Eliot provides native `journald` support, with the following features:

- The default message field (`MESSAGE`) stores the Eliot message as JSON.

- Failed actions get priority 3 (“err”) and tracebacks get priority 2 (“crit”).
- The `ELIOT_TASK` field stores the task UUID.
- The `ELIOT_TYPE` field stores the message or action type if available.
- The `SYSLOG_IDENTIFIER` stores `sys.argv[0]`.

Installation

Journald requires additional libraries that are not installed by default by Eliot. You can install them by running:

```
$ pip install eliot[journald]
```

Generating logs

The following example demonstrates how to enable `journald` output.

```
"""
Write some logs to journald.
"""

from __future__ import print_function

from eliot import log_message, start_action, add_destinations
from eliot.journald import JournaldDestination

add_destinations(JournaldDestination())

def divide(a, b):
    with start_action(action_type="divide", a=a, b=b):
        return a / b

print(divide(10, 2))
log_message(message_type="inbetween")
print(divide(10, 0))
```

Querying logs

The `journalctl` utility can be used to extract logs from `journald`. Useful options include `--all` which keeps long fields from being truncated and `--output cat` which only outputs the body of the `MESSAGE` field, i.e. the JSON-serialized Eliot message.

Let’s generate some logs:

```
$ python journald.py
```

We can find all messages with a specific type:

```
$ sudo journalctl --all --output cat ELIOT_TYPE=inbetween | eliot-prettyprint
32ab1286-c356-439d-86f8-085fec3b65d0 -> /1
2015-09-23 21:26:37.972403Z
    message_type: inbetween
```

We can filter to those that indicate errors:

```
$ sudo journalctl --all --output cat --priority=err ELIOT_TYPE=divide | eliot-
↳prettyprint
ce64eb77-bb7f-4e69-83f8-07d7cdaffaca -> /2
2015-09-23 21:26:37.972945Z
  action_type: divide
  action_status: failed
  exception: exceptions.ZeroDivisionError
  reason: integer division or modulo by zero
```

We can also search by task UUID, in which case `eliot-tree` can also be used to process the output:

```
$ sudo journalctl --all --output cat ELIOT_TASK=ce64eb77-bb7f-4e69-83f8-07d7cdaffaca_
↳| eliot-tree
ce64eb77-bb7f-4e69-83f8-07d7cdaffaca
+-- divide@1/started
  |-- a: 10
  |-- b: 0
  `-- timestamp: 2015-09-23 17:26:37.972716
+-- divide@2/failed
  |-- exception: exceptions.ZeroDivisionError
  |-- reason: integer division or modulo by zero
  `-- timestamp: 2015-09-23 17:26:37.972945
```

3.5.3 Using Logstash and Elasticsearch to Process Eliot Logs

Note: Logstash, Elasticsearch and Kibana change frequently. These instructions might not be quite accurate.

[ElasticSearch](#) is a search and analytics engine which can be used to store Eliot logging output. The logs can then be browsed by humans using the [Kibana](#) web UI, or on the command-line using the `logstash-cli` tool. Automated systems can access the logs using the ElasticSearch query API. [Logstash](#) is a log processing tool that can be used to load Eliot log files into ElasticSearch. The combination of ElasticSearch, Logstash, and Kibana is sometimes referred to as ELK.

Example Logstash Configuration

Assuming each Eliot message is written out as a JSON message on its own line (which is the case for `eliot.to_file()` and `eliot.logwriter.ThreadedFileWriter`), the following Logstash configuration will load these log messages into an in-process ElasticSearch database:

logstash_standalone.conf

```
input {
  stdin {
    codec => json_lines {
      charset => "UTF-8"
    }
  }
}

filter {
  date {
    # Parse Eliot timestamp filed into the special @timestamp field Logstash
    # expects:
    match => [ "timestamp", "UNIX" ]
  }
}
```

(continues on next page)

(continued from previous page)

```

    target => ["@timestamp"]
  }
}

output {
  # Stdout output for debugging:
  stdout {
    codec => rubydebug
  }

  elasticsearch {
    # We make the document id unique (for a specific index/mapping type pair) by
    # using the relevant Eliot fields. This means replaying messages will not
    # result in duplicates, as long as the replayed messages end up in the same
    # index.
    document_id => "%{task_uuid}_%{task_level}"
  }
}

```

We can then pipe JSON messages from Eliot into ElasticSearch using Logstash:

```
$ python examples/stdout.py | logstash web -- agent --config logstash_standalone.conf
```

You can then use the Kibana UI to search and browse the logs by visiting <http://localhost:9292/>.

3.6 Reading Logs

3.6.1 Reading and Filtering Eliot Logs

Eliot includes a command-line tool that makes it easier to read JSON-formatted Eliot messages:

```
$ python examples/stdout.py | eliot-prettyprint
af79ef5c-280c-4b9f-9652-e14deb85d52d@/1
2015-09-25T19:41:37.850208Z
  another: 1
  value: hello

0572701c-e791-48e8-9dd2-1fb3bf06826f@/1
2015-09-25T19:41:38.050767Z
  another: 2
  value: goodbye

```

Run `eliot-prettyprint --help` to see the various formatting options; you can for example use a more compact one-message-per-line format.

Additionally, the **highly recommended** third-party `eliot-tree` tool renders JSON-formatted Eliot messages into a tree visualizing the tasks' actions.

Filtering logs

Eliot logs are structured, and by default stored in one JSON object per line. That means you can filter them in multiple ways:

1. Line-oriented tools like `grep`. You can `grep` for a particular task's UUIDs, or for a particular message type (e.g. `tracebacks`).
2. JSON-based filtering tools. `jq` allows you to filter a stream of JSON messages.
3. `eliot-tree` has some filtering and searching support built-in.

For example, here's how you'd extract a particular field with `jq`:

```
$ python examples/stdout.py | jq '.value'
"hello"
"goodbye"
```

Parsing Logs

Eliot also includes a parser for parsing logs into Python objects:

```
import json
from eliot.parse import Parser

def load_messages(logfile_path):
    for line in open(logfile_path):
        yield json.loads(line)

def parse(logfile_path):
    for task in Parser.parse_stream(load_messages(logfile_path)):
        print("Root action type is", task.root().action_type)
```

3.6.2 Message Fields in Depth

Structure

Eliot messages are typically serialized to JSON objects. Fields therefore must have `str` as their name. Message values must be supported by JSON: `int`, `float`, `None`, `str`, `dict` or `list`. The latter two can only be composed of other supported types.

Built-in Fields

A number of fields are reserved by Eliot's built-in message structure and should not be added to messages you create.

All messages contain `task_uuid` and `task_level` fields. Each message is uniquely identified by the combined values in these fields. For more information see the [actions and tasks](#) documentation.

In addition, the following field will also be present:

- `timestamp`: Number of seconds since Unix epoch as a `float` (the output of `time.time()`). Since system time may move backwards and resolution may not be high enough this cannot be relied on for message ordering.

Every logged message will have either `message_type` or `action_type` fields depending whether they originated as a standalone message or as the start or end of an action. Present in regular messages:

- `message_type`: The type of the message, e.g. `"yourapp:yoursubsystem:yourmessage"`.

Present in action messages:

- `action_type`: The type of the action, e.g. `"yourapp:yoursubsystem:youraction"`.

- `action_status`: One of "started", "succeeded" or "failed".

The following fields can be added to your messages, but should preserve the same meaning:

- `exception`: The fully qualified Python name (i.e. import path) of an exception type, e.g. "yourpackage.yourmodule.YourException".
- `reason`: A prose string explaining why something happened. Avoid usage if possible, better to use structured data.
- `traceback`: A string with a traceback.

User-Created Fields

It is recommended, but not necessary (and perhaps impossible across organizations) that fields with the same name have the same semantic content.

3.7 Scientific Computing with Eliot

When it takes hours or days to run your computation, it can take a long time before you notice something has gone wrong, so your feedback cycle for fixes can be very slow. If you want to solve problems quickly—whether it's inconsistent results, crashes, or slowness—you need to understand what was going on in your process as it was running: you need logging.

Eliot is an ideal logging library for these cases:

- It provides structured logging, instead of prose, so you can see inputs, outputs, and intermediate results of your calculations.
- It gives you a trace of what happened, including causality: instead of just knowing that `f()` was called, you can distinguish between calls to `f()` from different code paths.
- It supports scientific libraries: NumPy and Dask. By default, Eliot will automatically serialize NumPy integers, floats, arrays, and booleans to JSON (see *Customizing JSON Encoding* for details).

At PyCon 2019 Itamar Turner-Trauring gave talk about logging for scientific computing, in part using Eliot—you can [watch the video](#) or [read a prose version](#).

3.7.1 Logging large arrays

Logging large arrays is a problem: it will take a lot of CPU, and it's no fun discovering that your batch process was slow because you mistakenly logged an array with 30 million integers every time you called a core function.

So how do you deal with logging large arrays?

1. **Log a summary (default behavior):** By default, if you log an array with size > 10,000, Eliot will only log the first 10,000 values, along with the shape.
2. **Omit the array:** You can also just choose not to log the array at all. With `log_call` you can use the `include_args` parameter to ensure the array isn't logged (see *Logging Functions*). With `start_action` you can just not pass it in.
3. **Manual transformation:** If you're using `start_action` you can also manually modify the array yourself before passing it in. For example, you could write it to some sort of temporary storage, and then log the path to that file. Or you could summarize it some other way than the default.

3.7.2 Using Dask

If you're using the `Dask` distributed computing framework, you can automatically use Eliot to trace computations across multiple processes or even machines. This is mostly useful for Dask's Bag and Delayed support, but can also be used with arrays and dataframes.

In order to do this you will need to:

- Ensure all worker processes write the Eliot logs to disk (if you're using the multiprocessing or distributed backends).
- If you're using multiple worker machines, aggregate all log files into a single place, so you can more easily analyze them with e.g. `eliot-tree`.
- Replace `dask.compute()` with `eliot.dask.compute_with_trace()`.
- Replace `dask.persist()` with `eliot.dask.persist_with_trace()`.

In the following example, you can see how this works for a Dask run using `distributed`, the recommended Dask scheduler for more sophisticated use cases. We'll be using multiple worker processes, but only use a single machine:

```
from os import getpid

from dask.bag import from_sequence
import dask.config
from dask.distributed import Client
from eliot import log_call, to_file
from eliot.dask import compute_with_trace

@log_call
def multiply(x, y=7):
    return x * y

@log_call
def add(x, y):
    return x + y

@log_call
def main_computation():
    bag = from_sequence([1, 2, 3])
    bag = bag.map(multiply).fold(add)
    return compute_with_trace(bag)[0] # instead of dask.compute(bag)

def _start_logging():
    # Name log file based on PID, so different processes so stomp on each
    # others' logfiles:
    to_file(open("{}_log".format(getpid()), "a"))

def main():
    # Setup logging on the main process:
    _start_logging()

    # Start three worker processes on the local machine:
    client = Client(n_workers=3, threads_per_worker=1)

    # Setup Eliot logging on each worker process:
    client.run(_start_logging)
```

(continues on next page)

(continued from previous page)

```
# Run the Dask computation in the worker processes:
result = main_computation()
print("Result:", result)

if __name__ == '__main__':
    import dask_eliot
    dask_eliot.main()
```

In the output you can see how the various Dask tasks depend on each other, and the full trace of the computation:

```
$ python examples/dask_eliot.py
Result: 42
$ ls *.log
7254.log 7269.log 7271.log 7273.log
$ eliot-tree *.log
ca126b8a-c611-447e-aaa7-f61701e2a371
├─ main_computation/1 started 2019-01-01 17:27:13 0.047s
│   └─ dask:compute/2/1 started 2019-01-01 17:27:13 0.029s
│       └─ eliot:remote_task/2/8/1 started 2019-01-01 17:27:13 0.001s
│           └─ dask:task/2/8/2 2019-01-01 17:27:13
│               dependencies:
│               └─ 0: map-multiply-75feec3a197bf253863e330f3483d3ac-0
│                   key: reduce-part-71950de8264334e8cea3cc79d1c2e639-0
│               └─ multiply/2/8/3/1 started 2019-01-01 17:27:13 0.000s
│                   x: 1
│                   y: 7
│                   └─ multiply/2/8/3/2 succeeded 2019-01-01 17:27:13
│                       result: 7
│           └─ eliot:remote_task/2/8/4 succeeded 2019-01-01 17:27:13
│       └─ eliot:remote_task/2/9/1 started 2019-01-01 17:27:13 0.001s
│           └─ dask:task/2/9/2 2019-01-01 17:27:13
│               dependencies:
│               └─ 0: map-multiply-75feec3a197bf253863e330f3483d3ac-1
│                   key: reduce-part-71950de8264334e8cea3cc79d1c2e639-1
│               └─ multiply/2/9/3/1 started 2019-01-01 17:27:13 0.000s
│                   x: 2
│                   y: 7
│                   └─ multiply/2/9/3/2 succeeded 2019-01-01 17:27:13
│                       result: 14
│           └─ eliot:remote_task/2/9/4 succeeded 2019-01-01 17:27:13
│       └─ eliot:remote_task/2/10/1 started 2019-01-01 17:27:13 0.001s
│           └─ dask:task/2/10/2 2019-01-01 17:27:13
│               dependencies:
│               └─ 0: map-multiply-75feec3a197bf253863e330f3483d3ac-2
│                   key: reduce-part-71950de8264334e8cea3cc79d1c2e639-2
│               └─ multiply/2/10/3/1 started 2019-01-01 17:27:13 0.000s
│                   x: 3
│                   y: 7
│                   └─ multiply/2/10/3/2 succeeded 2019-01-01 17:27:13
│                       result: 21
│           └─ eliot:remote_task/2/10/4 succeeded 2019-01-01 17:27:13
│       └─ eliot:remote_task/2/11/1 started 2019-01-01 17:27:13 0.001s
│           └─ dask:task/2/11/2 2019-01-01 17:27:13
│               dependencies:
│               └─ 0: reduce-part-71950de8264334e8cea3cc79d1c2e639-0
│                   └─ 1: reduce-part-71950de8264334e8cea3cc79d1c2e639-1
```

(continues on next page)

(continued from previous page)

```

└─ 2: reduce-part-71950de8264334e8cea3cc79d1c2e639-2
   key: reduce-aggregate-71950de8264334e8cea3cc79d1c2e639
└─ add/2/11/3/1 started 2019-01-01 17:27:13 0.000s
   └─ x: 7
   └─ y: 14
   └─ add/2/11/3/2 succeeded 2019-01-01 17:27:13
      └─ result: 21
└─ add/2/11/4/1 started 2019-01-01 17:27:13 0.000s
   └─ x: 21
   └─ y: 21
   └─ add/2/11/4/2 succeeded 2019-01-01 17:27:13
      └─ result: 42
└─ eliot:remote_task/2/11/5 succeeded 2019-01-01 17:27:13
└─ dask:compute/2/12 succeeded 2019-01-01 17:27:13
└─ main_computation/3 succeeded 2019-01-01 17:27:13
   └─ result: 42

```

Warning: Retries within Dask will result in confusing log messages; this will eventually be fixed in a future release.

3.8 Python 2.7 Support

The last version of Eliot to support Python 2.7 was release 1.7.

If you are using Eliot with Python 2, keep the following in mind:

- I will provide critical bug fixes for Python 2 until March 2020. I will accept patches for critical bug fixes after that (or you can [pay for my services](#) to do additional work).
- Make sure you use an up-to-date `setuptools` and `pip`; in theory this should result in only downloading versions of the package that support Python 2.
- For extra safety, you can pin Eliot in `setup.py` or `requirements.txt` by setting: `eliot < 1.8`.
- Critical bug fixes for Python 2 will be released as 1.7.1, 1.7.2, etc..

3.9 Contributing to Eliot

To run the full test suite, the `Daemontools` package should be installed.

All modules should have the `from __future__ import unicode_literals` statement, to ensure Unicode is used by default.

Coding standard is PEP8, with the only exception being camel case methods for the Twisted-related modules. Some camel case methods remain for backwards compatibility reasons with the old coding standard.

You should use `black` to format the code.