Lithoxyl Documentation

Release 0.1.0

Mahmoud Hashemi

February 28, 2017

Contents

Lithoxyl is a next-generation instrumentation toolkit for Python applications, offering a semantic, action-oriented approach to logging and metrics collection. Lithoxyl integration is compact and performant, minimizing impact on codebase readability and system performance.

CHAPTER 1

Sections

Lithoxyl Overview

Lithoxyl is next-generation logging and instrumentation for Python. This practical tutorial walks new users through the fundamentals necessary to get up and running with Lithoxyl in under 10 minutes.

Motivating factors

Lithoxyl began as a response to the tired traditions of logging. Traditions that included omission, procrastination, and only adding it once things break.

Logging is not the last step anymore. Lithoxyl makes instrumentation worthwhile from day 1, so all your projects are designed for introspection. Lithoxyl achieves this by taking full advantage of Python's rich syntax and runtime, providing features ranging from metrics collection to structured logging to interactive debugging hooks.

The Lithoxyl approach is practical. After running `pip install lithoxyl`, integrating Lithoxyl comes down to two steps: instrumentation and configuration. First, instrumentation.

Instrumenting with Actions

With Lithoxyl, all instrumentation, including logging, starts with knowing your application. We want to find the important parts of your application and wrap them in microtransactions, called Actions.

Much more than print statements, Actions are lightweight objects that track the state of code execution, from timing information to uncaught exceptions. Each Action also has a name and a level, to enable aggregation and filtering.

Actions are created with Loggers. We get into creating and configuring Loggers later in the overview, but here's a basic example of creating an *info*-level Action with a preconfigured Logger:

```
import backend            # some convenient backend logic for brevity
from log import app_log   # preconfigured Lithoxyl Logger

def create_entry(name):
    with app_log.info('adding entry by name'):
        name = name.strip()
        backend.add_by_name(name)
    return True
```

As you can see, the transactionality of Actions translates well to Python's *with* context manager syntax. A single line of logging code succinctly records the beginning and ending of this code block. Even better, there's no chance of missing an unexpected exception. For instance, if **name** is not a string, and **.strip()** raises an `AttributeError`, then that exception is guaranteed to be captured and recorded.

You can do so much more with actions. Using dictionary syntax, arbitrary data can be added to the action. And while actions finish with a success status and autogenerate a message if no exception is raised, failures and exceptions can also be set manually:

```
import backend
from log import app_log

def set_entry_state(name, state):

    with app_log.info('setting entry state') as act:
        act['name'] = name
        status = backend.STATE_MAP[state.lower()]
        success = backend.set_entry_state(name, state)
        if not success:
            act.failure('set {name} status to {state} failed', state=state)

     return success
```

As seen above, actions can also have a custom completion message, which supports templating with new-style formatting syntax, using data from within the action's data map (*name*), as well as arguments and keyword arguments (*state*).

Note: Even if message formatting fails, the log message will still be recorded. Only the failing segments will be left unformatted. As a rule, Lithoxyl degrades gracefully, to minimize impact to your application's primary functionality.

Furthermore, in cases like this, where you want the whole function logged, you can use the logger's `wrap()` method.:

```
import backend
from log import app_log

@app_log.wrap('critical', inject_as='act')
def delete_entry(name, act):
    try:
        ret = backend.delete_entry_by_name(name.strip())
    except backend.EntryNotFound:
        # log soft error, let other exceptions raise through
        act.failure('no entry with name: {}', name)
        ret = False
    return ret
```

Note the decorator syntax, as well as the ability to inject the action as one of the arguments of the function. This reduces the instrumentation's code footprint even further.

That about covers creating and interacting with actions. Now we turn to the origin and destination of the actions we create and populate: Loggers and Sinks.

Creating Loggers

Actions make up most of an application's interaction with Lithoxyl, but it would not be very easy to create an Action without a Logger.

As we learned above, before an Action can be populated, it must be created, and Actions are created through Logger. As for the Logger itself, here is how it is created:

```
from lithoxyl import Logger

app_log = Logger('entry_system')
```

Like that, the Logger we've been using above is ready to be imported. A Logger is a lightweight, simple object, requiring only a name. They are designed to be created once, configured, and imported by other modules. That said, they are conceptually very useful.

Loggers generally correspond to parts or aspects of the application. Small- to medium-sized applications can be fully instrumented with just one Logger, but as applications grow, they tend to add aspects. For example, if file access grows increasingly important to an application, it would make sense to add a dedicated low-level log just for instrumenting file access:

```
file_log = Logger('file_access')
```

In short, Loggers themselves are simple, and designed to be fit to your application, no matter how many aspects it may have. On their own, they are conceptually useful, but without Sinks, they are all potential.

Configuring Sinks

So far, we have discovered two uses of the Lithoxyl Logger:

- Creating actions
- Segmenting and naming aspects of an application

Now, we are ready to add the third: publishing log events to the appropriate handlers, called Sinks. Actions can carry all manner of messages and measurements. That variety is only surpassed by the Sinks, which handle aggregation and persistence, through log files, network streams, and much more. Before getting into those complexities, let's configure our `app_log` with a simple but very useful sink:

```
from lithoxyl import AggregateSink

agg_sink = AggregateSink(limit=100)
app_log.add_sink(agg_sink)
```

Now, by adding an instance of the AggregateSink to the `app_log`, we have a technically complete system. At any given point after this, the last 100 events that passed through our application log will be available inside `agg_sink`. However, AggregateSinks only provide in-memory storage, meaning data must be pulled out, either through a monitoring thread or network service. Most developers expect persistent logging to streams (stdout/stderr) and files. Lithoxyl is more than capable.

Logging Sensibly

For developers who want a sensible and practical default Sink, Lithoxyl provides the SensibleSink. The Sensible Suite chapter has a full introduction, so let's just cover the basics.

The Sensible approach has 3 steps:

1. **Filter** - Optionally ignore events for a given Sink.
2. **Format** - Convert an event into a string.
3. **Emit** - Output the formatted string to a file, database, network, etc.

While totally pluggable and overridable, the Sensible suite ships with types for each of these:

```
from lithoxyl import (SensibleFilter,
                      SensibleFormatter,
                      StreamEmitter,
                      SensibleSink)

# Create a filter that controls output verbosity
fltr = SensibleFilter(success='critical',
                      failure='info',
                      exception='debug')

# Create a simple formatter with just two bits of info:
# The time since startup/import and end event message.
# These are just two of the built-in "fields",
# and the syntax is new-style string formatting syntax.
fmtr = SensibleFormatter('+{import_delta_s} - {end_message}')

# Create an emitter to write to stderr. 'stdout' and open file objects
# also behave predictably.
emtr = StreamEmitter('stderr')

# Tie them all together. Note that filters accepts an iterable
sink = SensibleSink(filters=[fltr], formatter=fmtr, emitter=emtr)

# Add the sink to app_log, a vanilla Logger created above
app_log.add_sink(sink)
```

In these six lines of code, using only built-in Lithoxyl types, we create a filter, formatter, and emitter, then we bind them all together with a SensibleSink. The output is first filtered by our SensibleFilter, which only shows critical-level successes and info-level failures, but shows all exceptions. Our SensibleFormatter provides a simple but practical output, giving us a play-by-play timing and message. That message is output to stderr by our StreamEmitter. Just don't forget to add our newly-created SensibleSink to the app_log.

As configured, the app_log will now write to stderr output that looks like:

```
+0.015255 - "load credential succeeded"
+0.179199 - "client authorization succeeded"
+0.344523 - "load configuration succeeded"
+0.547119 - "optional backup failed"
+1.258266 - "processing task succeeded"
```

Ain't it a thing of beauty? Here we see the SensibleFormatter at work. It may not look like much, but there is a powerful feature at work.

The ambitious aim underlying the Sensible approach is to create human-readable structured logs. These are logs that are guaranteed to be uniformly formatted and escaped, allowing them to be loaded for further processing steps, such as collation with other logs, ETL into database/OLAP, and calculation of system-wide statistics. Extending the flow of logged information opens up many new roads in debugging, optimization, and system robustification, easily justifying a bit of extra up-front setup.

Here we only used two fields, *import_time_s* and *end_message*. The list of Sensible built-in fields is quite expansive and worth a look when designing your own log formats.

The Action

Actions are Lithoxyl's primary interface for instrumenting your application. Actions are created with a `Logger` instance, and are used to wrap functions and code blocks.

At their most basic, Actions have a:

- **name** - A string description of the behavior being wrapped.
- **level** - An indicator of the importance of the action (debug, info, critical).
- **status** - The state of the action (begin, success, failure, exception).
- **duration** - The time between the begin and end events of a completed action, i.e., the time between entering and exiting a code block.

To track this information, Lithoxyl wraps important pieces of your application in microtransactions called Actions:

```
with log.info('user creation', username=name) as act:
    succeeded = _create_user(name)
    if not succeeded:
        act.failure()
```

This pattern is using an *info*-level Action as a *context manager*. The indented part of the code after the *with* statement is the code block managed by the Action. Here is how the basics of the Action are populated in our example:

- **name** - "user creation"
- **level** - INFO
- **status** - *failure* if `_create_user(name)` returns a falsey value, *exception* if it raises an exception, otherwise defaults to *success*.
- **duration - Set automatically, duration is the time difference** from before the execution of the first line of the code block to after the execution of the last line in the code block, or the `r.failure()` call, depending on the outcome of `_create_user(name)`.

There's quite a bit going on, but Lithoxyl has several tricks that let it flow with the semantics of applications. First, let's learn a bit about these attributes, starting with the Action level.

Action level

Levels are a basic indicator of how important a block of application logic is. Lithoxyl has three built-in levels. In order of increasing importance:

- **debug** - Of interest to developers. Supplementary info for when something goes wrong.
- **info** - Informational. Can be helpful to know even when there are no problems.
- **critical** - Core functionality. Essential details at all times.

When instrumenting with Lithoxyl, the developer is always asking, how significant is the success of this code block, how catastrophic is a failure in this function?

It's only natural that instrumented code will start with more *critical* actions. The most important parts should be instrumented first. Eventually the instrumentation spreads to lower levels.

Note: As a general tendency, as code gets closer to the operating system, the corresponding Action also gets a lower level. High-level operations get higher levels of Actions. Start high and move lower as necessary.

Action status

The Lithoxyl Action has an eventful lifetime. Even the most basic usage sees the Action going from creation to beginning to one of the ending states: success, failure, or exception.

First, simply creating an Action does not "begin" it. An action begins when it is entered with a **with** statement, as we saw in the example above. Entering an action creates a timestamp and makes it the parent of future actions, until it is ended.

There are three end statuses:

- **success** - The action described by the action completed without issue. This is the automatic default when no exception is raised.
- **failure** - The action did not complete successfully, and the failure was expected and/or handled within the application.
- **exception** - The action terminated unexpectedly, likely with a Python exception. This is the automatic default when an exception is raised within an action context manager.

The split between *failure* and *exception* should be familiar to users of standard testing frameworks like py.test. Test frameworks distinguish between a test that fails and a test that could not be fully run because the test code raised an unexpected exception. Lithoxyl brings these semantics into an application's runtime instrumentation.

Note: If an action is manually set to complete with `success()` or `failure()`, and an unexpected exception occurs, the Action will end with the *exception* status.

Action API

Actions are usually constructed through Loggers, but it can help to know the underlying API and see the obvious parallels.

class `lithoxyl.action.`**`Action`**(*logger*, *level*, *name*, *data=None*, *reraise=True*, *parent=None*, *frame=None*)

The Action type is one of the core Lithoxyl types, and the key to instrumenting application logic. Actions are usually instantiated through convenience methods on `Logger` instances, associated with their level (e.g., `critical()`).

Parameters

- **`logger`** – The Logger instance responsible for creating and publishing the Action.
- **`level`** – Log level of the Action. Generally one of `DEBUG`, `INFO`, or `CRITICAL`. Defaults to `None`.
- **`name`** (*`str`*) – A string description of some application action.
- **`data`** (*`dict`*) – A mapping of non-builtin fields to user values. Defaults to an empty dict (`{}`) and can be populated after Action creation by accessing the Action like a `dict`.
- **`reraise`** (*`bool`*) – Whether or not the Action should catch and reraise exceptions. Defaults to `True`. Setting to `False` will cause all exceptions to be caught and logged appropriately, but not reraised. This should be used to eliminate `try`/`except` verbosity.
- **`frame`** – Frame of the callpoint creating the Action. Defaults to the caller's frame.

Most of these parameters are managed by the Actions and respective `Logger` themselves. While they are provided here for advanced use cases, usually only the *name* and *raw_message* are provided.

Actions are `dict`-like, and can be accessed as mappings

and used to store additional structured data:

```
>>> action['my_data'] = 20.0
>>> action['my_lore'] = -action['my_data'] / 10.0
>>> from pprint import pprint
>>> pprint(action.data_map)
{'my_data': 20.0, 'my_lore': -2.0}
```

exception(*message=None*, **a*, ***kw*)
Mark this Action as having had an exception. Also sets the Action's *message* template similar to `Action.success()` and `Action.failure()`.

Unlike those two attributes, this method is rarely called explicitly by application code, because the context manager aspect of the Action catches and sets the appropriate exception fields. When called explicitly, this method should only be called in an `except` block.

failure(*message=None*, **a*, ***kw*)
Mark this Action failed. Also set the Action's *message* template. Positional and keyword arguments will be used to generate the formatted message. Keyword arguments will also be added to the Action's `data_map` attribute.

get_elapsed_time()
Simply get the amount of time that has passed since begin was called on this action, or 0.0 if it has not begun. This method has no side effects.

success(*message=None*, **a*, ***kw*)
Mark this Action successful. Also set the Action's *message* template. Positional and keyword arguments will be used to generate the formatted message. Keyword arguments will also be added to the Action's `data_map` attribute.

Action concurrency

TODO

The Logger

The `Logger` is the application developer's primary interface to using Lithoxyl. It is used to conveniently create `Actions` and publish them to `sinks`.

class `lithoxyl.logger.`**Logger**(*name*, *sinks=None*, ***kwargs*)
The `Logger` is one of three core Lithoxyl types, and the main entrypoint to creating `Action` instances, and publishing those *actions* to *sinks*.

Parameters

- **name** (*str*) – Name of this Logger.
- **sinks** (*list*) – A list of *sink* objects to be attached to the Logger. Defaults to `[]`. Sinks can be added later with `Logger.add_sink()`.
- **module** (*str*) – Name of the module where the new Logger instance will be stored. Defaults to the module of the caller.

Most Logger methods and attributes fal into three categories: `Action` creation, Sink registration, and Event handling.

Action creation

The Logger is primarily used through its `Action`-creating convenience methods named after various log levels: `debug()`, `info()`, and `critical()`.

Each creates a new *action* with a given name, passing any additional keyword arguments on through to the `lithoxyl.action.Action` constructor.

`Logger.`**`debug`**`(`*action_name*, ***kw*`)`
Returns a new `DEBUG`-level `Action` named *name*.

`Logger.`**`info`**`(`*action_name*, ***kw*`)`
Returns a new `INFO`-level `Action` named *name*.

`Logger.`**`critical`**`(`*action_name*, ***kw*`)`
Returns a new `CRITICAL`-level `Action` named *name*.

The action level can also be passed in:

`Logger.`**`action`**`(`*level*, *action_name*, ***kw*`)`
Return a new `Action` named *name* classified as *level*.

Sink registration

Another vital aspect of Loggers is the *registration and management of Sinks*.

`Logger.`**`sinks`**
A copy of all sinks set on this Logger. Set sinks with `Logger.set_sinks()`.

`Logger.`**`add_sink`**`(`*sink*`)`
Add *sink* to this Logger's sinks. Does nothing if *sink* is already in this Logger's sinks.

`Logger.`**`set_sinks`**`(`*sinks*`)`
Replace this Logger's sinks with *sinks*.

`Logger.`**`clear_sinks`**`()`
Clear this Logger's sinks.

Event handling

The event handling portion of the Logger API exists for Logger-Sink interactions.

`Logger.`**`on_begin`**`(`*begin_event*`)`
Publish *begin_event* to all sinks with `on_begin()` hooks.

`Logger.`**`on_end`**`(`*end_event*`)`
Publish *end_event* to all sinks with `on_end()` hooks.

`Logger.`**`on_warn`**`(`*warn_event*`)`
Publish *warn_event* to all sinks with `on_warn()` hooks.

`Logger.`**`on_exception`**`(`*exc_event*, *exc_type*, *exc_obj*, *exc_tb*`)`
Publish *exc_event* to all sinks with `on_exception()` hooks.

The Sink

In Lithoxyl's system of instrumentation, Actions are used to carry messages, data, and timing metadata through the Loggers to their destination, the Sinks. This chapter focuses in on this last step.

Writing a simple Sink

Sinks can grow to be very involved, but a useful Sink can be as simple as:

```
import sys

class DotSink(object):
    def on_end(self, end_event):
        sys.stdout.write('.')
        sys.stdout.flush()
```

Note that our new Sink does not have to inherit from any special object. DotSink is a correct and capable Sink, ready to be instantiated and installed with `Logger.add_sink()`, just like in *the overview*. Once added to your Logger, every time an Action ends, a dot will be written out to your console.

In this example, `on_end` is the handler for just one of Lithoxyl's events. The next section takes a look at all five of them.

Events

Lithoxyl Events are state changes associated with a particular Action. Five types of events can happen in the Lithoxyl system:

- **begin** - The beginning of an Action, whether manually or through entering a context-managed block of code.

  The begin event corresponds to the method signature `on_begin(self, begin_event)`. Designed to be called once per Action.

- **end** - The completion of an Action, whether manually (`success()` and `failure()`) or through exiting a context-managed block of code. There are three ways an Action can end, **success**, **failure**, and **exception**, but all of them result in an *end* event.

  The end event corresponds to the method signature `on_end(self, end_event)`. Designed to be called once per Action.

- **exception** - Called immediately when an exception is raised from within the context-managed block, or when an exception is manually handled with Action.exception(). Actions ending in exception state typically fire two events, one for handling the exception, and one for ending the Action.

  The exception event corresponds to the Sink method signature `on_exception(self, exc_event, exc_type, exc_obj, exc_tb)`. Designed to be called up to once.

- **warn** - The registration of a warning within an Action.

  Corresponds to the Sink method signature `on_warn(self, warn_event)`. Can be called an arbitrary number of times.

- **comment** - The registration of a comment from a Logger. Comments are used for publishing metadata associated with a Logger.

  The comment event corresponds to the Sink method signature `on_comment(self, comment_event)`. See here for more about comments. Can be called an arbitrary number of times.

A Sink handles the event by implementing the respective method. The event objects that accompany every event are meant to be practically immutable; their values are set once, at creation.

The Sensible Suite

Structured logging creates logs with a consistent format, allowing them to be loaded later for further processing and analysis.

One of Lithoxyl's primary uses is as a toolkit for creating these structured logs. The Sensible Suite is the first generalized approach to offer structured logging without sacrificing human readability.

Let's look at an example. Perhaps the most common structured log is the HTTP server access log, such as the one created by Apache or nginx. A couple entries from that log might look like:

```
78.178.243.200 - - [22/Jun/2013:15:02:31 -0700] "GET /favicon.ico HTTP/1.1" 404 570 "-" "Mozilla/5.0
119.63.193.132 - - [22/Jun/2013:14:19:36 -0700] "GET / HTTP/1.1" 200 9755 "-" "Mozilla/4.0 (compatib
```

It's a bit on the wide side, but here we see:

- The IP of the client
- The local date and time the request was received
- The request line, including the method, path, and version
- The response code returned to the client
- The size of the response in bytes
- The user agent from the client browser

With the Sensible suite, each of these values becomes a *field*, represented by SensibleField objects. The Sensible suite comes with over twenty built-in fields to cover most use cases, and sensible default handling for other values. These fields are used to create a template for the SensibleFormatter, which knows how to turn a Lithoxyl Action into a structured string. Let's see how it all comes together by creating an equivalent log that uses Lithoxyl built-in behavior:

```python
from lithoxyl import SensibleFormatter, FileEmitter, Logger

a_log = Logger('access_log')

a_fmtr = SensibleFormatter('{ip} - [{iso_begin_local}] {req_line} {resp_code} {resp_len} {user_agent

a_sink = SensibleSink(formatter=fmtr, emitter=FileEmitter('access.log'))

a_log.add_sink(a_sink)
```

No arcane configuration format here. Everything is configured through explicit Python code. The `a_log` logger has only one sink right now, a SensibleSink that ties together three entities, in their running order:

- **Filters** - This list of objects checks each event, and returns True/False depending on whether it should be logged. See the *`SensibleFilter`* for more info.
- **Formatter** - Turns events that make it through the filters into strings. The *`SensibleFormatter`* is the canonical formatter of the suite, though you're free to provide your own.
- **Emitters** - Writes formatted strings into files or network streams. Emitters are not strictly a Sensible construct; several can be found in the `emitters` module.

The flow through the SensibleSink is clear: Filtration → Formatting → Output. Any actions passing through the `a_log` Logger will have their *end* events logged to *access.log*.

The Sensible Interfaces

To achieve human-readable strutured logging, Lithoxyl's Sensible suite uses four key types with a sensible naming scheme:

- The *`SensibleSink`*
- The *`SensibleFilter`*
- The *`SensibleFormatter`*
- The *`SensibleField`*

The first three are used fairly regularly, but SensibleField is mostly behind the scenes. That said, the built-in fields can in many ways the most important part. See the Sensible Fields section below for details on those.

class `sensible.`**`SensibleSink`**(*formatter=None*, *emitter=None*, *filters=None*, *on=('begin'*, *'warn'*, *'end'*, *'exception'*, *'comment')*)

class `sensible.`**`SensibleFilter`**(*base=None*, ***kw*)

class `sensible.`**`SensibleFormatter`**(*base=None*, ***kwargs*)

Sensible Fields

There are many built-in Sensible Fields, for a variety of use cases. First, some example code to set the context for the field examples:

```
logger = Logger('test_logger')
with logger.critical('test_task', reraise=False) as act:
    time.sleep(0.7)
    act['item'] = 'cur_item'
    act.failure('task status: {status_str}')
    raise ValueError('unexpected value for {item}')
return act
```

And now the fields themselves:

Name	Description	Example
logger_name	The name of the Logger, as set in the constructor. Quoted.	`"test_logger"`
logger_id	An automatic integer ID. See *Action concurrency*.	`3`
action_name	Short string description of the action. Quoted.	`"test_task"`
action_id	An automatic integer ID. See *Action concurrency*.	`17`
action_guid	A globally unique ID string. See *Action concurrency*.	`c3124107db02ff33dbde8e85`
status_str	The full name of action status. See *Action status*.	`exception`
status_char	A single-character action status. See *Action status*.	`E`
level_name	Full name of the action level.	`critical`
level_name_upper	Full name of the action level, in uppercase. See *Action level*.	`CRITICAL`
level_char	Single-character form of the action level.	`C`
level_number	The integer value associated with the action level.	`90`
data_map	JSON-serialized form of all values in the Action data map.	`{"item":  "cur_item"}`
data_map_repr	repr()-serialized form of all values in the Action data map.	`{"item":  "cur_item"}`
begin_message	The message associated with the event's action's begin event.	`"test_task beginning"`
begin_message_raw	Same as **begin_message**, before formatting.	`"test_task beginning"`
end_message	The message associated with the event's action's end event.	`"test_task raised ...  ue for`
end_message_raw	Same as **end_message**, before formatting.	`"test_task raised ...  lue for`
event_message	The message associated with the event.	`"test_task raised ...  ue for`

Continu

Table 1.1 – continued from previous page

Name	Description	Example
event_message_raw	Same as **event_message**, before formatting.	`"test_task raised ... lue for`
duration_s	Duration in floating point number of seconds.	`0.701`
duration_ms	Duration in floating point number of milliseconds (ms).	`700.908`
duration_us	Duration in floating point number of microseconds (us).	`700907.946`
duration_auto	Duration in floating point with automatic unit (s/ms/us).	`700.908ms`
module_name	The name of the module where the action was created.	`"__main__"`
module_path	The path of the module where the action was created.	`"misc/gen_field_table.py"`
func_name	The name of the function that created the action	`get_test_action`
line_number	The line number where the action was created.	`26`
exc_type	The name of the exception type, if an exception was caught.	`ValueError`
exc_message	The exception message, if there was one. Quoted.	`"unexpected value for {item}"`
exc_tb_str	The exception's full traceback, if there was one. Quoted.	`"Traceback (most r ... ue for`
exc_tb_list	A JSON representation of the exception traceback. Quoted.	`"[Callpoint('get_t ... for {i`
process_id	The integer process ID. See `os.getpid()`.	`19828`

There can be some subtle nuances when designing your log structure. For instance, when choosing which message to use for an event, you almost certainly want **event_message**, which works equally well with all event types, including begin, end, comment, and warn.

Timestamp fields

Timestamps are so important to logging, especially structured logging, that they get a table of their own:

Name	Description	Example
iso_begin	The full ISO8601 begin event UTC timestamp, with timezone.	`2016-05-22T10:41:06.470354+0000`
iso_end	The full ISO8601 end event UTC timestamp, with timezone.	`2016-05-22T10:41:07.171262+0000`
iso_begin_notz	The begin event ISO UTC timestamp, without timezone.	`2016-05-22T10:41:06.470354`
iso_end_notz	The end event ISO UTC timestamp, without timezone.	`2016-05-22T10:41:07.171262`
iso_begin_local	The begin event ISO local timestamp, with timezone.	`2016-05-22T03:41:06.470354-0700`
iso_end_local	The end event ISO local timestamp, with timezone.	`2016-05-22T03:41:07.171262-0700`
iso_begin_local_notz	The begin event ISO local timestamp, without timezone.	`2016-05-22T03:41:06.470354`
iso_end_local_notz	The end event ISO local timestamp, without timezone.	`2016-05-22T03:41:07.171262`
iso_begin_local_noms	The begin event ISO local timestamp, without subsecond timing.	`2016-05-22T03:41:06 PDT`
iso_end_local_noms	The end event ISO local timestamp, without subsecond timing.	`2016-05-22T03:41:07 PDT`
iso_begin_local_noms_notz	The begin event local times, without subsecond or timezone.	`2016-05-22T03:41:06`
iso_end_local_noms_notz	The end event local times, without subsecond or timezone.	`2016-05-22T03:41:07`

The timestamp fields above are geared toward long-running processes like servers. For shorter running processes, it's often more readable and more useful to know the time between the log message and process start.

Name	Description	Example
import_delta_s	Floating-point number of seconds since lithoxyl import.	`2.887265`
import_delta_ms	Floating-point number of milliseconds since lithoxyl import.	`2887.265`

Creating custom fields

Most custom data does not require new fields. Unrecognized fields are treated as quoted and escaped string data. If you want to change that representation, you can create a SensibleField and either register it locally with a Formatter, or globally, using sensible.register_field().

class `sensible.`**`SensibleField`**(*fname*, *fspec='s'*, *getter=None*, ***kwargs*)

Fields specify whether or not they should be *quoted* (i.e., whether or not values will contain whitespace or other delimiters), but not the exact method for their quoting. That aspect is reserved for the Formatter.

The Logging Tradition

For experienced engineers, it can help to understand Lithoxyl by taking a hard look at the past and current state of logging.

Logging in General

Without getting into Python specifics, most ecosystems have pretty low standards for logging. Logging is an afterthought, added when the application misbehaves and needs to be debugged. Just having *any* logging can easily put an application in the top quartile for quality.

And worse yet, the opposite can be true. Logging's place in software is so low that having logging is often a yellow flag for lower-quality code in need of constant debugging. If the code needed so much logging, it must have had a lot of problems.

This is the past and present reality of logging in general.

Logging in Python

This will be frank, so first things first: all due respect to Vinay Sajip and all the Python contributors who worked on Python logging. Without their work, there is no telling where we would be today. Now, the critique.

The built-in `logging` module itself followed this afterthought pattern. Little more than a knockoff of Log4j, `logging` pays virtually no mind to performance, practicality, or the fact that *Python is not Java*.

Application instrumentation is important. Good metrics are worth more than their weight in CPU cycles. By running a high-level language like Python, a design decision has already been made to achieve a richer, more featureful environment.

With that in mind, it is critical that Python libraries take the semantic high road. Always emphasize maintainability, introspectability, and reliability in Python code.

Because application instrumentation is vital to all these areas, the approach and framework used *must* be closely matched. The built-in `logging` library is a frumpy, secondhand suit, thrifted and worn without even a thorough cleaning. Lithoxyl is new, tailored to fit Python and its many, many modern applications.

The Lithoxyl Response

Python's power lets us do better. And we can't stop with just logging. We need to look at instrumentation as a whole.

Tradition is to add logging to indicate breakage. Little more than print statements and tracebacks piped to files.

Modern instrumentation is more than a debugging utility.

Lithoxyl provides structured data and online statistics to unlock your application's potential. Lithoxyl is a development tool, worth using from day one. Good instrumentation focuses on the whole application lifecycle. It helps with debugging problems, but it also offers direction when the sun is shining and the monitoring is green. Lithoxyl is the Pythonic step toward that bright, introspectable future.

Frequently Asked Questions

Lithoxyl's new approach answers quite a lot of questions, but raises a few others. These questions fall into two categories, *Design* and *Background*.

Design questions

Some questions are hard because they are ultimately decided by your application's design. Lithoxyl is mostly an API to instrumentation. There are many right ways.

What is the difference between failure status and exception status?

There are a couple angles to answer this. First, it is pretty rare to set an exception status manually, as exception information is usually populated automatically when there are uncaught exceptions. That contrasts with `failure()`, which is seen more often.

So when to call `failure()`? As with many design questions, an example is often best. With an HTTP server, returning a 4xx or even a 503 can be viewed as failures outside of the control of the application, which is performing fine. A 500, on the other hand, is generally unexpected and deserves an exception status.

Why does Lithoxyl sometimes fail silently?

Built-in to the design of Lithoxyl itself, there are several deviations from what one might consider standard practice. With most libraries, one expects that code will "fail fast". However, failing fast does not work well for instrumentation code.

Lithoxyl assumes that you are instrumenting a system which has behavior other than logging and statistics collection. Your system's primary functions take priority. Instrumentation must degrade gracefully.

This means if your message is malformed Lithoxyl will do its best to output the most that it can and no exception will be raised. If your logging service is down, maybe the Sink queues the message, but eventually that queues bounds will be overrun and messages may silently drop.

This graceful degradation takes place at all the runtime integration points, i.e., action usage within your application code. For Sink and Logger configuration, actions which are typically performed at startup and import time, exceptions are still raised as usual. In fact, it is considered good Lithoxyl practice to forward-check these configurations. This means checking that callable arguments are

If you discover a runtime scenario that should degrade with more grace or a configuration-time scenario which could prevent runtime failures through more forward checking, please do file an issue.

Background questions

Unlike the design questions above, background questions relate to just the objective facts.

What’s with the name, Lithoxyl, what’s that even mean?

Lithoxyl is a geological term for petrified wood. Fossilized trees. Rock-solid logs.

Glossary

action An instance of the `Action` type, and one of the three fundamental Lithoxyl types. The `Action` type is rarely instantiated directly, instead they are created by *loggers*, manipulated, and automatically published to *sinks*.

emitter An object capable of publishing formatted messages out of the process. Emitters commonly publish to network services, local services, and files. The last step in the Sensible Filter-Format-Emit logging process.

event An occurence associated with a Logger and Action. One of:

- **begin** - The start of an Action.
- **end** - The completion of an Action (success, failure, or exception)
- **warn** - A warning related to an Action.
- **comment** - A metadata event associated with a Logger
- **exception** - An unhandled exception during an Action.

Sinks implement methods to handle each of these events.

formatter An object responsible for transforming a *action* into a string, ready to be encoded and *emitted*

lithoxyl Mineralized wood.

logger An instance of the `Logger` type. Responsible for facilitating the creation and publication of *actions*. Generally there is one logger per aspect of an application. For example, a request logger and a database query logger.

sink Any object implementing the Sink protocol for handling *events*. Typically subscribed to *actions* by being attached to a *logger*. Some basic types of sinks include action emitters, statistics collectors, and profilers.

status The completion state of an *action*, meant to represent one of four possible task outcomes:

- **Begin** - not yet completed
- **Success** - no exceptions or failures
- **Failure** - anticipated or application-level unsuccessful completion (e.g., invalid username)
- **Exception** - unanticipated or lower-level unsuccessful completion (e.g., database connection interrupted)

with Python’s compact context manager syntax, roughly approximating a “try-finally” block. With blocks have *enter* and *exit* hooks that enable tracking of Action events, no matter whether the wrapped code executes successfully or raises an exception.

Python Module Index