Jotting Documentation

Release 0.2.1

Ryan Morshead

Mar 04, 2018

Contents

jotting is a log system for Python 2 and 3 that can be used to record the causal history of an asynchronous or distributed system. These histories are composed of actions which, once "started", will begin "working", potentially spawn other actions, and eventually end as a "success" or "failure". In the end you're left with a breadcrumb trail of information that you can use to squash bugs with minimal boilerplate.

CHAPTER 1

Install

Install `jotting` with pip:

```
pip install jotting
```

Then requests and flask to follow along with the examples:

```
pip install requests flask
```

1.1 Development

If you'd like to work the source code, then clone the repository from github:

```
git clone git@github.com:rmorshea/jotting.git && cd jotting
```

And do an editable install with pip that includes *requirements.txt*:

```
pip install -e . -r requirements.txt
```

CHAPTER 2

Quickstart

We'll begin with a function that uses *requests* to return a response from a url:

```
import requests


def get_url(url):
    r = requests.get(url)
    r.raise_for_status()
    return r


response = get_url("https://google.com")
```

To log when *get_url* is called, we can add *book.mark* as a decorator:

```
import requests
from jotting import book


@book.mark
def get_url(url):
    r = requests.get(url)
    r.raise_for_status()
    return r


response = get_url("https://google.com")
```

Once we've done this *get_url* will immediately begin to get print logs:

```
|-- started: __main__.get_url
|   @ 2018-01-14 17:08:19.223383
|   | url: https://google.com
|   `-- success: __main__.get_url
|       @ 2018-01-14 17:08:20.101563
```

```
|       | returned: <Response [200]>
|       | duration: 0.879 seconds
```

If we want more than what *book.mark* gives us we can also use *book.write*:

```
import requests
from jotting import book


@book.mark
def get_url(url):
    r = requests.get(url)
    book.write(debug="checking status...")
    r.raise_for_status()
    return r


response = get_url("https://google.com")
```

And now we get an extra log telling us what's going on inside *get_url*:

```
|-- started: __main__.get_url
|   @ 2018-01-14 17:08:19.223383
|   | url: https://google.com
|   |-- working: __main__.get_url
|   |   @ 2018-01-14 17:08:20.101401
|   |   | debug: checking status...
|   `-- success: __main__.get_url
|       @ 2018-01-14 17:08:20.101563
|       | returned: <Response [200]>
|       | duration: 0.879 seconds
```

2.1 Putting Things In Context

But wait! We have scripts or functions that have subtasks we'd like to monitor:

```
import requests

urls = ("https://google.com", "not-here")

responses = []
for u in urls:
    r = requests.get(u)
    r.raise_for_status()
    responses.append(r)
```

We can use *book* as a context manager to log anywhere we'd like:

```
import requests
from jotting import book

urls = ("https://google.com", "not-here")

responses = []
for u in urls:
```

```
    with book("getting %s" % u):
        r = requests.get(u)
        r.raise_for_status()
        responses.append(r)
```

This will produce just the kind of fine grained logs we need:

```
|-- started: getting https://google.com
|   @ 2018-01-14 17:06:22.016731
|   `-- success: getting https://google.com
|       @ 2018-01-14 17:06:23.006855
|       | duration: 0.990 seconds
|-- started: getting not-here
|   @ 2018-01-14 17:06:23.007092
|   `-- failure: getting not-here
|       @ 2018-01-14 17:06:23.007587
|       | MissingSchema: Invalid URL 'not-here': No schema supplied. Perhaps you␣
↪meant http://not-here?
|       | duration: 0.001 seconds
```

CHAPTER 3

Documentation

3.1 Outlets

We can configure where `jotting` sends logs by choosing new "outlets". Outlets can be any callable object, but we can also use builtin outlets from `jotting.to` in order to save logs to a file:

```
import requests
from jotting import book, to

book.distribute(to.File(path="~/Desktop/logbox.txt"))


# we can format the title with
# the inputs of the function
@book.mark("getting {url}")
def get_url(url):
    r = requests.get(url)
    r.raise_for_status()
    return r


response = get_url("https://google.com")
```

Now we will find a `logbox.txt` file on our desktop with the following contents:

```
{"metadata": {"title": "getting https://google.com", "timestamps": [1519973286.
↪701371], "tag": "d6154a2a16db4561b151fc43b3781f75", "parent": null, "status":
↪"started"}, "content": {"url": "https://google.com"}}
{"metadata": {"title": "getting https://google.com", "timestamps": [1519973286.701371,
↪ 1519973286.991931], "tag": "d6154a2a16db4561b151fc43b3781f75", "parent": null,
↪"status": "success", "stop": 1519973286.991928}, "content": {"returned": "<Response␣
↪[200]>"}}
```

In all the examples we've seen so far, `jotting` has produced clean nested tree of log statements. However, these saved logs show us that under the hood `jotting` isn't magic - each log is a dictionary that contains the information

required to reconstruct a history of actions.

3.1.1 Your Own Outlets

You can make your own outlets with the *jotting.to.outlet()* decorator. The decorator takes in functions and returns a new *jotting.to.Outlet* class. It expects functions of the form `(log, *args, **kwargs)` where `log` is a formatted log string generated by a user, and `*args, **kwargs` were the parameters that construct the outlet instance. Given this, we can easily recreate the `jotting.to.File` outlet:

```
import os
from jotting.to import outlet


@outlet
def File(log, path):
    path = os.path.realpath(os.path.expanduser(path))
    with open(path, "a+") as f:
        f.write(log)
```

or the `jotting.to.Print` outlet:

```
import sys
from jotting.to import outlet


@outlet
def Print(log):
    """Send logs directly to ``sys.stdout``"""
    sys.stdout.write(log)
    sys.stdout.flush()
```

3.2 Styling

In the last section we saw how to save logs to files, and we also noticed, that under the hood, each log message was actually a dictionary of data. Yet we know that `jotting` can produce human readable readouts. This is possible by "styling" log statements - reformatting the raw data into a more presentable form. Each *jotting.to.Outlet* accepts as its first argument, and style - a callable object which, given raw log data, returns a formatted string. The default style for outlets is *jotting.style.Raw* which simply encodes the data as a json blob, but we could also use *jotting.style.Tree* to produce nested ascii readouts instead:

```
import requests
from jotting import book, to, style

tree = style.Tree()
path = "~/Desktop/logbox.txt"
tree_to_file = to.File(tree, path=path)
book.distribute(tree_to_file)


@book.mark("getting {url}")
def get_url(url):
    r = requests.get(url)
    r.raise_for_status()
    return r
```

```
response = get_url("https://google.com")
```

Now instead of raw log data, we'll find an ascii tree in `logbox.txt`:

```
|-- started: getting https://google.com
|   @ 2018-03-03 16:53:45.380436
|   | url: https://google.com
|   `-- success: getting https://google.com
|       @ 2018-03-03 16:53:45.692461
|       | returned: <Response [200]>
|       | duration: 0.312 seconds
```

3.3 Async

In the real world we aren't working with single threads, processes, or services. Modern systems are asynchronous and distributed. Following the causes and effects within them quickly becomes impossible. However with *jotting*, it's possible to begin a *book* using the *tag* of a parent task that triggered it. In this way logs can be linked across any context. We can build a very simple *Flask* app to demonstrate how we might link a book between a client and server:

```
from flask import Flask, jsonify, request
from jotting import book
import json

# Server
# ------

app = Flask(__name__)


@app.route("/api/task", methods=["PUT"])
def task():
    data = json.loads(request.data)
    # link the tag of a parent book
    with book('api', data["parent"]):
        book.conclude(status=200)
        return jsonify({"status": 200})


# Client
# ------

with book('put') as b:
    route = '/api/task'
    # hand off the tag of the current book
    data = json.dumps({'parent': b.tag})
    app.test_client().put(route, data=data)
```

```
|-- started: get
|   @ 2018-02-25 18:22:45.912632
|   |-- started: api
|   |   @ 2018-02-25 18:22:45.922958
|   |   `-- success: api
|   |       @ 2018-02-25 18:22:45.923105
```

```
|   |       | status: 200
|   |       | duration: 0.000 seconds
|   `-- success: get
|       @ 2018-02-25 18:22:45.928721
|       | duration: 0.016 seconds
```

CHAPTER 4

Examples

Listing 4.1: The `book.mark` as a decorator

```
import requests
from jotting import book


@book.mark
def get_url(url):
    r = requests.get(url)
    book.write(debug="checking status...")
    r.raise_for_status()
    return r


response = get_url("https://google.com")
```

Listing 4.2: Using `book` as a context manager

```
import requests
from jotting import book

urls = ("https://google.com", "not-here")

responses = []
for u in urls:
    with book("getting %s" % u):
        r = requests.get(u)
        r.raise_for_status()
        responses.append(r)
```

Listing 4.3: Changing where you jot down your logs

```
import requests
from jotting import book, to, style
```

```
to_print = to.Print(style.Log())
to_file = to.File(path="~/Desktop/logbox.txt")
book.distribute(to_print, to_file)


# we can format the title with
# the inputs of the function
@book.mark("getting {url}")
def get_url(url):
    r = requests.get(url)
    r.raise_for_status()
    return r


response = get_url("https://google.com")
```

Listing 4.4: A toy example showing how to link logs across the web with flask

```
from flask import Flask, jsonify, request
from jotting import book
import json

# Server
# ------

app = Flask(__name__)


@app.route("/api/task", methods=["PUT"])
def task():
    data = json.loads(request.data)
    with book('api', data["parent"]):
        book.conclude(status=200)
        return jsonify({"status": 200})


# Client
# ------

with book('put') as b:
    route = '/api/task'
    data = json.dumps({'parent': b.tag})
    app.test_client().put(route, data=data)
```

Listing 4.5: Link logs between asynchronous threads

```
import sys
import time
if sys.version_info < (3, 0):
    from Queue import Queue
else:
    from queue import Queue
import threading, requests
from jotting import book, to, read

logbox = "~/Desktop/logbox.txt"
```

```
book.distribute(to.File(path=logbox))


def get(queue, url, parent):
    """Get the URL and queue the response."""
    with book("get({url})", parent, url=url):
        try:
            response = requests.get(url)
        except Exception as e:
            queue.put(e)
        else:
            queue.put(response.status_code)


@book.mark
def schedule(function, *args):
    """Create a thread for each mapping of the function to args."""
    q = Queue()
    for x in args:
        # we want to resume the current book
        inputs = (q, x, book.current("tag"))
        threading.Thread(target=function, args=inputs).start()
        book.write(scheduled=x)
    return [q.get(timeout=5) for i in range(len(args))]


urls = ["https://google.com", "https://wikipedia.org"]
responses = schedule(get, *urls)

time.sleep(0.1) # give time for logs to flush

read.Complete(logbox)
```

Listing 4.6: Link logs between asynchronous processes

```
import sys
import time
import threading, requests
from jotting import book, to, read
from multiprocessing import Process, Queue

logbox = "~/Desktop/logbox.txt"
book.distribute(to.File(path=logbox))


def get(queue, url, parent):
    """Get the URL and queue the response."""
    book.distribute(to.File(path=logbox))
    with book("get({url})", parent, url=url):
        try:
            response = requests.get(url)
        except Exception as e:
            queue.put(e)
        else:
            queue.put(response.status_code)


@book.mark
```

```
def schedule(function, *args):
    """Create a process for each mapping of the function to args."""
    q = Queue()
    for x in args:
        # we want to resume the current book
        inputs = (q, x, book.current("tag"))
        Process(target=function, args=inputs).start()
        book.write(scheduled=x)
    return [q.get(timeout=5) for i in range(len(args))]


urls = ["https://google.com", "https://wikipedia.org"]
responses = schedule(get, *urls)

time.sleep(0.1) # give time for logs to flush

read.Complete(logbox)
```

CHAPTER 5

API

class jotting.book.**book**(*title*, *parent=None*, ***content*)

Create a new book for logging.

Parameters

- **title** (*string, function, or class*) – A string representing the title of the book. For functions and classes, a title is infered from its title, and module. Inference attempt to drill down into closures to determin the root function, or class. This typically happens when a function or class has many decorators.
- **parent** (*string or None*) – The tag of the last book. If `None` then the last book within the current thread is used. To link across threads or processes, you must manually communicate this.
- ****content** (*any*) – A dictionary of content that will be logged when the book is opened.

classmethod conclude(**args*, ***kwargs*)

Write the content that will be logged when the book closes.

classmethod current(*data=None*)

Get the current book, or metadata from the current book.

Parameters **data** (*string or None*) – A string indicating a desired piece of metadata from the current book. If `None`, then the current book is returned instead.

Returns The current book, or an entry in its metadata.

Return type *book*

classmethod distribute(**outlets*)

Set which *jotting.to.Outlet* objects recieve logs.

metadata

Get a copy of this book's metadata.

classmethod outlets()

Get the outlets for all books.

status
Get this book's tag.

Returns 'started', 'working', 'success', or 'failure'.

Return type string

tag
Get this book's tag.

classmethod write(**args*, ***kwargs*)
Write a log to the currently open book.

class jotting.to.**Outlet**(*style=<jotting.style.Raw object>*, **args*, ***kwargs*)
A base *Outlet* class.

Subclasses should override the _handler() method.

Parameters

- **style** (*callable*) – A function that returns a formated log string. This is usually a jotting.to.Style object that returns a string.
- ***args** (*any*) – Positional arguments passed to Outlet._handler() - a method meant to be overriden in subclasses. You can use *outlet()* to turn functions into an Outlets whose handler is that function.
- ****kwargs** (*any*) – Keyword arguments passed to Outlet._handler() - a method meant to be overriden in subclasses. You can use *outlet()* to turn functions into an Outlets whose handler is that function.

jotting.to.**outlet**(*handler*)
Turn a function into an *Outlet*.

This decorator should wrap functions that send logs wherever they need to go.

Parameters **handler** (*callable*) – A function of the form (log, *args, **kwargs), where log is a log generated by a user, and *args, **kwargs are the parameters that initialize the outlet (e.g. File(path=/path/to/file)).

class jotting.read.**Complete**(*source*)
Read a complete set of logs from a file or list of dictionaries.

This class will organize a given set of logs such that they are contextually, but not necessarilly chronologically ordered. You can get a string representation of the given logs styled as a *jotting.style.Tree* simply by converting it to a string (e.g. str(Complete(my_source))), or you can iterate over the reordered logs (e.g. list(Complete(my_source))) and style them yourself later.

Parameters **source** (*string or iterable containing log strings*) – If given as a string source will be interpreted as a filepath. Otherwise source should it should be a list of log strings.

class jotting.read.**Stream**(**outlets*)
Read a stream of log strings or dictionaries.

Parameters ***outlets** (*callable*) – A series of callable Outlet objects that will receive logs one at a time. Logs will be collected and then distributed in batches, in order to make guesses about causes and effects. This only works for logs that were created synchronously.

Notes

The stream attempts to make educated guesses about causes and effects. In other words, it will attempt to reorder the logs. This works well for logs that were synchronously created.

For logs created in parrallel threads or processes, you should store your logs in a file, and read them back with *Complete*.

class jotting.style.**Log**
A basic formater that only creates successes, and failures.

class jotting.style.**Raw**
Creates a string representation of the log object.

class jotting.style.**Style**
The base *Style* type.

class jotting.style.**Tree**
An ascii tree representation for logs.

Python Module Index

j

Index