Tubing Documentation

Release 0.2

Bob Corsaro

March 12, 2016

Table of Contents

Tubing is a Python I/O library. What makes tubing so freakin' cool is the gross abuse of the bit-wise OR operator (|). Have you ever been writing python code and thought to yourself, "Man, this is great, but I really wish it was a little more like bash." Whelp, we've made python a little more like bash.If you are a super lame nerd-kid, you can replace any of the bit-wise ORs with the tube() function and pray we don't overload any other operators in future versions. Here's how you install tubing:

```
$ pip install tubing
```

Tubing is pretty bare-bones at the moment. We've tried to make it easy to add your own functionality. Hopefully you find it not all that unpleasant. There are three sections below for adding sources, tubes and sink. If you do make some additions, think about committing them back upstream. We'd love to have a full suite of tools.

Now, witness the power of this fully operational I/O library.

```
from tubing import sources, tubes, sinks

objs = [
    dict(
        name="Bob Corsaro",
        birthdate="08/03/1977",
        alignment="evil",
    ),
    dict(
        name="Tom Brady",
        birthdate="08/03/1977",
        alignment="good",
    ),
]
sources.Objects(objs) \
     | tubes.JSONDumps() \
     | tubes.Joined(by=b"\n") \
     | tubes.Gzip() \
     | sinks.File("output.gz", "wb")
```

Then in our old friend bash.

```
$ zcat output.gz
{"alignment": "evil", "birthdate": "08/03/1977", "name": "Bob Corsaro"}
{"alignment": "good", "birthdate": "08/03/1977", "name": "Tom Brady"}
$
```

You can find more documentation on readthedocs

CHAPTER 1

Catalog

1.1 Sources

Objects	Takes a *list* of python objects.
File	Creates a stream from a file.
Bytes	Takes a byte string.
IO	Takes an object with a read function.
Socket	Takes an addr, port and socket() args. .
HTTP	Takes an method, url and any args that can be passed to requests library.

1.2 Tubes

Gunzip	Unzips a binary stream.
Gzip	Zips a binary stream.
JSONLoads	Parses a byte string stream of raw JSON objects. Will try to use ujson, then built-in json.
JSONDumps	Serializes an object stream using *json.dumps*. Will try to use ujson, then built-in json.
Split	Splits a stream that supports the *split* method.
Joined	Joins a stream of the same type as the *by* argument.
Tee	Takes a sink and passes chunks along apparatus.
Map	Takes a transformer function for single items in stream.
Filter	Takes a filter test callback and only forwards items that pass.
ChunkMap	Takes a transformer function for batch of stream items.

1.3 Sinks

Objects	A list that stores all passed items to self.
Bytes	Saves each chunk self.results.
File	Writes each chunk to a file.
HTTPPost	Writes data via HTTPPost.
Hash	Takes algorithm name, updates hash with contents.
Debugger	Writes each chunk to the tubing.tubes debugger with level DEBUG.

1.4 Extensions

s3.S3Source	Create stream from an S3 object.
s3.MultipartUploader	Stream data to S3 object.
elasticsearch.BulkSink	Stream *elasticsearch.DocUpdate* objects to the elasticsearch _bulk endpoint.

CHAPTER 2

Sources

To make your own source, create a Reader class with the following interface.

```
class MyReader(object):
    """
    MyReader returns count instances of data.
    """
    def __init__(self, data="hello world\n", count=10):
        self.data = data
        self.count = count

    def read(self, amt):
        """
        read(amt) returns $amt of data and a boolean indicating EOF.
        """
        if not amt:
            amt = self.count
        r = self.data * min(amt, self.count)
        self.count -= amt
        return r, self.count <= 0
```

The important thing to remember is that your read function should return an iterable of units of data, not a single piece of data. Then wrap your reader in the loving embrace of MakeSourceFactory.

```
from tubing import sources

MySource = sources.MakeSourceFactory(MyReader)
```

Now it can be used in a apparatus!

```
from __future__ import print_function

from tubing import tubes
sink = MySource(data="goodbye cruel world!", count=1) \
     | tubes.Joined(by=b"\n") \
     | sinks.Bytes()

print(sinks.result)
# Output: goodbye cruel world!
```

CHAPTER 3

Tubes

Making your own tube is a lot more fun, trust me. First make a Transformer.

```
class OptimusPrime(object):
    def transform(self, chunk):
        return list(reversed(chunk))
```

chunk is an iterable with a len() of whatever type of data the stream is working with. In Transformers, you don't need to worry about buffer size or closing or exception, just transform an iterable to another iterable. There are lots of examples in tubes.py.

Next give Optimus Prime a hug.

```
from tubing import tubes

AllMixedUp = tubes.MakeTranformerTubeFactory(OptimusPrime)
```

Ready to mix up some data?

```
from __future__ import print_function

import json
from tubing import sources, sinks

objs = [{"number": i} for i in range(0, 10)]

sink = sources.Objects(objs) \
     | AllMixedUp(chunk_size=2) \
     | sinks.Objects()

print(json.dumps(sink))
# Output: [{"number": 1}, {"number": 0}, {"number": 3}, {"number": 2}, {"number": 5}, {"number": 4},
```

CHAPTER 4

Sinks

Really getting tired of making documentation... Maybe I'll finish later. I have real work to do.

Well.. I'm this far, let's just push through.

```
from __future__ import print_function
from tubing import sources, tubes, sinks

class StdoutWriter(object):
    def write(self, chunk):
        for part in chunk:
            print(part)

    def close(self):
        # this function is optional
        print("That's all folks!")

    def abort(self):
        # this is also optional
        print("Something terrible has occurred.")

Debugger = sinks.MakeSinkFactory(StdoutWriter)

objs = [{"number": i} for i in range(0, 10)]

sink = sources.Objects(objs) \
     | AllMixedUp(chunk_size=2) \
     | tubes.JSONDumps() \
     | tubes.Joined(by=b"\n") \
     | Debugger()
# Output:
#{"number": 1}
#{"number": 0}
#{"number": 3}
#{"number": 2}
#{"number": 5}
#{"number": 4}
#{"number": 7}
#{"number": 6}
#{"number": 9}
#{"number": 8}
#That's all folks!
```

4.1 API Docs

There are two types of tubing users. Irish users, and users who wish they were Irish. j/k. Really, they are:

- casual users want to use whatever we have in tubing.
- advanced users who want to extend tubing to their own devices.
- contributor users want to contribute to tubing.

Our most important users are the casual users. They are also the easiest to satisfy. For them, we have tubes. Tubes are easy to use and understand.

Advanced users are very important too, but harder to satisfy. We never know what crazy plans they'll have in mind, so we must be ready. They need the tools to build new tubes that extend our apparatus in unexpected ways.

For the benefit of the contributors, and ourselves, we're going to outline exactly how things work now. This documentation is also an exercise in understanding and simplifying the code base.

We'll call a tubing pipeline an apparatus. An apparatus has a Source, zero to many Tubes, and a Sink.

A stream is what we call the units flowing through our Tubes. The units can be bytes, characters, strings or objects.

4.1.1 Tubes

For most users, they simply want to transform elements of data as it goes through the stream. This can be achieved simply with the following idiom:

```
SomeSource | [Tubes ..] | tubes.Map(lambda x: transform(x)) | [Tubes ..] | SomeSink
```

Sometimes you'd like to transform an entire chunk of data at a time, instead of one element at a time:

```
SomeSource | [Tubes ..] | tubes.ChunkMap(lambda x: transform(x)) | [Tubes ..] | SomeSink
```

Other times you just want to filter out some data:

```
SomeSource | [Tubes ..] | tubes.Filter(lambda x: x > 10) | [Tubes ..] | SomeSink
```

All of these general tube tools also take close_fn and abort_fn params and are shorthand for creating your own Tube class.

Of course, if you need to keep state, you can create a closure, but at some point, that can become cumbersome. You might also want to make a reusable Tube, in that case it could be nice to make a

The easiest way to extend tubing is to create a Transformer, and use TransformerTubeFactory decorator to turn it into a Tube. A Transformer has the following interface:

```
@tubes.TransformerTubeFactory()
class NewTube(object):
    def transform(self, chunk):
        return new_chunk

    def close(self):
        return last_chunk or None

    def abort(self):
        pass
```

A chunk is an iterable of whatever type of stream we are working on, whether it be bytes, Unicode characters, strings or python objects. We can index it, slice it, or iterate over it. *transform* simple takes a chunk, and makes a new chunk

out of it. *TransformerTubeFactory* will take care of all the dirty work. Transformers are enough for most tasks, but if you need to do something more complex, you may need to go deeper.

First let's describe how tubes work in more detail. Here's the Tube interface:

```
# tube factory can be a class

class TubeFactory(object):
    # This is what we export, and what is called when users create a tube.
    # The syntax looks like this:
    # SourceFactory() | [TubeFactory()...] | SinkFactory()
    def __call__(self, *args, **kwargs):
        return Tube()

#  or a function

def TubeFactory(*args, **kwargs):
    return Tube()

# ------------------------
class Tube(object):
    def receive(self, source):
        # return a TubeWorker
        tw = TubeWorker
        tw.source = source
        return tw

class TubeWorker(object):
    def tube(self, receiver):
        # receiver is they guy who will call our `read` method. Either
        # another Tube or a Sink.
        return receiver.receive(self)

    def __or__(self, *args, **kwargs):
        # Our reason for existing.
        return self.tube(*args, **kwargs)

    def read(self):
        # our receiver will call this guy. We return a tuple here of
        # `chunk, eof`.  We should return a chunk of len amt of whatever
        # type of object we produce. If we've exhausted our upstream
        # source, then we should return True as the second element of our
        # tuple. The chunk size should be configuratable and read should
        # return a len() of chunk size or less.
        return [], True
```

A TubeFactory is what casual users deal with. As you can see, it can be an object or a function, depending on your style. It's easier for me to reason about state with an object, but if you prefer a closure, go for it! Classes are just closures with more verbose states, after all.

When a casual is setting up some tubing, the TubeFactory returns a Tube, but this isn't the last object we'll create. The Tube doesn't have a source connected, so it's sort of useless. It's just a placeholder waiting for a source. As soon as it gets a source, it will hand off all of it's duties to a TubeWorker.

A TubeWorker is ready to read from it's source, but it doesn't. TubeWorkers are pretty lazy and need someone else to tell them what to do. That's where a receiver comes in to play. A receiver can be another Tube, or a Sink. If it's another Tube, you know the drill. It's just another lazy guy that will only tell his source to read when his boss tells him to read. Ultimately, the only guy who wants to do any work is the Sink. At the end of the chain, a sink's receive function will be called, and he'll get everyone to work.

Technically, we could split the TubeWorker interface into two parts, but it's not really necessary since they share the same state. We could also combine TubeFactory, Tube and TubeWorker, and just build up state overtime. I've seriously consider this, but I don't know, this feels better. I admit, it is a little complicated, but one advantage you get is that you can do something like this:

```
from tubing import tubes

tube = tubes.GZip(chunk_size=2**32)

source | tube | output1
source | tube | output2
```

Since tube is a factory and not an object, each tubeline will have it's own state. tubeline.... I just made that up. That's an execution pipeline in tubing. But we don't want to call it a pipeline, it's a tubeline. Maybe there's a better name? I picture a chemistry lab with a bunch of transparent tubes connected to beakers and things like that.

Let's call it an apparatus.

TransformerTubeFactory

So how does TransformerTubeFactory turn a Transformer into a TubeFactory? TransformerTubeFactory is a utility that creates a function that wrap a transformer in a tube. Sort of complicate, eh? I'm sorry about that, but let's see if we can break it down.

TransformerTubeFactory returns a partial function out of the TransformerTube instantiation. For the uninitiated, a partial is just a new version of a function with some of the parameters already filled in. So we're currying the transformer_cls and the default_chunk_size back to the casuals. They can fill in the rest of the details and get back a TransformerTube.

The TransformerTubeWorker is where most of the hard work happens. There's a bunch of code related to reading just enough chunks from our source to satisfy our receiver. Remember, Workers are lazy, that's good because we won't waste a bunch of space doing work we don't need to and then waiting for our work to be consumed.

default_chunk_size is sort of important, by default it's something like 2**18. It's the size of the chunks that we request from upstream, in the read function (amt). That's great for byte streams(maybe?), but it's not that great for large objects. You'll probably want to set it if you are using something other than bytes. It can be overridden by plebes, this is just the default if they don't specify it. Remember, we should be making the plebes job easy, so try and be a nice noble and set it to something sensible. In our own tests, using 2**3 for string or object streams and 2**18 for bytes streams seemed to give the best trade off between speed and memory usage. YMMV.

We've explained Tubes, very well I might add. And it's a good thing. They are the most complicated bit in tubing. All that's left is Sources and Sinks.

4.1.2 Sources

TODO

4.1.3 Sinks

TODO

4.1.4 Things You Can't do with Tubing

- Tee to another apparatus

- async programming
- your laundry

4.1.5 Subpackages

tubing.ext package

Submodules

tubing.ext.elasticsearch module

`tubing.ext.elasticsearch.`**`BulkUpdate`**(*base_url*, *index*, *username=None*, *password=None*, *chunks_per_post=512*, *fail_on_error=True*)

Docs per post is source.chunk_size * chunks_per_post.

class `tubing.ext.elasticsearch.`**`DocUpdate`**(*doc*, *doc_type*, *esid=None*, *parent_esid=None*, *doc_as_upsert=True*)

Bases: `object`

DocUpdate is an ElasticSearch document update object. It is meant to be used with BulkBatcher and returns an action and update.

`action`(*encoding*)

`serialize`(*encoding*)

`update`(*encoding*)

exception `tubing.ext.elasticsearch.`**`ElasticSearchError`**

Bases: `exceptions.Exception`

ElasticSearchError message is the text response from the elasticsearch server.

class `tubing.ext.elasticsearch.`**`Scroller`**(*base_url*, *index*, *typ*, *query*, *timeout='10m'*, *scroll_id=None*, *username=None*, *password=None*)

Bases: `object`

`get_hits`()

tubing.ext.s3 module

Module contents

4.1.6 Submodules

4.1.7 tubing.compat module

compat provides tools to make code compatible across python versions.

`tubing.compat.`**`python_2_unicode_compatible`**(*klass*)

lifted from Django A decorator that defines __unicode__ and __str__ methods under Python 2. Under Python 3 it does nothing.

To support Python 2 and 3 with a single code base, define a __str__ method returning text and apply this decorator to the class.

4.1.8 tubing.sinks module

class tubing.sinks.**HTTPPost**(*url*, *username=None*, *password=None*, *chunks_per_post=1024*, *response_handler=<function <lambda>>*)
Bases: object

HTTPPost doesn't support the write method, and therefore can not be used with tubes.Tee.

gen()
We have to do this goofy shit because requests.post doesn't give access to the socket directly. In order to stream, we need to pass a generator object to requests.

4.1.9 tubing.sources module

class tubing.sources.**MakeSourceFactory**(*reader_cls*, *default_chunk_size=65536*)
Bases: object

MakeSourceFactory takes a reader object and returns a Source factory.

class tubing.sources.**Source**(*reader*, *chunk_size*)
Bases: object

Source is a wrapper for Readers that allows piping.

4.1.10 tubing.tubes module

tubing.tubes.**MakeTransformerTubeFactory**(*transformer_cls*, *default_chunk_size=262144*)
Returns a TransformerTubeFactory, which in turn, returns a TransformerTube.

class tubing.tubes.**TransformerTube**(*transformer_cls*, *default_chunk_size*, **args*, ***kwargs*)
Bases: object

TransformerTube is what is returned by a TransformerTubeFactory. It manages the initialization of the TransformerTubeWorker.

tubing.tubes.**TransformerTubeFactory**(*default_chunk_size=262144*)
TransformerTubeFactory is a decorator to turn a Transformer class into a TransformerTubeFactory.

class tubing.tubes.**TransformerTubeWorker**(*apparatus*, *chunk_size*, *transformer*)
Bases: object

TransformerTubeWorker wraps a Transformer and does all the grunt work that most tubes need to do. Transformers should implement transform(chunk), and optionally close() and abort().

append(*chunk*)
append to the buffer, creating it if it doesn't exist.

buffer_len()
buffer_len even if buffer is None.

read()
This is where the rubber meets the snow.

read_complete()
read_complete tells us if the current request is fulfilled. It's fulfilled if we've reached the EOF in the source, or we have $amt parts. If amt is None, we should read to the source's EOF.

shift_buffer(*amt*)
Remove $amt data from the front of the buffer and return it.

class tubing.tubes.**TubeIterator**(*tube*)
Bases: object

TubeIterator wraps a tube in an iterator object.

t