processor

Release 0.9.0

December 06, 2015

Contents

CHAPTER 1

Simple rules

Python processor is a tool for creating chained pipelines for dataprocessing. It have very few key concepts:

Data object Any python dict with two required fields: `source` and `type`.

Source An iterable sequence of `data objects` or a function which returns `data objects`. See full list of sources in the docs.

Output A function which accepts a `data object` as input and could output another. See full list of outputs in the docs. (or same) `data object` as result.

Predicate Pipeline consists from sources outputs, but `predicate` decides which `data object` should be processed by which `output`.

CHAPTER 2

Quick example

Here is example of pipeline which reads IMAP folder and sends all emails to Slack chat:

```
run_pipeline(
    sources.imap('imap.gmail.com'
                 'username',
                 'password'
                 'INBOX'),
    [prepare_email_for_slack, outputs.slack(SLACK_URL)])
```

Here you construct a pipeline, which uses `sources.imap` for reading imap folder "INBOX" of `username@gmail.com`. In more complex case `outputs.fanout` can be used for routing dataobjects to different processors and `sources.mix` can be used to merge items two or more sources into a one stream.

Functions `prepare_email_to_slack` and `outputs.slack(SLACK_URL)` are processors. First one is a simple function which accepts data object, returned by imap source and transforming it to the data object which could be used by slack.output. We need that because slack requires a different set of fields. Call to `outputs.slack(SLACK_URL)` returns a function which gets an object and send it to the specified Slack's endpoint.

It is just example, for working snippets, continue reading this documention ;-)

Note: By the way, did you know there is a Lisp dialect which runs on Python virtual machine? It's name is HyLang, and python processor is written in this language.

CHAPTER 3

Installation

Create a virtual environment with python3::

```
virtualenv --python=python3 env
source env/bin/activate
```

Install required version of hylang (this step is necessary because Hy syntax is not final yet and frequently changed by language maintainers)::

```
pip install -U 'git+git://github.com/hylang/hy.git@a3bd90390cb37b46ae33ce3a73ee84a0feacce7d#egg=hy'
```

If you are on OSX, then install lxml on OSX separately::

```
STATIC_DEPS=true pip install lxml
```

Then install the `processor`::

```
pip install processor
```

CHAPTER 4

Usage

Now create an executable python script, where you'll place your pipline's configuration. For example, this simple code creates a process line which searches new results in Twitter and outputs them to console. Of cause, you can output them not only to console, but also post by email, to Slack chat or everywhere else if there is an output for it:

```
#!env/bin/python3
import os
from processor import run_pipeline, sources, outputs
from twiggy_goodies.setup import setup_logging


for_any_message = lambda msg: True

def prepare(tweet):
    return {'text': tweet['text'],
            'from': tweet['user']['screen_name']}

setup_logging('twitter.log')

run_pipeline(
    sources=[sources.twitter.search(
        'My Company',
        consumer_key='***', consumer_secret='***',
        access_token='***', access_secret='***',
        )],
    rules=[(for_any_message, [prepare, outputs.debug()])])
```

Running this code, will fetch new results for search by query `My Company` and output them on the screen. Of course, you could use any other `output`, supported by the `processor`. Browse online documentation to find out which sources and outputs are supported and for to configure them.

CHAPTER 5

Ideas for Sources and Outputs

- `web-hook` endpoint *(in progress)*.
- `tail` source which reads file and outputs lines appeared in a file between invocations or is able to emulate `tail -f` behaviour. Python module tailer could be used here.
- `grep` output – a filter to grep some fields using patterns. With `tail` and `grep` you could build a pipeline which watch on a log and send errors by email or to the chat.
- `xmpp` output.
- `irc` output.
- `rss/atom feed reader`.
- `weather` source which tracks tomorrow's weather forecast and outputs a message if it was changed significantly, for example from "sunny" to "rainy".
- `github` some integrations with github API?
- `jira` or other task tracker of your choice?
- *suggest your ideas!*

CHAPTER 6

What is next?

Read about sources, outputs and try to build you own pipeline!

And please, send you contributions as pull requests. Writing new sources and outputs is easier than you think!

6.1 Sources

6.1.1 mix

This is a helper to mix data objects from two or more sources into one stream. When mixed, dataobjects are interleaved. For example:

```
>>> from processor import sources
>>> source1 = [1,2,3]
>>> source2 = [5,6,7,8]
>>> print(list(sources.mix(source1, source2)))

[1, 5, 2, 6, 3, 7, 8]
```

Mix source iterates through each given source until it raises StopIteration. That means, if you'll give it an infinite sources like a *web.hook*, then resulting source also will be infinite.

6.1.2 imap

Imap source is able to read new emails from specified folder on IMAP server. All you need is to specify server's address, optional port and user credentials:

Example:

```
from processor import run_pipeline, source, outputs
run_pipeline(
    sources.imap("imap.gmail.com",
                           "username",
                           "****word",
                           "Inbox"),
    outputs.debug())
```

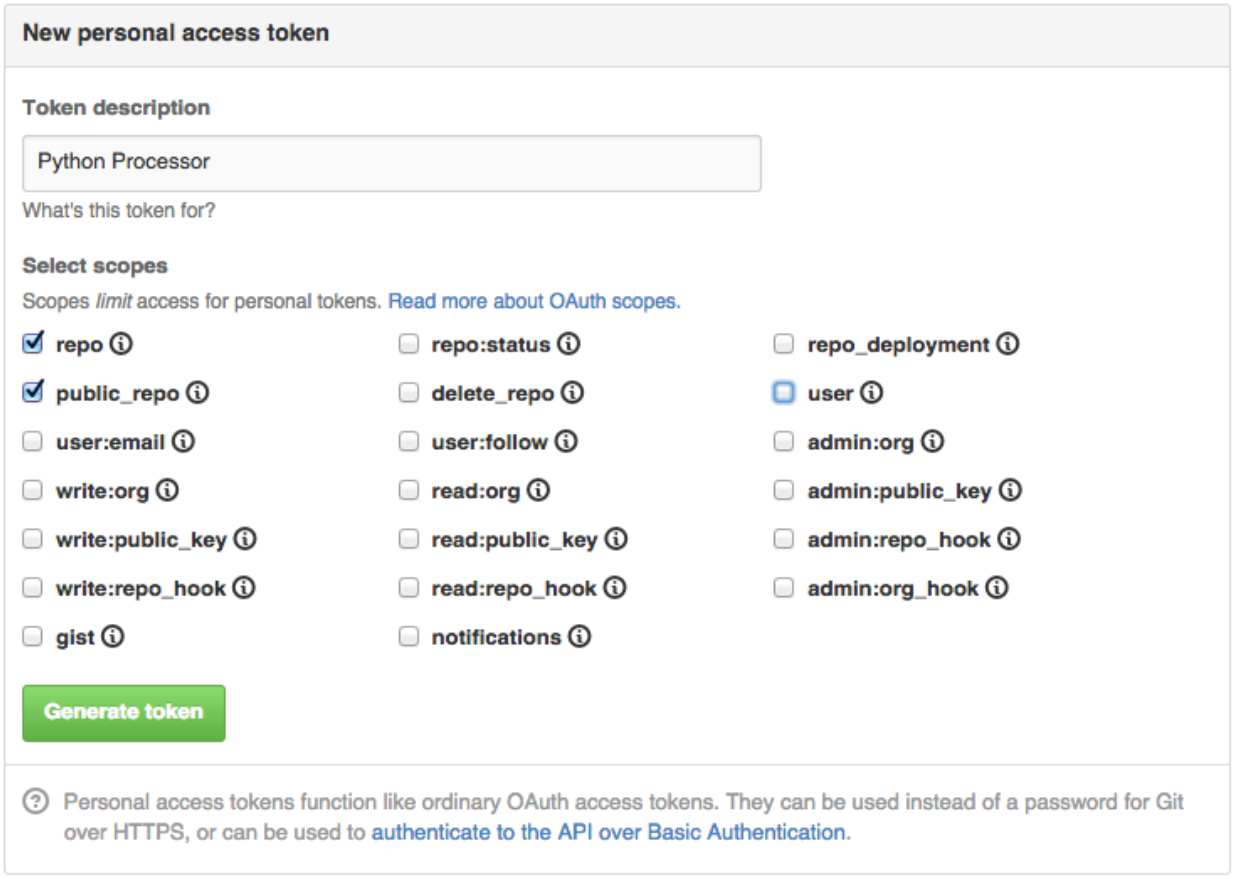
This script will read `Inbox` folder at server `imap.gmail.com` and print resulting dicts to the terminal's screen.

6.1.3 github

Access to private repositories

To have access to private repositories, you need to generate a "personal access token" at the GitHub.

All you need to do this, is to click on the image below and it will open a page with only scopes needed for the Processor:



Then copy this token into the clipboard and pass it as a `access_token` parameter to each `github.****` source.

Note: Access token not only let the processor read from private repositories, but also makes rate limits higher, so you could poll GitHub's API more frequently.

Without token you can make only 60 request per hour, but with token – 5000 requests per hour.

---

github.releases

Outputs new releases of the given repository. On first call, it will output all the most recent releases, then remeber position on next calls will return only new releases if any were found.

Example:

```
from processor import run_pipeline, source, outputs

github_creds = dict(access_token='keep-it-in-secret')
run_pipeline(
```

```
    sources.github.releases('https://github.com/mozilla/metrics-graphics', **github_creds),
    outputs.debug())
```

This source returns following fields:

source github.releases

type github.release

payload The object returned by GitHub's API. See section "Response" at GitHub's docs on repos/releases.

6.1.4 twitter

Note: To use this source, you need to obtain an access token from twitter. There is a detailed instruction how to do this Twitter's documentation. You could encapsulate twitter credentials into a dict:

```
twitter_creds = dict(consumer_key='***', consumer_secret='***',
                     access_token='***', access_secret='***')
sources.twitter.search('Some query', **twitter_creds)
sources.twitter.followers(**twitter_creds)
```

twitter.search

This source runs search by given query in Twitter and returns fresh results:

```
from processor import run_pipeline, source, outputs
run_pipeline(
    sources.twitter.search('iOS release notes', **twitter_creds),
    outputs.debug())
```

It returns following fields:

source twitter.search

type twitter.tweet

other Other fields are same as them returns Twitter API. See section "Example Result" at twitter's docs on search/tweets.

twitter.followers

First invocation returns all who you follows, each next – only new followers:

```
from processor import run_pipeline, source, outputs
run_pipeline(
    sources.twitter.followers(**twitter_creds),
    outputs.debug())
```

It returns following fields:

source twitter.followers

type twitter.user

other Other fields are same as them returns Twitter API. See section "Example Result" at twitter's docs on followers/list.

6.1.5 web.hook

This source starts a webserver which listens on a given interface and port. All GET and POST requests are transformed into the data objects.

Configuration example:

```
run_pipeline(sources.web.hook(host='0.0.0.0', port=1999),
             outputs.debug())
```

By default, it starts on `localhost:8000`, but in this case on `0.0.0.0:1999`.

Here is example of data objects, produced by this source when somebody posts JSON:

```
{'data': {'some-value': 0},
 'headers': {'Accept': 'application/json',
   'Accept-Encoding': 'gzip, deflate',
   'Connection': 'keep-alive',
   'Content-Length': '17',
   'Content-Type': 'application/json; charset=utf-8',
   'Host': '127.0.0.1:1999',
   'User-Agent': 'HTTPie/0.8.0'},
 'method': 'POST',
 'path': '/the-hook',
 'query': {'query': ['var']},
 'source': 'web.hook',
 'type': 'http-request'}
```

This source returns data objects with following fields:

source web.hook

type http-request

method GET or POST

path Resource path without query arguments

query Query arguments

headers A headers dictionary. Please, note, this is usual dictionary with case sensitive keys.

data Request data, if this was a POST, None for GET. If requests has `application/json` content type, then data decoded automatically into the python representation. For other content types, if there is charset part, then data is decoded from bytes into a string, otherwise, it remains as bytes.

Note: This source runs in blocking mode. This means it blocks `run_pipeline` execution until somebody interupt it.

No other sources could be processed together with `web.hook`.

6.2 Outputs

6.2.1 debug

This output is very useful for debugging you input. All it does right now – returns `pprint` function, but possible interface will be extended in future to select which fields to ouput or suppress, cache or something like that.

6.2.2 fanout

Fanout output is useful, when you want to feed one data objects stream to two or more pipelines. For example, you could send some events by email and into the *slack* chat simultaneously:

```
run_pipeline(some_source(),
             outputs.fanout(
                outputs.email('vaily@pupkin.name'),
                outputs.slack(SLACK_URL)))
```

Or if you need to preprocess data objects for each output, then code will looks like this:

```
run_pipeline(some_source(),
             outputs.fanout(
                [prepare_email, outputs.email('vaily@pupkin.name')],
                [prepare_slack, outputs.slack(SLACK_URL)]))
```

Where `prepare_email` and `prepare_slack` just a functions which return data objects with fields for *email* and *slack* outputs.

6.2.3 email

Sends an email to given address via configured SMTP server. When configuring, you have to specify `host`, `port`, `user` and `password`. And also a `mail_to`, which is an email of recipient who should receive a message and `mail_from` which should be a tuple like `(name, email)` and designate sender. Here is an example:

```
run_pipeline(
  [{'subject': 'Hello from processor',
    'body': 'The <b>HTML</b> body.'}],
  outputs.email(mail_to='somebody@gmail.com',
                mail_from=('Processor', 'processor@yandex.ru'),
                host='smtp.yandex.ru',
                user='processor',
                password='***',
                port=465,
                ssl=True,
            ))
```

Each data object should contain these fields:

subject Email's subject

body HTML body of the email.

6.2.4 rss

Creates an RSS feed on the disk. Has one required parameter – `filename` and one optional – `limit`, which is `10` by default and limiting result feed's length.

Each data object should contain these fields:

title Feed item's title.

id (optional) Feed item's unique identifier. If not provided, then md5 hash from title will be used.

body Any text to be placed inside of rss item's body.

6.2.5 slack

Write a message to Slack chat. A message could be sent to a channel or directly to somebody.

This output has one required parameter `url`. You could obtain it at the Slack's integrations page. Select "Incoming WebHooks" among all available integrations. Add a hook and copy it's `url` into the script. Other parameter is `defaults`. It is a dict to be merged with each data object and by default it has `{"renderer": "markdown", "username": "Processor"}` value.

Each data object should contain these fields:

text Text of the message to be posted. This is only required field. Other fields are optional and described on Slack's integration page.

username (optional) A name to be displayed as sender's name.

icon_url (optional) A link to png icon. It should be 57x57 pixels.

icon_emoji (optional) An emoji string. Choose one at **'Emoji Cheat Sheet'_**.

channel A public channel can be specified with `#other-channel`, and a Direct Message with `@username`.

6.2.6 XMPP

XMPP output sends messages to given jabber id (JID). It connects as a Jabber client to a server and sends messages through it.

Note: If you use Google's xmpp, then you will need to add Bot's JID into your roster. Otherwise, messages will not be accepted by server.

This output is configured by three parameters `jid`, `password` and `host`. They are used to connect to a server as a jabber client. Optionally, you could specify `port` (which is 5222 by default) and `recipients` – a list of who need to be notified. Recipients list could be overriden if data object contains field `recipients`.

Each data object should contain these fields:

text Text of the message to be posted.

recipients (optional) A list of JIDs to be notified.

6.3 Contributing

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

6.3.1 Bug reports

When reporting a bug please include:

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

6.3.2 Documentation improvements

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

6.3.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/svetlyak40wt/python-processor/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.3.4 Development

To set up *python-processor* for local development:

1. Fork python-processor on GitHub.
2. Clone your fork locally:

```
git clone git@github.com:your_name_here/python-processor.git
```

3. Create a branch for local development:

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

   Now you can make your changes locally.

4. When you're done making changes, run all the checks, doc builder and spell checker with tox one command:

```
tox
```

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

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

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`) [1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

4. Add yourself to `AUTHORS.rst`.

Tips

To run a subset of tests:

```
tox -e envname -- py.test -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

6.4 Authors

- Alexander Artemenko - http://dev.svetlyak.ru

6.5 Changelog

6.5.1 0.9.0 (2015-12-06)

Code was fixed to work with HyLang from `a3bd90390cb37b46ae33ce3a73ee84a0feacce7d` commit. Please, use this pinned version of HyLang and subscribe on future release notes to know when this requirement will change.

6.5.2 0.8.0 (2015-11-16)

- Code was fixed to work with latest Hy, from GitHub.
- Added `twitter.mentions` source, to read stream of mentions from the Twitter.
- Fixed a way how number of messages from IMAP folder is limited. Previously limit was applied even when we already know an ID of the last seen message, but now limit is ignored in this case and only applied when visiting the folder first time.

6.5.3 0.7.0 (2015-05-05)

New output – XMPP was added and now processor is able to notify Jabber users.

6.5.4 0.6.0 (2015-05-01)

The biggest change in this release is a new source – `github.releases`. It is able to read all new releases in given repository and send them into processing pipeline. This works as for public repositories, and for private too. Read the docs for futher details.

Other changes are:

- Storage backend now saves JSON database nicely pretty printed for you could read and edit it in your favorite editor. This is Emacs, right?

- Twitter.search source now saves state after the tweet was processed. This way processor shouldn't loose tweets if there was exception somewhere in processing pipeline.
- IMAP source was fixed and now is able to fetch emails from really big folders.

6.5.5 0.5.0 (2015-04-15)

Good news, everyone! New output was added - `email`. Now Processor is able to notify you via email about any event.

6.5.6 0.4.0 (2015-04-06)

- Function `run_pipline` was simplified and now accepts only one source and one ouput. To implement more complex pipelines, use `sources.mix` and `outputs.fanout` helpers.

6.5.7 0.3.0 (2015-04-01)

- Added a web.hook source.
- Now *source* could be not only a iterable object, but any function which returns values.

6.5.8 0.2.1 (2015-03-30)

Fixed error in `import-or-error` macro, which prevented from using 3-party libraries.

6.5.9 0.2.0 (2015-03-30)

Most 3-party libraries are optional now. If you want to use some extension which requires external library, it will issue an error and call `sys.exit(1)` until you satisfy this requirement.

This should make life easier for thouse, who does not want to use `rss` output which requires `feedgen` which requires `lxml` which is hard to build because it is C extension.

6.5.10 0.1.0 (2015-03-18)

- First release on PyPI.