tammy Documentation

Release 0.0.1

Timothée Poisot

July 22, 2014

Contents

`tammy` is a python module to manage your bibliography in a sane, minimalistic, scriptable, hacker-ish way.:

```
>>> import tammy
>>> lib = tammy.library()
>>> lib.new(tammy.from_crossref_doi('10.7717/peerj.426'))
>>> lib.keys()
['leh14']
>>> lib.records['leh14'].generate_key(tammy.keygen.AuthorYear)
>>> lib.keys()
['Lehiy2014']
>>> lib.new(tammy.from_peerj('403', 'preprint'))
>>> lib.write()
```

CHAPTER 1

User guide

1.1 Introduction

1.1.1 Why tammy?

tammy is a python 3 module that allows managing your bibliography in a simple way. I was dissatisfied by all the tools I used, so I decided to build my own. I was looking for a reference manager that...

Can be used programmatically

Being able to manage my references in a programmatic way is important to me, because it means that I can automate a lot of things. Importing, exporting, etc etc. And because tammy is essentially an API to manage bibliographic records, along with a few helper functions, automation is easy.

Works well with unicode

Us Europeans tend to have accents in our names. BibTeX is bad at that, and unicode conversion is in my experience the first source of screw-ups when opening a bib file in different programs. I wanted something that would play nicely with unicode.

Is not based on a database

There were two things I absolutely wanted to avoid: relying on a database, and relying on a single, massive file. I wanted something light, that I can easily manipulate using grep and other nice things if I feel like it. And because there is no reason that a single corrupted record should render your whole database useless, I decided to assign each record to its own file, and build export functions instead.

Integrates in my pandoc-based workflow

I use pandoc and markdown to write papers. pandoc can read bibliography files in citeproc-JSON, so using bibtex as a storage format makes very little sense (see also: unicodes, and the fact that it's not the 1980s anymore).

Can do web search

Getting the informations on a paper with just a `DOI`, `PMID`, `ArXiV` identifier, etc, is useful, so I am building a set of functions to do that. Also, if journals expose their papers in citeproc-JSON (as *PeerJ* does), it's easy to write a function for integration.

With all these informations in hand, if you think `tammy` is right for you, read on!

1.2 First steps

This page will show you the very first steps of installing and configuring `tammy` to your liking.

1.2.1 Installing `tammy`

1.2.2 Configuration file

There are three possible locations for the configuration file. First, wherever you feel like, as the `library` class accepts a `cfile` argument with a path. Second, in the directory in which you are currently working. Finally, in your `$HOME`. Note that in the last two situations (I expect that the later is the standard), the file *must* be called `.tammy.yaml`.

At the moment, the only configurable options are the `bib_dir` and `export_dir` variables, which will give respectively the the roots of your library, and where to export lists of references. By default, your library lives in `$HOME/.bib`. You can change it with

```
bib_dir: $HOME/.references
```

Also by default, the `export_dir` is `$HOME/.pandoc`, so that the files generated can be used directly from `pandoc`.

When `tammy` will read the content of your library, it will go look for references here. Over time, I will add options for the default citation key format (currently `AutYr`), and things related to the maybe-coming-soon `ncurses` interface.

1.2.3 The bib folder

For the moment, `tammy` will *assume* that the `bib_dir` folder has two sub-folders, called `records` and `files`. There is *currently* no check for the presence of these sub-folders, so crashes are to be expected if this is not the case. Is that poor design? For sure. Will it change? Hopefully. Is it hard to do? Not even, no. That's just how I roll.

1.2.4 Creating a first library

Whether or not you already have records on the disk, creating a bibliography is as simple as:

```
>>> import tammy
>>> my_lib = tammy.library()
```

Note that the term *creating* is misleading: your library won't be re-created every time, because it doesn't *exist* outside of your session. Rather, the `python` objects that allow you to interact with it will be created. Loading a lot of records *can* take some time, but it's a one-time thing. Future operations are really fast.

1.3 A short note about design

library / record objects

1.4 Adding references

This page will go over how to add records to your bibliography. Creating a new record is done through the `new` method of the `library` class (it actually creates an object of class `record`). Whenever a new object is created *by the user* (remember that one object is created for each file in the bibliography folder when starting up), two things happen. First, the object is checked, and a unique `id` (citation key) is generated. Then, a new file is created to store this reference.

1.4.1 From a DOI

Most *recent* papers have a Digital Object Identifier. It's a Good Thing. Whenever you have the choice, and unless there is a special function for the database you are querying, import things from their DOI. This is simple:

```
>>> my_doi = 'doi.journal/xx.xxx.xxxxx'
>>> record = tammy.from_crossref_doi(my_doi)
>>> lib.new(record)
```

Note that the function is called `from_crossref_doi`, because at the moment, the *CrossRef* API is the easiest way to get a `json` output from a DOI. Essentially, these commands will (i) get the record as a `json` string (it will be converted in a `python` object on the fly), then add this object to the library.

1.4.2 From PeerJ

1.4.3 From a file

Nothing prevents you from manually writing a reference (besides mental sanity and a sense of priorities, that is). Any `.yaml` file in the `$bib_dir/records` will be read and parsed when the `library` is created. Alternatively, you *can* do:

```
>>> import json
>>> with ('my/file/ref.json', 'r') as ref_file:
...   record = json.load(ref_file)
>>> lib.new(record)
```

This will read a `json` file, and create a new record from it.

1.5 Writing the library to disk

One important thing to keep in mind is that, unless explicitely asked, `tammy` will *not* write the contents of the records to the disk. Actually, `tammy` will never manipulate the library as a whole, only apply actions to the different records.

There are two ways to write *something* to disk.

1.5.1 Exporting the library

The `library` class has an `export` method, that allows you to export either the entire content of the library, or only some keys, to the disk.:

```
my_lib.export()
```

Without any arguments, this will write a `default.json` file to the path specificied in the `export_dir` configuration option. Available options are `keys` (a list of keys to export), `path` (where the exported file will be), and `output`. Output can be one of the values in `tammy.IO.serializers`, and defaults to `citeproc-json`.

Despite its simplicity, this function is all you need to export a file with your references. If you want all citations published in `Nature` in a `citeproc-yaml` file in your `~/Dropbox` folder, then this is:

```
keys = [k for k, v in my_lib.records.items() if 'container-title' in v.content and v['container-title
my_lib.export(keys=keys, path="~/Dropbox", output="citeproc-yaml")
```

As long as you have some familiarity with the citeproc format, selecting the keys you want is relatively easy.

1.6 Dealing with attachments

CHAPTER 2

Developer guide

2.1 Classes

2.1.1 Library

class `tammy.`**`library`**(*cfile=None*)

`read`(*force=False*)
Read the yaml files from the references folder

This method is called when the `library` class is instanciated, and it ensures that all records are loaded. Because it calls the `new` method of the `record` class, if for some weird reason a file has no `id` field (*e.g.* you added it yourself), the key will be generated at this point.

Args: force: a boolean to force the method to read all files, or only ...

`update`()
Update the keys in the library dict

This function will loop through all the references, and if the record id do not match with the key in the records dict, it will fix things up. Additionally, this function will rename the file in the references folder, and make sure that the linked files are renamed too.

2.1.2 Record

class `tammy.`**`record`**(*library*, *content*, *new=True*)

`generate_key`(*keymaker=<function autYr at 0x7f36bf5d1950>*)
Generates a citation key from the record information

At the moment, citations keys are created as autyr scheme plus one letter if this is required to make the citation key unique. Note that the citation key is also the filename of the record, so that a record whose key is `Doe2004` will be written at `Doe2004.yaml`.

Args: keymaker: a function returning a string with the record id, based on the contents of the record.

`key`()
Outputs the unique citation key for the record

Returns: a unicode string with the citation key

`write``()`

Writes the content of a record to disk

This will write the content of the record in the `records` folder of the `bib_dir` folder. The filename is the unique record key and the `.yaml` extension.

This method is usually called by `library.write()`, but it can be used to update the content of any file.

2.2 Key generators

CHAPTER 3

Indices and tables

- *genindex*
- *modindex*
- *search*

Python Module Index