Writing bibliographic tools with *pybliographer*

Frédéric Gobry

August 18, 2007

Contents

Chapter 1

Introduction

pybliographer is a developer-oriented framework for manipulating bibliographic data. It is written in *python*[1], and uses extensively the dynamic nature of this language.

pybliographer does not try to define another standard format for bibliographic data, nor does it solely rely on a single existing standards. Standards are important in order to allow for interoperability and durability. Unfortunately, real-world data often contain mistakes (sometimes systematic mistakes due to a misunderstanding of the meaning of a field), or reflect certain local conventions which are not part of a standard. *pybliographer* is on the *pragmatic* side of considering these issues as part of its business: most of the parsing and processing tasks it performs can be easily overridden and specialized in order to *fit the code to the data*, and not the other way around.

1.1 Basic concepts

pybliographer deals with sets of `Record`s, stored in a so-called `Database`. This database can be actually implemented on top of different systems. Three are available today:

- in-memory: useful when the data is converted from one format to another and doesn't need to be stored in *pybliographer*.
- file: as a single XML file, using a custom XML dialect, suitable for small to medium databases (thousands of records).
- Berkeley DB[2]: this is a very efficient database system, suitable for larger databases. In this case the limitations are due to some *pybliographer* design decisions, and should be reached after a million records or so.

Each record represents an elementary object you want to describe, and has a number of *attributes*. For instance, if you are describing a book, one attribute will be its *title*, another its *ISBN*, etc. Each of these attributes can contain one or more values, all of the same *type*. To continue the description of our book, we

[1] see `http://python.org/`
[2] see `http://www.sleepycat.com/`

probably have the *author* attribute, which contains as many `Person` values as there are authors for the book. All the values of a given attribute are of the same type.

In some cases, simply having this flat key/value model to describe an object is not enough. *pybliographer* allows every value of every attribute to provide a set of *qualifiers*. These qualifiers are also attributes which can hold one or more values. If my book, or information about the book, is available via the internet, I can provide a *link* attribute, but for each of the actual `URL`s provided, I might wish to add a *description* qualifier, which will indicate, say, if the URL points to the editor's website, or to a review, etc.

This nesting of objects is best described in figure 1.1.

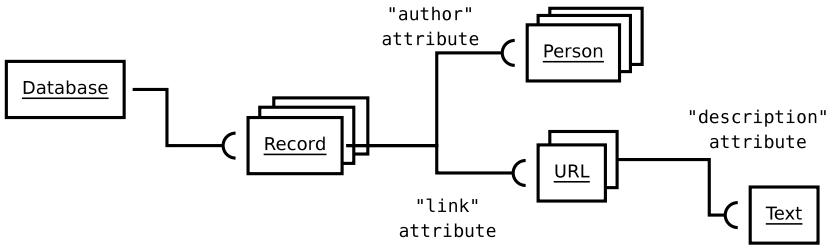


Figure 1.1: Objects manipulated in *pybliographer*

pybliographer comes with a set of defined attribute types, like `Person`, `Text`, `Date`, `ID` (see the `Pyblio.Attribute` module for a complete list), and can be extended to support your own types.

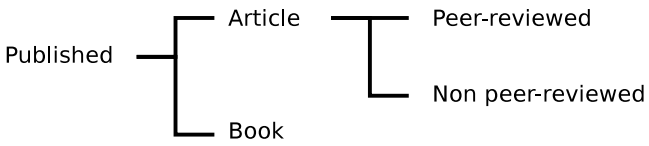
1.1.1 The database schema

Even though attributes are typed, the data model described above is quite flexible. In order for *pybliographer* to help you checking that your records are properly typed, it needs to know the database schema you are using. This schema, usually stored in an XML file with the extension `.sip`, simply lists the known attributes with their type and the qualifiers it allows for its values. Some `.sip` files are distributed with *pybliographer*, and can be seen in the `Pyblio.RIP` directory.

In addition to validation information, the schema contains human-readable description of the different fields, possibly in several languages, so that it can be automatically extracted by user interfaces to provide up-to-date information.

1.1.2 Taxonomies

Taxonomies can be used as *enumerated values*, say for listing the possible types of a document, or the language in which a text is written. They have however the extra capability of being hierarchical: you can define subcategories of a main category. For instance, imagine a `doctype` taxonomy with the following values:



You can tag an article as `Peer-reviewed`, but you are not required to use the *leaf* values in this tree. In the case you don't know if a publication is reviewed or not, you can use the `Article` tag. Similarly, if you search for all the `Published` documents, you will retrieve all those that have the `Published` tag, but also those that are articles (either peer-reviewed or not), books,...

pybliographer uses the `Pyblio.Attribute.Txo` object to *represent* a logical value in a given taxonomy. A record can be tagged with this `Txo` object by adding a `Pyblio.Schema.TxoItem` value in the corresponding attribute.

Taxonomies are declared in a database schema, and thus cannot change unless you change the schema itself.

1.1.3 Result sets

Result sets are used to manipulate an explicit list of records, among all the records kept in a database. They are returned from queries on the database, and can be manipulated by the user. Result sets are somewhat like mathematical sets, as you cannot put duplicate values in them, and they have no default ordering of their elements. You can create result sets via the `rs` attribute of your database, which is an instance of the `Pyblio.Store.ResultSetStore`.

A special result set is available as `Pyblio.entries`, and contains at every time **all** the records of the database.

1.1.4 Views

We have seen that result sets are **not** ordered. However, in many cases, one needs to provide the records in a specific order. To do so, you can create a *view* on top of a result set. This view is created by calling the `view` method of the result set, with an `order` parameter being the description of the sort order you wish to have. The module `Pyblio.Sort` provides elementary constructs to build such a description.

Once the view is created, modifying the corresponding result set leads to updating the view accordingly.

1.2 Manipulating data

This section describes some simple operations you can perform on some subset of a *pybliographer* database.

1.2.1 Loading and saving

The first thing you need to do is of course *actually having* a database available. The following code does the job:

```
from Pyblio import Store, Schema

schema = Schema.Schema('myschema.sip')
store = Store.get('file')

db = store.dbcreate('mydb.bip', schema)
```

This example relies on the fact that you already have a schema at hand. There are schemas available in the `Pyblio.RIP` directory. It the starts by reading the schema. The next step is to select the actual physical store which will hold your database. We choose to store it in a simple XML file, whose canonical extension is `.bip`. The last operation actually creates the database with the specified schema.

Independently of the selected store, it is always possible to *export* a database in the `.sip` format, by calling the `db.xmlwrite(...)` method of the database. Such a file can then be reused later on by using `store.dbimport(...)` instead of `store..dbcreate(...)`.

When you have finished modifying your database, you can call `db.save()` method to ensure that it is properly saved.

Caution: the bsddb store for instance is updated at every actual modification, not only when you call the `save` method. Don't rely on it to provide some kind of *rollback* feature.

1.2.2 Using the registry

pybliographer has a mechanism to register known schemas, and specify which import and export filters can properly work with each schema. This mechanism can be used to create our database by asking for a specific schema, as shown below:

```
from Pyblio import Store, Registry

Registry.load_default_settings()

schema = Registry.getSchema("org.pybliographer/bibtex/0.1")
store = Store.get('file')

db = store.dbcreate('mydb.bip', schema)
```

The registry must be first initialized. Then you can ask for a specific schema, in that case a schema that supports BibTeX databases.

1.2.3 Updating records

The next example will loop over all the records in a database, and add a new author to the list of authors.

```
from Pyblio import Attribute

for record in db.entries.itervalues():
    person = Attribute.Person(last=u"Gobry",
```

```
                                first=u"Frédéric")

    record.add('author', person)

    db[record.key] = record

db.save()
```

We use the `itervalues()` iterator to loop over all the records stored in the database. Then, we simply insert a new value in the `author` attribute. The `record.add(...)` method takes care of creating the attribute if it does not exist yet.

One thing not to forget is to store the record back in the database once the modification is performed. Without this step, you might experience weird behavior where some modifications are not properly kept.

We finish by saving the database.

1.2.4 Sorting

To sort records, you create *views* (see section 1.1.4 on page 4). You can of course create multiple views on top of a single result set. In order to sort the whole database, simply create the view on `database.entries` instead of a result set. If you want to sort your database by decreasing year and then by author, you can use a view like that:

```
from Pyblio.Sort import OrderBy

view = db.entries.view(OrderBy('year', asc=False) &
                       OrderBy('author'))

for record in view.itervalues():
    # do something with the record
    # ...
```

So, sorting constraints can be arbitrarily chained with the `&` operator,and each constraint can be either *ascending* (the default), or *descending*. This defines a very simple *Domain Specific Language*, or DSL for short. Such languages also appear in other part of *pybliographer* (searching, citation formatting), as they are a convenient way to describe complex abstraction without having to reinvent a complete environment.

1.2.5 Searching

To search, you call the `database.query(...)` method. The method takes a query specification as argument, which is constructed with the help of another DSL, similar to the one used for sorting. You have access to a certain number of primitive queries, which are then linked together with the usual boolean operators, as in the following example:

```
from Pyblio import Query
```

```
article = db.txo['doctype'].byname('article')

result = db.query(~ Query.Txo('doctype', article) &
                  Query.AnyWord('laziness'))
```

We first get the taxonomy item corresponding to articles, and we then compose the following query: get all the documents that are *not* articles, and which contain the word *laziness* in any attribute.

1.3 Importing and exporting

As *pybliographer* is not bound to a single data schema, importing and exporting from specific formats (like MARC, BibTeX, Dublin Core,...) cannot be achieved once for all. In order to avoid the need to recreate a BibTeX parser for every database schema invented, *pybliographer* makes a clear separation between *syntactic parsers*, located in `Pyblio.Parsers.Syntactic` and *semantic parsers*, in `Pyblio.Parsers.Semantic`. A syntactic parser is only in charge of analyzing or generating a file format, without any assumption regarding the meaning of the fields it reads. These syntactic parsers are then reused by the semantic code, which relates the meaning of the fields to the corresponding database.

In addition, the parsers are written so that the handling of separate fields can be easily overridden in a subclass. This makes it possible to extend them or take some local *specificities* into account (if you need to massage data that contains systematic errors, this proves *very* useful).

The following example assumes you have created a BibTeX-compatible database, as explained in the section 1.2.2 on page 5. It will then open a proper BibTeX file, and merge it into the current database. The list of imported references is returned as a result set.

```
from Pyblio.Parsers.Semantic import BibTeX

parser = BibTeX.Reader()

rs = parser.parse(open('example.bib'), db)
```

1.4 Citation formatting

The *painful* part of writing citation formatting code is to take into account incomplete records (sometimes you don't know the volume or the pages, or you only know one of the two,...) without multiplying explicit checks that would quickly be boring. In addition, it is important to make it easy to factor out common operations, like formatting a list of authors, so that you can reuse them in different contexts.

pybliographer provides a *domain specific language* that addresses these problems. A domain specific language (or DSL for short) is a language specifically intended to solve a given problem, but which is usually built up from a more general language. In our case, it means that *pybliographer* provides a set of

classes, functions and constructs that are highly specialized to make the business of writing the citation code easy. The beauty of the idea is that, in case of a missing feature in this DSL, you still have all the power of python at your fingertips.

In any case, this DSL is not intended as a complete formatting language, so you cannot use it to lay out your citations in a full blown HTML web page for instance. However, once a citation is built up from a record, the specific part of putting it in a larger context is comparatively easy.

Back to practice. You can define some citation fragments like this:

```
from Pyblio.Format import People, all, one

authors = People.lastFirst(all('author'))
title = one('title') | u'(no title)'
```

In this example, the `authors` variable is build up by taking all the values in the author field (`all('author')`), and by passing them through the `lastFirst` function, which will format them as *Last Name, First Name*. The `Person` module contains other formatting variants for person names if you want to use initials for instance.

The `title` variable is built by taking the first value of the title field (via the `one` operator), and in case it does not exist, by using the string *no title* instead. This `|` alternative operator can be used everywhere to express a fallback value where a definition can be invalid.

You can then group the authors and the title together, possibly while adding some typographic style information in the process:

```
from Pyblio.Format import B

citation = join(', ')[B[title], authors]
```

The `join` operator will take the parts between square braces and bind them together with the text specified in parameter, a comma in that case. When one of the composing parts is not available, it is simply ignored, unless no part is available, in which case the whole expression is invalid (which can be trapped by using the `|` operator). In the example, the title is enclosed in a bold `B` tag.

Once the citation style is defined, it must be *compiled* on a specific database:

```
formatter = citation(db)
```

This operation checks that all the fields accessed are actually part of the schema. It also pre-computes certain information, so that the actual formatting of specific records can be faster.

Then, you can use the returned formatter and apply it to any number of records from the corresponding database:

```
cited = formatter(record)
```

You still don't get a definitive result, as you need to select the output format for your citation. If you want it in HTML, you can do this last operation:

```
from Pyblio.Format import HTML

html = HTML.generate(cited)
```

Now, `html` contains a properly escaped HTML fragment which you can use in your own context.

1.5 Querying external databases

pybliographer has a standard interface for querying external databases, like *PubMed* or the *Web of Science*. These queries rely on the asynchronous `twisted` library, which makes it possible to run such a query from a graphical interface without blocking, and to interrupt it easily.

An example of such a query is described below.

```
from Pyblio.External import WOK

s = Registry.getSchema('org.pybliographer/wok/0.1')
db = Store.get('file').dbcreate(output, s)

wok = WOK.WOK(db)

d, rs = wok.search(query)

def success(total):
    print "wok: successfully fetched %d records" % total
    # do something with the database?

d.addCallback(success).addErrback(failure)

reactor.run()
```

In this code, a query object `wok` is created, which will directly modify the database passed in parameter. Then a actual query is registered by calling `wok.search`. This returns two values: a *deferred object*, which is a twisted abstraction. You can then *plug* a callback, in our case the `success()` function, to be called when the search succeeds. The second value returned is a result set, which will be filled with the entries retrieved by the query.

Note that so far the query did not run. To start it, you need to run *twisted*'s main loop, called the reactor. This function won't exit unless you call `reactor.stop()` somewhere. Please have a look at *twisted*'s documentation[3] to learn how to make use of this powerful framework.

1.6 Adapting to another schema

In the preceding example, we fetched results from the Web of Science in a database of type `org.pybliographer/wok/0.1`. This is a schema specially crafted for records coming from this database, but usually this is not what you expect to store in your own, say, BibTeX database. So what, do you need to create another query engine specially for your database? Fortunately no, you can use the *adaptation mechanism* to make this Web of Science database look like a BibTeX database with a simple call like:

[3] see `http://twistedmatrix.com/`

```
from Pyblio import Adapter

bibtex = Adapter.adapt_schema(
      db, 'org.pybliographer/bibtex/0.1')
```

When this call succeeds, you will get in return a new database called `bibtex`, which will contain everything contained in `db`, but in the BibTeX schema. Please note that your database is *not* duplicated, the `bibtex` database is just some kind of overlay that behaves as a normal database, but uses the initial content dynamically.

Unfortunately, these adapters objects that do the mapping do not come out of thin air, and need to be registered in the system, via the usual `.rip` registry mechanism.

Chapter 2

Extending *pybliographer*

2.1 Specializing a parser

Let's say your BibTeX file uses a field called `status` that is not standard. You need to create a new schema that declares it, and derive the base BibTeX parser to provide an extra field handler:

```
class MyBibTeXReader(Semantic.BibTeX.Reader):
    def do_status_field(self, key, value):
        # do things for the field "status"
```

Once this is done, you can register the new reader in a RIP file.

In some cases, it is necessary to perform cross-field checks and modifications. This can be achieved by using the following simple extension hooks:

`Parser.record_begin(self)` is invoked at the beginning of each record.

`Parser.record_end(self)` is invoked once all the fields of a record have been parsed.

`Parser.do_default(self, field, value)` will be invoked for unknown fields.

2.2 Writing an external query engine