# python-glyr Documentation

***Release 1.0.2***

**Christoper Pahl**

June 10, 2015

Contents

Contents:

CHAPTER 1

# Salve te Wanderer!

## 1.1 What is this about?

This is the documentation to **plyr**, a Wrapper around **libglyr**, a library for finding and downloading all sort of music-related metadata from a collection of providers. It's written in C and therefore available for musicplayers written directly in that language, or for programs that used the commandline interface **glyrc**.

## 1.2 Useful links

**libglyr:**

https://github.com/sahib/glyr

**libglyr's API doc:**

http://sahib.github.com/glyr/doc/html/index.html

**plyr on github:**

https://github.com/sahib/python-glyr

**plyr on PyPI:**

http://pypi.python.org/pypi/plyr

## 1.3 Installing

**Using PyPI:**

Use *pip* to install the package from source:

```
sudo pip install plyr
```

**Manual Installation (most recent):**

Install libglyr if not done yet. Either..

- ... compile from Source: https://github.com/sahib/glyr/wiki/Compiling
- ... or use the package your distribution provides.
- ... in doubt, compile yourself. I only test the latest version.

Install *cython* if not done yet:

```
sudo pip install cython
```

Build & install the Wrapper:

```
git clone git://github.com/sahib/python-glyr.git
cd python-glyr
sudo python setup.py install
```

## 1.4 Documentation?

Silly question. You're looking at it. But when we're on it: There are only a few chapters, since there is not so much to cover. Every chapter is split into a description, and a reference. After all those theory chapters you are going to be rewarded by some practical examples.

Please have *fun*.

## 1.5 Other things to note?

Please use a own useragent, if you integrate glyr into your application. You can set it via:

```
qry.useragent = 'projectname/ver.si.on +(https://project.site)'
```

Why? In case your application makes strange things and causes heavy traffic on the provider's sites, they may ban the user-agent that makes this requests. So, only your project gets (temporarly) banned, and not all libglyr itself.

CHAPTER 2

# Building Queries

## 2.1 A minimal Example

Whenever you want to retrieve some metadata you use a Query:

```
# one import for everything
import plyr

# A query understands lots of options, they can be either passed
# on __init__ as keywords, or can be set afterwards as properties
qry = plyr.Query(artist='Tenacious D', title='Deth Starr', get_type='lyrics')

# Once a query is readily configured you can commit it
# (you can even commit it more than once)
items = qry.commit()

# Well, great. Now we have one songtext.
# Since libglyr may return more than one item,
# commit() always returns a list. The default value is 1 item though.
# The list contains ,,Caches'', which are basically any sort of metadata.
print(items[0].data)
```

## 2.2 Accessing Default Values

Default Values for any option can be accessed by instantiating an empty Query, and using the provided properties.

## 2.3 Property Reference

CHAPTER 3

# Retrieved Items (aka „Caches'')

## 3.1 What is a Cache?

Basically every single bit of metadata returned from **libglyr** is a Cache. It is basically a *result* or it is sometimes referred to as an item. It encapsulates basic attributes like a bytearray of data, a checksum, an optional imageformat and some more.

## 3.2 What can I do with one?

Caches can be...

- ... committed to a local database (see next section)
- ... written to HD via it's `write()` method.
- ... printed to stdout via `print()`, currently `__repr__` is not implemented.

You can use the *data* property to get the actual metadata. This is always a bytestring, even with lyrics. You have to convert it to the desired encoding (e.g. utf-8) like this:

```
str(some_cache.data, 'utf-8')
```

**Note:** This might throw an UnicodeError on non-utf8 input, or imagedata. Use the Query-Property **force_utf8** to only retrieve valid utf-8 textitems.

The *data* property is settable. Setting will also cause the size and checksum to be adjusted.

**Note:** Cache implements the __eq__ and __ne__ operator, which will compare only the data properties. This is meant to be a shortcut, since comparing data is usually the only thing you want to compare.

## 3.3 Reference

CHAPTER 4

# Looking up Providers

`plyr.PROVIDERS` contains a nested dict modelling all compiled in fetchers and providers.

```
'albumlist': {                                  # The name of this fetcher
             'required' : ('artist'),           # A list of required properties
             'optional' : (),                   # A list of properties that are optional
             'provider' : [{                    # A list of providers this fetcher can use
                    'key'     : 'm',            # one-char identifier of this provider (rarely used)
                    'name'    : 'musicbrainz',  # Full name of this provider
                    'quality' : 95,             # subjective quality (0/100)
                    'speed'   : 95              # subjective speed (0/100)
             },
             {
                    ... next_provider ...
             }]
},
'lyrics': {
     ... next_fetcher ...
}
```

You can use this for example to get a list of all fetchers:

```
list(plyr.PROVIDERS.keys())
```

You get the idea. Use `itertools` and friends to get the data you want in a oneliner.

**Note:** This dictionary gets built on import. Different version of libglyr may deliver different providers.

CHAPTER 5

# Using a Database to cache items locally

Sometimes it's nice to have a local database that caches already downloaded items, so they don't get downloaded over and over. This can be achieved quite easy:

```
# Create a database in /tmp/metadata.db
db = Database('/tmp/')

# use it in queries
qry = Query(database=db,
            artist='The Cranberries',
            title='Zombie',
            get_type='lyrics')

...
```

But what happenes if a item is not found? Nothing is written to the db, and the next time it is requeried. Not always what you want. If you get an empty return from qry.commit() you could do the following:

```
def insert_dummy(db, used_query):
    dummy = Database.make_dummy()
    db.insert(used_query, dummy)
```

On the next commit you will get this item instead of an empty return, you can check for it via:

```
if returned_cache.rating is -1:
   pass  # it's a dummy
else:
   pass  # real item
```

## 5.1 Reference

CHAPTER 6

# Miscellaneous Functions

CHAPTER 7

# Examples

## 7.1 Most Simple

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import plyr

if __name__ == '__main__':
    # Create a Query, so plyr knows what you want to search
    qry = plyr.Query(get_type='lyrics', artist='Tenacious D', title='Deth Starr')

    # Now let it search all the providers
    items = qry.commit()

    # Convert lyrics (bytestring) to a proper UTF-8 text
    try:
        if len(items) > 0:
            print(str(items[0].data, 'UTF-8'))
    except UnicodeError as err:
        print('Cannot display lyrics, conversion failed:', err)
```

## 7.2 Callbacks (also with Database)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import plyr


def on_item_received(cache, query):
    cache.print_cache()
    return 'post_stop'


if __name__ == '__main__':
    # Use a local db
    db = plyr.Database('/tmp')

    # Create a Query, so plyr knows what you want to search
```

```
    qry = plyr.Query(
            get_type='lyrics',
            artist='Tenacious D',
            title='Deth Starr',
            callback=on_item_received,
            verbosity=3,
            database=db)

    # Now let it search all the providers
    # Even with callback, it will return a list of caches
    qry.commit()
```

## 7.3 Threads and Cancellation

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import plyr
import random
from time import sleep
from threading import Thread

"""
Example on how to use the cancel() function of the Query.

If a Query has been started it wil block the calling thread.
This may be bad, when you want to do something
different in the meantime. You can safely run qry.commit()
in a seperate thread and do some data-sharing. But sometimes
(e.g. on application exit) you may want to stop all running queries.
This can be done via the cancel() function - it will stop
all running downloads associated with this query.

There is no cancel_all() function.
You gonna need a pool with all Queries you're running.

Note: cancel() will not stop __imediately__,
      since some provider may do some
      stuff that is hard to interrupt,
      but at least you do not need to care about cleanup.
"""

if __name__ == '__main__':
    # Some real query, that may take a bit longer
    # Notice this nice unicode support :-)
    qry = plyr.Query(artist='', title='  !', get_type='lyrics')

    # Our worker thread
    def cancel_worker():
        sleep(random.randrange(1, 4))
        print('cancel()')
        qry.cancel()

    # Spawn the thread.
    Thread(target=cancel_worker).start()
```

```
    # Start the query, and let's see.
    print('commit() started')
    items = qry.commit()
    print('commit() finished')
    print('Number of results:', len(items))

    # Print if any item was there, there shouldn't be any.
    if len(items) > 0:
        print(str(items[0].data, 'UTF-8'))
```

## 7.4 Database II

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

import plyr


# This is called on each item found in the db
# The query is the search-query used to search this
# item, with artist, album and title filled.
# The query is useful to do delete or insert operations
# cache is the actual item, with all data filled.
def foreach_callback(query, cache):
    print(query)
    cache.print_cache()


if __name__ == '__main__':
    db = plyr.Database('/tmp')

    # Insert at least one dummy item, just in case the db is empty et
    # This will grow on each execution by one item.
    dummy_qry = plyr.Query(artist='Derp', album='Derpson', get_type='cover')
    dummy_itm = db.make_dummy()
    db.insert(dummy_qry, dummy_itm)

    # Now iterate over all items in the db
    # and exit afterwards
    db.foreach(foreach_callback)
```

**Note:** The version of this wrapper will mirror libglyr's version.

CHAPTER 8

# Indices and tables

- genindex
- modindex
- search