

---

# pygraph<sub>redis</sub>*Documentation*

***Release stable***

September 04, 2015



<b>1</b>	<b>License</b>	<b>3</b>
<b>2</b>	<b>Description</b>	<b>5</b>
<b>3</b>	<b>Dependancies</b>	<b>7</b>
<b>4</b>	<b>Write atomicity</b>	<b>9</b>
<b>5</b>	<b>Installation</b>	<b>11</b>
<b>6</b>	<b>How to use</b>	<b>13</b>
6.1	Cheat Sheet . . . . .	13
6.2	Initialization . . . . .	14
6.3	Node manipulation . . . . .	15
6.4	Node attributs manipulation . . . . .	16
6.5	Graph navigation . . . . .	17
6.6	About the redis keys . . . . .	17
6.7	About the logs . . . . .	17



Simple python library to manipulate directed graphs in redis



---

**License**

---

pygraph\_redis is released under the MIT Public License





---

### Description

---

`pygraph_redis` is a simple library to manipulate directed graphs inside a redis database.

In this library, a graph is a bunch of nodes, each node knows its predecessors and its successors. A node can store some attributes (strings or sets of strings).



---

## **Dependancies**

---

`pygraph_redis` relies on `redis` and `redis-py`.

For atomicity of transaction, it requires lua scripting support (`redis-py` `>= 2.7.0` and `redis` `>= 2.6.0`), but it provides a legacy mode, without atomicity for older `redis` and `redis-py`.



---

## Write atomicity

---

With proper versions, `pygraph_redis` provided the atomicity of transaction when adding or removing a node.



---

## Installation

---

to install:

```
$ python setup.py install
```





---

## How to use

---

First you need a redis database, it's up to you to install it.

The library itself is quite simple:

### 6.1 Cheat Sheet

```
#           initialization
#   arg1      |   arg2      |   arg3
#-----
# redis connexion | graph_name |   logger
#   redis obj    | unicode    | logger obj

mygraph1 = Directed_graph(r_server, u'mygraph1', logger)

#optional args:
#   arg4      |   arg5
#-----
# separator | has_root
# unicode    | bool

mygraph1 = Directed_graph(r_server,
    u'mygraph1', logger, u'mysep', True)
)
```

```
#           create or add elements to a node
#   arg1      |   arg2      |   arg3      |   arg4
#-----
# node name | successors | predecessors |   attributs
# unicode   | unicode list | unicode list |   dictionnary of unicode
#           |           |           | or set of unicode (key: unicode)

mygraph1.write_on_node(u'm1',
    [u's2'],
    [u'p1'],
    {u'a3': set([u'69']), u'a2': u'42'})
)
```

```
#           delete elements from a node
#   arg1      |   arg2      |   arg3      |   arg4
#-----
# node name | successors | predecessors | attributs names
```

```
# unicode / unicode list / unicode list / list of unicode
```

```
mygraph1.write_off_node(u'm1', [u's2'], [u'p1'], [u'attr3', u'attr2'])
```

```
# delete a node
```

```
#     arg1
```

```
#-----
```

```
# node name
```

```
#     unicode
```

```
mygraph1.remove_node(u'm1')
```

```
# get attributes list
```

```
#     arg1
```

```
#-----
```

```
# node name
```

```
#     unicode
```

```
mygraph1.get_attributes_list(u'm1')
```

```
# get an attribute
```

```
#     arg1 /     arg2
```

```
#-----/-----
```

```
# node name / attribute name
```

```
#     unicode /     unicode
```

```
mygraph1.get_attribute(u'm1', u'a2')
```

```
# get an attribute length
```

```
#     arg1 /     arg2
```

```
#-----/-----
```

```
# node name / attribute name
```

```
#     unicode /     unicode
```

```
mygraph1.get_attribute_len(u'm1', u'a2')
```

```
# get successors
```

```
#     arg1
```

```
#-----
```

```
# node name
```

```
#     unicode
```

```
mygraph1.get_successors(u'm1')
```

```
# get predecessors
```

```
#     arg1
```

```
#-----
```

```
# node name
```

```
#     unicode
```

```
mygraph1.get_predecessors(u'm1')
```

## 6.2 Initialization

Create an instance of “Directed\_graph”:

```
#importing directed_graph
from pygraph_redis.directed_graph import Directed_graph
import redis

#creating a basic logger
import logging
logging.basicConfig(format = u'%(message)s')
logger = logging.getLogger(u'redis')
logger.parent.setLevel(logging.DEBUG)

#creating the redis connexion
r_server = redis.Redis("localhost")

#creating the graph object
mygraph1 = Directed_graph(r_server, u'mygraph1', logger)

#creating the graph object with a different separator
mygraph2 = Directed_graph(r_server, u'mygraph2', logger, separator = u'mysep')

#creating the graph object with a "root" (improper name, I know)
mygraph2 = Directed_graph(r_server, u'mygraph2', logger, has_root = True)
# "has_root = True" ensures that every node has a predecessor
# if enabled, a node has at least root as a predecessor,
# but if it has any other predecessor it doesn't have root as predecessor
```

## 6.3 Node manipulation

Node creation:

```
#add node 'm1' to 'mygraph1' with:
#successors: 's1' and 's2'
#predecessors: 'p1' and 'p2'
#attributs:
# * 'attr1': set([u'51',u'69'])
# * 'attr2': '42'

mygraph1.write_on_node(u'm1',
    [u's1', u's2'],
    [u'p1', u'p2'],
    {u'attr1': set([u'51', u'69']), u'attr2': u'42'})
)
```

About *successors* and *predecessors*, if node was already declared as a predecessor of one of its successors, it's not necessary to add this successor in node successors set. Same with *predecessors*.

example:

```
mygraph1.write_on_node(u'pred',
    [u'succ'],
    [],
    {})
)
```

```
mygraph1.write_on_node(u'succ',
    [],
    [],
```

```
{ }  
)
```

Gives the same result that:

```
mygraph1.write_on_node(u'pred',  
    [u'succ'],  
    [],  
    {}  
)
```

```
mygraph1.write_on_node(u'succ',  
    [],  
    [u'pred'],  
    {}  
)
```

Node edition:

```
#add new elements or edit existing elements of a node  
#it's exactly the same function as before  
mygraph1.write_on_node(u'm1',  
    [u's4'],  
    [],  
    {u'attr3': set([u'16', u'32', u'64']), u'attr2': u'5150'})  
)  
  
#remove some elements of a node (successors, predecessors, attributes)  
mygraph1.write_off_node(u"m1", [u"s1"], [u"p2"], [u'attr2'])  
  
#completely delete a node  
mygraph1.remove_node(u'm1')
```

## 6.4 Node attributes manipulation

To manipulate the attributes of a node:

```
#create the node 'm2'  
mygraph1.write_on_node(u'm2',  
    [u's1', u's2'],  
    [u'p1', u'p2'],  
    {u'attr1': set([u'51', u'69']), u'attr2': u'42'})  
)  
  
#get the set of attribute names  
set_of_attributes = mygraph1.get_attributes_list(u'm2')  
print set_of_attributes  
  
#get a specific attribute  
attr2 = mygraph1.get_attribute(u'm2', u'attr2')  
print attr2  
  
#get a specific attribute length  
# 1 if it's a string  
# cardinal of set if it's a set  
# 0 if attribute doesn't exists
```

```
attr2 = mygraph1.get_attribut_len(u'm2', u'attr2')
print attr2
```

## 6.5 Graph navigation

To navigate inside the graph, you have two functions:

```
#get the predecessors of 'm2'
predecessors = mygraph1.get_predecessors(u'm2')
print predecessors

#get the successors of 'm2'
successors = mygraph1.get_successors(u'm2')
```

if you have the *has\_root* flag enable:

```
#get the "root" name
root = mygraph1.get_root_name()

print root

#get the successors of 'root'
successors = mygraph1.get_successors(root)
print successors
```

## 6.6 About the redis keys

Redis key format:

```
<graph name><sep><node_name><sep><variable_name>[<sep><other>]*

<graph name>: name of the graph
<sep>: the key fields separator
      (this string should not be in node_name or variable_name,
       otherwise, there is a redis key collision possibility)
<node_name>: name of the node
<variable_name>: name of the variable
[<sep><other>]: optional extension
```

To avoid key collision, you must carefully choose the key separator, it must not be included in any node name or node attribut name (possible redis key collision).

## 6.7 About the logs

This library provides a lot of logs, mainly debug, some info (ex: legacy modes), some warning (ex: possible key collision)