# coopy Documentation

*Release 0.4b*

Felipe Cruz

November 27, 2013

# Contents

**coopy** is a simple, transparent, non-intrusive persistence library for python language. It's released under BSD License

- **Simple** - you don't have to learn an API. You can use it with just one line of code.
- **Transparent** - you don't need to call any API functions, just your Object methods.
- **Non-Intrusive** - no inheritance, no interface.. only pure-python-business code.

It is based on the techniques of system snapshotting and transaction journalling. In the prevalent model, the object data is kept in memory in native object format, rather than being marshalled to an RDBMS or other data storage system. A snapshot of data is regularly saved to disk, and in addition to this, all changes are serialised and a log of transactions is also stored on disk.

http://en.wikipedia.org/wiki/Object_prevalence

CHAPTER 1

# Using

Simple, transparent and non-intrusive. Note that `Todo` could be any class that you want to persist state across method calls that modifies it's internal state:

```
from coopy.base import init_persistent_system

class Todo(object):
    def __init__(self):
        self.tasks = []

    def add_task(self, name, description):
        task = dict(name=name, description=description)
        self.tasks.append(task)

persistent_todo_list = init_persistent_system(Todo())
persistent_todo_list.add_task("Some Task Name", "A Task Description")
```

Check out how coopy works with this little *1 minute coopy tutorial* and then...

It's very important to know how coopy works, to use it. Check out *coopy basics*

CHAPTER 2

# Restrictions

This should not affect end-user code. To get datetime or date objects you need to get from an internal clock. Check this page *How to Use Clock*

# Status

coopy is compatible with py2.6, py2.7, py3.2, py3.3 and pypy.

CHAPTER 4

# contribute

**coopy** code is hosted on github at: http://github.com/felipecruz/coopy

Found a bug? http://github.com/felipecruz/coopy

CHAPTER 5

# contents

## 5.1 Installation

```
$ pip install coopy
```

### 5.1.1 Development Version

You can always check our bleeding-edge development version:

```
$ git clone http://github.com/felipecruz/coopy.git
```

and then:

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

## 5.2 1 minute coopy tutorial

**coopy** enforces you to implement code in the object-oriented way. Imagine a wiki system:

```
class WikiPage():
    def __init__(self, id, content):
        self.id = id
        self.content = content
        self.history = []
        self.last_modify = datetime.datetime.now()

class Wiki():
    def __init__(self):
        self.pages = {}
    def create_page(self, page_id, content):
        page = None
        if page_id in self.pages:
            page = self.pages[page_id]
        if not page:
            page = WikiPage(page_id, content)
            self.pages[page_id] = page
        return page
```

It's very easy to implement a wiki system thinking only on it's objects. Let's move forward:

```
from coopy import init_system
wiki = init_system(Wiki(), "/path/to/somedir")
wiki.create_page('My First Page', 'My First Page Content')
```

That's all you need to use coopy. If you stop your program and run again:

```
from coopy import init_system
wiki = init_system(Wiki(), "/path/to/somedir")
page = wiki.pages['My First Page']
```

If you want to know how coopy works, check out *coopy basics*

## 5.3 Using coopy

There are many different ways to use **coopy**. Let me show you some:

```
class WikiPage():
    def __init__(self, id, content):
        self.id = id
        self.content = content
        self.history = []
        self.last_modify = datetime.datetime.now()

class Wiki():
    def __init__(self):
        self.pages = {}
    def create_page(self, page_id, content):
        page = None
        if page_id in self.pages:
            page = self.pages[page_id]
        if not page:
            page = WikiPage(page_id, content)
            self.pages[page_id] = page
        return page

wiki = init_system(Wiki)
```

or:

```
wiki = init_system(Wiki())
```

or:

```
wiki = init_system(Wiki(),'/path/to/log/files')
```

or setup a Master node:

```
init_system(Wiki, master=True)
```

or setup a Slave node:

```
init_system(Wiki, replication=True)
```

or check all arguments:

```
def init_system(obj, basedir=None, snapshot_time=0, master=False, replication=False, port=5466, host=
```

## 5.4 coopy basics

**coopy** returns to you a proxy to your object. Everytime you call some method on this proxy, coopy will log to disk this operation, so it can be re-executed later on a restore process. This behaviour assures that you object will have their state persisted.

So far, you know that you are manipulating a proxy object and when you call methods on this object, this invocation will be written to disk. We call **log file** the files that contains all operations executed on your object. This log files are created on what we call **basedir**. You can specify basedir or coopy will lowercase your object class name and create a directory with this name to store all log files.

**coopy logger** is responsible to receive this methods invocations, create **Action** objects and serialize to disk. It automatically handles file rotations, like python logging RotateFileHandler, in order to keep log files not too big.

As your application is running, your log file number will be increasing and restore process can start to run slowly, becase it'll open many log files. To avoid that you can take **snapshots** from your object. **Snapshot file** is a copy of your objects in memory serialized trhough the disk. As you take a snapshot, all log files older than this snapshots can be deleted if you want. Take snapshots will also speed up the restore process, because is much more fast open 1 file and deserialize to memory than open 10 files to execute each action inside of them.

Now, you know everything about how information are stored. Let's see how this information are restored.

**Restore process** is what coopy do to restore your object state. It checks for **log files** and **snapshot files** on your **basedir** to look to the last snapshot taken and all log files created after. It'll deserialize this **snapshot file** and then open all log files to re-execute all **Actions** that were executed after the snapshot was created. This will assure that your object will have the same state as your object had once in the past when your program was terminated or maybe killed.

The bascis of **coopy** is covered here

- You are manipulating a proxy object that delegates memory execution to your **domain** object
- Once you call a method on proxy, this call turns into a **Action** object and then serialized to disk.
- **Log files** contain **Action** objects to be re-executed
- You can take **Snapshots** of your object to increase your **restore process** and have a small number of files on your **basedir**
- Everytime you use **coopy** it'll look to your **basedir** and restore your object state with the files there

All this is done using python cPickle module.

## 5.5 How to Use Clock

### 5.5.1 Date problem

**coopy** is based on re-execute actions performed in the past. When you call datetime.now() inside an 'business' method, when your actions are executed in restore process, datetime.now() will be executed again. This behaviour will produce unexpected results.

### 5.5.2 Why use Clock?

Clock uses **coopy** timestamp. When you execute a 'business' method, coopy takes the current timestamp and persist inside action object. Clock object has his timestamp updated with action timestamp so in a restore process, Clock will have the original timestamp, and not the timestamp from the re-execution process.

Wrong code:

```
def create_page(self, wikipage):
    page = None
    wikipage.last_modify = datetime.now()
    ....
```

Right code:

```
from coopy import clock
def create_page(self, wikipage):
    page = None
    wikipage.last_modify = self._clock.now()
    ....
```

### 5.5.3 Clock API

#### Clock-aware code validation

coopy has a validation mechanism that will not accept obvious code errors such as calling `datetime.now()` inside a system method.

#### API

Take note that a `_clock` attribute is injected on your system instance and the API is always called via `self._clock`.

For Clock instances

`clock.`**`now`**`()`
**`Return datetime.now()`**

**Return type** datetime

`clock.`**`utcnow`**`()`
**`Return datetime.now()`**

**Return type** datetime

`clock.`**`today`**`()`
**`Return date.today()`**

**Return type** date

## 5.6 Method Decorators

### 5.6.1 @readonly

This decorator means thar your method will not mofidy business objects. Like a get method from a Wiki class. Therefore, this method will not generate a log entry at coopy actions log:

```python
from coopy.decorators import readonly
@readonly
def get_page(self, id):
    if id in self.pages:
        return self.pages[id]
    else:
        return None
```

### 5.6.2 @unlocked

How coopy assures thread-safety? By synchronizing method invocations using a reetrant lock.

This decorator provides a means of leaving the thread safety in your hands via the @unlocked decorator. Using this decorator, you should implement concurrency mechanism by yourself.

### 5.6.3 @abort_exception

Default behaviour is to log on disk, even if your code raises an exception.

If your 'business' method raises an exception and your method is decoreted by @abort_exception, this execution will not be logged at disk. This means that during restore process, this invocation that raised an exception will not be re-executed:

```python
from coopy.decorators import abort_exception
@abort_exception
def create_page(self, wikipage):
    page = None
    wikipage.last_modify = coopy.clock.now()
    if wikipage.id in self.pages:
        page = self.pages[wikipage.id]
    if not page:
        self.pages[wikipage.id] = wikipage
        raise Exception('Exemple error')
    else:
        self.update_page(wikipage.id, wikipage.content)
```

**Restore process will not execute this method because it wasn't logged at disk.**

## 5.7 Snapshots

### 5.7.1 Motivation

If your domain is really active and generates tons of logs, we suggest you to take snapshots from your domain periodically. A snapshot allows you to delete your logs older then it's timestamp and make the restore process faster. Today, while taking a snapshot the domain is *locked*. It's fairly common setup a local slave just for taking snapshots.

### 5.7.2 Example

Example:

```
from coopy.base import init_persistent_system

persistent_todo_list = init_persistent_system(Todo())
persistent_todo_list.add_task("Some Task Name", "A Task Description")

# Take snapshot
persistent_todo_list.take_snapshot()
```

### 5.7.3 API

For domain instances

`domain.`**`take_snapshot`**`()`
**`Takes the domain snapshot.`**

## 5.8 Utilitary API

There are some utilitary methods to help you.

Given:

```
wiki = init_system(Wiki)
```

**`basedir_abspath`**`()`
    Return a list with all basedirs absolute paths

### 5.8.1 Tests utils

If your domain uses the *How to Use Clock* feature, you'll likely to face errors while testing your pure domain since the *_clock* is injected by coopy.

There are 2 ways of handle this: Enable a regular clock on your domain, for testing or mock your clock to return the same date.

`TestSystemMixin.`**`mock_clock`**`(`*domain*, *mocked_date*`)`
    This method will inject a clock that always return *mocked_date*

`TestSystemMixin.`**`enable_clock`**`(`*domain*`)`
    This method will inject a regular coopy clock on your domain instance

## 5.9 Client-Server

When you want to detach client from server, you can use coopy + Pyro (or xmlrpclib) in order to have a client and a server (running coopy).

This is useful when you want to have only one machine dedicated to have it's ram memory filled with python objects.

Note, that this example uses Pyro.core.ObjBase instead of Pyro.core.SynchronizedObjBase, because by default, coopy proxy (wiki object) is already thread-safe unless you decorate your business methods with @unlocked decorator.

Server Code:

```
#coopy code
wiki = coopy.init_system(Wiki(), "pyro")

#pyro code
obj = Pyro.core.ObjBase()
obj.delegateTo(wiki)
Pyro.core.initServer()
daemon=Pyro.core.Daemon()
uri=daemon.connect(obj,"wiki")
daemon.requestLoop()
```

Client code:

```
#pyro code
wiki = Pyro.core.getProxyForURI("PYRO://127.0.0.1:7766/whatever")
```

## 5.10 Master/Slave replication

**coopy** comes with master/slave replication mechanism.

Basically:

- Master instance are read/write
- Slaves are read only
- Slaves can only execute @readonly methods.

Another detail, is that you can set a password on master. This password provides basich auth to slaves connects to a master instance.

When slaves connects to master and passes authentication process, it will receive all data to synchronize with master state. Commands executed further on master will be replicated to slave node.

Slaves are useful to take snapshots without needing master to be locked as well to provide load balancing for reading.

Snipets to run master and slave instances

Master instance:

```
init_system(Wiki, master=True)
```

Slave instance, default host and default port:

```
init_system(Wiki, replication=True)
```

## 5.11 Tests

First time:

```
pip install -r requirements.txt
```

To actually run the tests:

```
make test
```

## 5.12 Coverage Report

First time:

```
pip install -r requirements.txt
```

And then:

```
make coverage
```

Coverage report:

```
$ py.test --cov coopy

Name                             Stmts   Miss  Cover
----------------------------------------------------
coopy/__init__                       0      0   100%
coopy/base                         135     17    87%
coopy/decorators                     9      0   100%
coopy/error                          3      0   100%
coopy/fileutils                    125      5    96%
coopy/foundation                    71      8    89%
coopy/journal                       30      2    93%
coopy/network/__init__               1      0   100%
coopy/network/default_select       192     56    71%
coopy/network/linux_epoll            0      0   100%
coopy/network/network               42     10    76%
coopy/network/osx_kqueue             0      0   100%
coopy/restore                       42      6    86%
coopy/snapshot                      45      3    93%
coopy/utils                          9      0   100%
coopy/validation                    45      1    98%
----------------------------------------------------
TOTAL                              749    108    86%
```

## 5.13 Roadmap

- First stable release

## 5.14 Changelog

Nothing so far.

## 5.15 TODO

- Finish Documentation