tasklib Documentation

Release 1.0.0

Rob Golding

March 14, 2016

Contents

tasklib is a Python library for interacting with taskwarrior databases, using a queryset API similar to that of Django's ORM.

Supports Python 2.6, 2.7, 3.2, 3.3 and 3.4 with taskwarrior 2.1.x and above. Older versions of taskwarrior are untested and may not work.

CHAPTER 1

Requirements

- taskwarrior v2.1.x or above, although newest minor release is recommended.

CHAPTER 2

Installation

Install via pip (recommended):

```
pip install tasklib
```

Or clone from github:

```
git clone https://github.com/robgolding63/tasklib.git
cd tasklib
python setup.py install
```

CHAPTER 3

Initialization

Optionally initialize the `TaskWarrior` instance with `data_location` (the database directory). If it doesn't already exist, this will be created automatically unless `create=False`.

The default location is the same as taskwarrior's:

```
>>> tw = TaskWarrior(data_location='~/.task', create=True)
```

The `TaskWarrior` instance will also use your .taskrc configuration (so that it recognizes the same UDAs as your task binary, uses the same configuration, etc.). To override the location of the .taskrc, use `taskrc_location=~/some/different/path`.

CHAPTER 4

Creating Tasks

To create a task, simply create a new `Task` object:

```
>>> new_task = Task(tw, description="throw out the trash")
```

This task is not yet saved to TaskWarrior (same as in Django), not until you call `.save()` method:

```
>>> new_task.save()
```

You can set any attribute as a keyword argument to the Task object:

```
>>> complex_task = Task(tw, description="finally fix the shower", due=datetime(2015,2,14,8,0,0), pri
```

or by setting the attributes one by one:

```
>>> complex_task = Task(tw)
>>> complex_task['description'] = "finally fix the shower"
>>> complex_task['due'] = datetime(2015,2,14,8,0,0)
>>> complex_task['priority'] = 'H'
```

CHAPTER 5

Modifying Task

To modify a created or retrieved `Task` object, use dictionary-like access:

```
>>> homework = tw.tasks.get(tags=['chores'])
>>> homework['project'] = 'Home'
```

The change is not propagated to the TaskWarrior until you run the `save()` method:

```
>>> homework.save()
```

Attributes, which map to native Python objects are converted. See Task Attributes section.

CHAPTER 6

Task Attributes

Attributes of task objects are accessible through indices, like so:

```
>>> task = tw.tasks.pending().get(tags__contain='work')  # There is only one pending task with 'work
>>> task['description']
'Upgrade Ubuntu Server'
>>> task['id']
15
>>> task['due']
datetime.datetime(2015, 2, 5, 0, 0, tzinfo=<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>)
>>> task['tags']
['work', 'servers']
```

The following fields are deserialized into Python objects:

- `due`, `wait`, `scheduled`, `until`, `entry`: deserialized to a `datetime` object
- `annotations`: deserialized to a list of `TaskAnnotation` objects
- `tags`: deserialized to a list of strings
- `depends`: deserialized to a set of `Task` objects

Attributes should be set using the correct Python representation, which will be serialized into the correct format when the task is saved.

CHAPTER 7

Task properties

Tasklib defines several properties upon `Task` object, for convenience:

```
>>> t.save()
>>> t.saved
True
>>> t.pending
True
>>> t.active
False
>>> t.start()
>>> t.active
True
>>> t.done()
>>> t.completed
True
>>> t.pending
False
>>> t.delete()
>>> t.deleted
True
```

CHAPTER 8

Operations on Tasks

After modifying one or more attributes, simple call `save()` to write those changes to the database:

```
>>> task = tw.tasks.pending().get(tags__contain='work')
>>> task['due'] = datetime(year=2014, month=1, day=5)
>>> task.save()
```

To mark a task as complete, use `done()`:

```
>>> task = tw.tasks.pending().get(tags__contain='work')
>>> task.done()
>>> len(tw.tasks.pending().filter(tags__contain='work'))
0
```

To delete a task, use `delete()`:

```
>>> task = tw.tasks.get(description="task added by mistake")
>>> task.delete()
```

To update a task object with values from TaskWarrior database, use `refresh()`. Example:

```
>>> task = Task(tw, description="learn to cook")
>>> task.save()
>>> task['id']
5
>>> task['tags']
[]
```

Now, suppose the we modify the task using the TaskWarrior interface in another terminal:

```
$ task 5 modify +someday
Task 5 modified.
```

Switching back to the open python process:

```
>>> task['tags']
[]
>>> task.refresh()
>>> task['tags']
['someday']
```

Tasks can also be started and stopped. Use `start()` and `stop()` respectively:

```
>>> task.start()
>>> task['start']
datetime.datetime(2015, 7, 16, 18, 48, 28, tzinfo=<DstTzInfo 'Europe/Prague' CEST+2:00:00 DST>)
```

```
>>> task.stop()
>>> task['start']
>>> task.done()
>>> task['end']
datetime.datetime(2015, 7, 16, 18, 49, 2, tzinfo=<DstTzInfo 'Europe/Prague' CEST+2:00:00 DST>)
```

CHAPTER 9

Retrieving Tasks

`tw.tasks` is a `TaskQuerySet` object which emulates the Django QuerySet API. To get all tasks (including completed ones):

```
>>> tw.tasks.all()
['First task', 'Completed task', 'Deleted task', ...]
```

CHAPTER 10

Filtering

Filter tasks using the same familiar syntax:

```
>>> tw.tasks.filter(status='pending', tags__contains=['work'])
['Upgrade Ubuntu Server']
```

Filter arguments are passed to the `task` command (___ is replaced by a period) so the above example is equivalent to the following command:

```
$ task status:pending tags.contain=work
```

Tasks can also be filtered using raw commands, like so:

```
>>> tw.tasks.filter('status:pending +work')
['Upgrade Ubuntu Server']
```

Although this practice is discouraged, as by using raw commands you may lose some of the portablility of your commands over different TaskWarrior versions.

However, you can mix raw commands with keyword filters, as in the given example:

```
>>> tw.tasks.filter('+BLOCKING', project='Home')  # Gets all blocking tasks in project Home
['Fix the toilette']
```

This can be a neat way how to use syntax not yet supported by tasklib. The above is excellent example, since virtual tags do not work the same way as the ordinary ones, that is:

```
>>> tw.tasks.filter(tags=['BLOCKING'])
>>> []
```

will not work.

There are built-in functions for retrieving pending & completed tasks:

```
>>> tw.tasks.pending().filter(tags__contain='work')
['Upgrade Ubuntu Server']
>>> len(tw.tasks.completed())
227
```

Use `get()` to return the only task in a `TaskQuerySet`, or raise an exception:

```
>>> tw.tasks.get(tags__contain='work')['status']
'pending'
>>> tw.tasks.get(status='completed', tags__contains='work')  # Status of only task with the work tag
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "tasklib/task.py", line 224, in get
    'Lookup parameters were {0}'.format(kwargs))
tasklib.task.DoesNotExist: Task matching query does not exist. Lookup parameters were {'status': 'co
>>> tw.tasks.get(status='pending')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "tasklib/task.py", line 227, in get
    'Lookup parameters were {1}'.format(num, kwargs))
ValueError: get() returned more than one Task -- it returned 23! Lookup parameters were {'status': 'p
```

Additionally, since filters return `TaskQuerySets` you can stack filters on top of each other:

```
>>> home_tasks = tw.tasks.filter(project='Wife')
>>> home_tasks.filter(due__before=datetime(2015,2,14,14,14,14))  # What I have to do until Valentine
['Prepare surprise birthday party']
```

CHAPTER 11

Equality of Task objects

Two Tasks are considered equal if they have the same UUIDs:

```
>>> task1 = Task(tw, description="Pet the dog")
>>> task1.save()
>>> task2 = tw.tasks.get(description="Pet the dog")
>>> task1 == task2
True
```

If you compare the two unsaved tasks, they are considered equal only if it's the same Python object:

```
>>> task1 = Task(tw, description="Pet the cat")
>>> task2 = Task(tw, description="Pet the cat")
>>> task1 == task2
False
>>> task3 = task1
>>> task3 == task1
True
```

CHAPTER 12

Accessing original values

To access the saved state of the Task, use dict-like access using the `original` attribute:

```
>>> t = Task(tw, description="tidy up")
>>> t.save()
>>> t['description'] = "tidy up the kitchen and bathroom"
>>> t['description']
"tidy up the kitchen and bathroom"
>>> t.original['description']
"tidy up"
```

When you save the task, original values are refreshed to reflect the saved state of the task:

```
>>> t.save()
>>> t.original['description']
"tidy up the kitchen and bathroom"
```

CHAPTER 13

Dealing with dates and time

Any timestamp-like attributes of the tasks are converted to timezone-aware datetime objects. To achieve this, Tasklib leverages `pytz` Python module, which brings the Olsen timezone databaze to Python.

This shields you from annoying details of Daylight Saving Time shifts or conversion between different timezones. For example, to list all the tasks which are due midnight if you're currently in Berlin:

```
>>> myzone = pytz.timezone('Europe/Berlin')
>>> midnight = myzone.localize(datetime(2015,2,2,0,0,0))
>>> tw.tasks.filter(due__before=midnight)
```

However, this is still a little bit tedious. That's why TaskWarrior object is capable of automatic timezone detection, using the `tzlocal` Python module. If your system timezone is set to 'Europe/Berlin', following example will work the same way as the previous one:

```
>>> tw.tasks.filter(due__before=datetime(2015,2,2,0,0,0))
```

You can also use simple dates when filtering:

```
>>> tw.tasks.filter(due__before=date(2015,2,2))
```

In such case, a 00:00:00 is used as the time component.

Of course, you can use datetime naive objects when initializing Task object or assigning values to datetime atrributes:

```
>>> t = Task(tw, description="Buy new shoes", due=date(2015,2,5))
>>> t['due']
datetime.datetime(2015, 2, 5, 0, 0, tzinfo=<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>)
>>> t['due'] = date(2015,2,6,15,15,15)
>>> t['due']
datetime.datetime(2015, 2, 6, 15, 15, 15, tzinfo=<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>)
```

However, since timezone-aware and timezone-naive datetimes are not comparable in Python, this can cause some unexpected behaviour:

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> t = Task(tw, description="take out the trash now")
>>> t['due'] = now
>>> now
datetime.datetime(2015, 2, 1, 19, 44, 4, 770001)
>>> t['due']
datetime.datetime(2015, 2, 1, 19, 44, 4, 770001, tzinfo=<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>)
>>> t['due'] == now
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
  TypeError: can't compare offset-naive and offset-aware datetimes
```

If you want to compare datetime aware value with datetime naive value, you need to localize the naive value first:

```
>>> from datetime import datetime
>>> from tasklib.task import local_zone
>>> now = local_zone.localize(datetime.now())
>>> t['due'] = now
>>> now
datetime.datetime(2015, 2, 1, 19, 44, 4, 770001, tzinfo=<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>)
>>> t['due'] == now
True
```

Also, note that it does not matter whether the timezone aware datetime objects are set in the same timezone:

```
>>> import pytz
>>> t['due']
datetime.datetime(2015, 2, 1, 19, 44, 4, 770001, tzinfo=<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>)
>>> now.astimezone(pytz.utc)
datetime.datetime(2015, 2, 1, 18, 44, 4, 770001, tzinfo=<UTC>)
>>> t['due'] == now.astimezone(pytz.utc)
True
```

Note: Following behaviour is available only for TaskWarrior >= 2.4.0.

There is a third approach to setting up date time values, which leverages the 'task calc' command. You can simply set any datetime attribute to any string that contains an acceptable TaskWarrior-formatted time expression:

```
$ task calc now + 1d
2015-07-17T21:17:54
```

This syntax can be leveraged in the python interpreter as follows:

```
>>> t['due'] = "now + 1d"
>>> t['due']
datetime.datetime(2015, 7, 17, 21, 19, 31, tzinfo=<DstTzInfo 'Europe/Berlin' CEST+2:00:00 DST>)
```

It can be easily seen that the string with TaskWarrior-formatted time expression is automatically converted to native datetime in the local time zone.

For the list of acceptable formats and keywords, please consult:

- http://taskwarrior.org/docs/dates.html
- http://taskwarrior.org/docs/named_dates.html

However, as each such assigment involves call to 'task calc' for conversion, it might cause some performance issues when assigning strings to datetime attributes repeatedly, in a automated manner.

CHAPTER 14

Working with annotations

Annotations of the tasks are represented in tasklib by `TaskAnnotation` objects. These are much like `Task` objects, albeit very simplified.

```
>>> annotated_task = tw.tasks.get(description='Annotated task')
>>> annotated_task['annotations']
[Yeah, I am annotated!]
```

Annotations have only defined `entry` and `description` values:

```
>>> annotation = annotated_task['annotations'][0]
>>> annotation['entry']
datetime.datetime(2015, 1, 3, 21, 13, 55, tzinfo=<DstTzInfo 'Europe/Berlin' CET+1:00:00 STD>)
>>> annotation['description']
u'Yeah, I am annotated!'
```

To add a annotation to a Task, use `add_annotation()`:

```
>>> task = Task(tw, description="new task")
>>> task.add_annotation("we can annotate any task")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    File "build/bdist.linux-x86_64/egg/tasklib/task.py", line 355, in add_annotation
tasklib.task.NotSaved: Task needs to be saved to add annotation
```

However, Task needs to be saved before you can add a annotation to it:

```
>>> task.save()
>>> task.add_annotation("we can annotate saved tasks")
>>> task['annotations']
[we can annotate saved tasks]
```

To remove the annotation, pass its description to `remove_annotation()` method:

```
>>> task.remove_annotation("we can annotate saved tasks")
```

Alternatively, you can pass the `TaskAnnotation` object itself:

```
>>> task.remove_annotation(task['annotations'][0])
```

CHAPTER 15

Running custom commands

To run a custom commands, use `execute_command()` method of `TaskWarrior` object:

```
>>> tw = TaskWarrior()
>>> tw.execute_command(['log', 'Finish high school.'])
[u'Logged task.']
```

You can use `config_override` keyword argument to specify a dictionary of configuration overrides:

```
>>> tw.execute_command(['3', 'done'], config_override={'gc': 'off'}) # Will mark 3 as completed and
```

Additionally, you can use `return_all=True` flag, which returns `(stdout, sterr, return_code)` triplet, and `allow_failure=False`, which will prevent tasklib from raising an exception if the task binary returned non-zero return code:

```
>>> tw.execute_command(['invalidcommand'], allow_failure=False, return_all=True)
([u''],
 [u'Using alternate .taskrc file /home/tbabej/.taskrc',
  u"[task next rc:/home/tbabej/.taskrc rc.recurrence.confirmation=no rc.json.array=off rc.confirmatio
  u'Configuration override rc.recurrence.confirmation:no',
  u'Configuration override rc.json.array:off',
  u'Configuration override rc.confirmation:no',
  u'Configuration override rc.bulk:0',
  u'Configuration override rc.dependency.confirmation:no',
  u'No matches.',
  u'There are local changes.  Sync required.'],
 1)
```

CHAPTER 16

Setting custom configuration values

By default, TaskWarrior uses configuration values stored in your .taskrc. To see what configuration value overrides are passed to each executed task command, have a peek into `overrides` attribute of `TaskWarrior` object:

```
>>> tw.overrides
{'confirmation': 'no', 'data.location': '/home/tbabej/.task'}
```

To pass your own configuration overrides, you just need to update this dictionary:

```
>>> tw.overrides.update({'hooks': 'off'})  # tasklib will not trigger hooks
```

CHAPTER 17

Creating hook scripts

From version 2.4.0, TaskWarrior has support for hook scripts. Tasklib provides some very useful helpers to write those. With tasklib, writing these becomes a breeze:

```
#!/usr/bin/python

from tasklib.task import Task
task = Task.from_input()
# ... <custom logic>
print task.export_data()
```

For example, plugin which would assign the priority "H" to any task containing three exclamation marks in the description, would go like this:

```
#!/usr/bin/python

from tasklib.task import Task
task = Task.from_input()

if "!!!" in task['description']:
    task['priority'] = "H"

print task.export_data()
```

Tasklib can automatically detect whether it's running in the `on-modify` event, which provides more input than `on-add` event and reads the data accordingly.

This means the example above works both for `on-add` and `on-modify` events!

Consenquently, you can create just one hook file for both `on-add` and `on-modify` events, and you just need to create a symlink for the other one. This removes the need for maintaining two copies of the same code base and/or boilerplate code.

In `on-modify` events, tasklib loads both the original version and the modified version of the task to the returned `Task` object. To access the original data (in read-only manner), use `original` dict-like attribute:

```
>>> t = Task.from_input()
>>> t['description']
"Modified description"
>>> t.original['description']
"Original description"
```

CHAPTER 18

Working with UDAs

Since TaskWarrior does read your .taskrc, you need not to define any UDAs in the TaskWarrior's config dictionary, as described above. Suppose we have a estimate UDA in the .taskrc:

```
uda.estimate.type = numeric
```

We can simply filter and create tasks using the estimate UDA out of the box:

```
>>> tw = TaskWarrior()
>>> task = Task(tw, description="Long task", estimate=1000)
>>> task.save()
>>> task['id']
1
```

This is saved as UDA in the TaskWarrior:

```
$ task 1 export
{"id":1,"description":"Long task","estimate":1000, ...}
```

We can also speficy UDAs as arguments in the TaskFilter:

```
>>> tw.tasks.filter(estimate=1000)
Long task
```

CHAPTER 19

Syncing

If you have configurated the needed config variables in your .taskrc, syncing is as easy as:

```
>>> tw = TaskWarrior()
>>> tw.execute_command(['sync'])
```

If you want to use non-standard server/credentials, you'll need to provide configuration overrides to the `TaskWarrior` instance. Update the `config` dictionary with the values you desire to override, and then we can run the sync command using the `execute_command()` method:

```
>>> tw = TaskWarrior()
>>> sync_config = {
...     'taskd.certificate': '/home/tbabej/.task/tbabej.cert.pem',
...     'taskd.credentials': 'Public/tbabej/34af54de-3cb2-4d3d-82be-33ddb8fd3e66',
...     'taskd.server': 'task.server.com:53589',
...     'taskd.ca': '/home/tbabej/.task/ca.cert.pem',
...     'taskd.trust': 'ignore hostname'}
>>> tw.config.update(sync_config)
>>> tw.execute_command(['sync'])
```