Atomos Documentation

Release 0.3.1

Max Countryman

July 05, 2015

Atomic primitives for Python.

Atomos is a library of atomic primitives, inspired by Java's java.util.concurrent.atomic. It provides atomic types for bools, ints, longs, and floats as well as a generalized object wrapper. In addition, it introduces atoms, a concept Clojure programmers will be familiar with.

CHAPTER 1

Installation

Atomos is available via PyPI.

```
$ pip install atomos
```

CHAPTER 2

Usage

A short tutorial is presented in the README.

CHAPTER 3

API

class atomos.atom.**Atom**(*state*)

Atom object type.

Atoms store mutable state and provide thread-safe methods for retrieving and altering it. This is useful in multi-threaded contexts or any time an application makes use of shared mutable state. By using an atom, it is possible to ensure that the read and write operations are always consistent.

For example, if an application uses a dictionary to store state, using an atom will guarantee that the dictionary is never in an inconsistent state as it is being updated:

```
>>> state = Atom({'active_conns': 0, 'clients': set([])})
>>> def new_client(cur_state, client):
...     cur_state['clients'].add(client)
...     cur_state['active_conns'] += 1
...     return cur_state
>>> state.swap(new_client, 'foo')
```

In the above example we use an atom to store state about connections. Our mutation function, *new_client* is a function which takes the existing state contained by the atom and a new client. Any part of our program which reads the atom's state by using *deref* will always see a consistent view of its value.

This is particularly useful when altering shared mutable state which cannot be changed atomically. Atoms enable atomic semantics for such objects.

Because atoms are themselves refs and inherit from *ARef*, it is also possible to add watches to them. Watches can be thought of callbacks which are invoked when the atom's state changes.

For example, if we would like to log each time a client connects, we can write a watch that will be responsible for this and then add it to the state atom:

```
>>> state = Atom({'active_conns': 0, 'clients': set([])})
>>> def log_new_clients(k, ref, old, new):
...     if not new['active_conns'] > old['active_conns']:
...         return
...     old_clients = old['clients']
...     new_clients = new['clients']
...     print 'new client', new_clients.difference(old_clients)
>>> state.add_watch('log_new_clients', log_new_clients)
```

We have added a watch which will print out a message when the client count has increased, i.e. a client has been added. Note that for a real world application, a proper logging facility should be preferred over print.

Watches are keyed by the first value passed to *add_watch* and are invoked whenver the atom changes with the key, reference, old state, and new state as parameters.

Note that watch functions may be called from multiple threads at once and therefore their ordering is not guaranteed. For instance, an atom's state may change, and before the watches can be notified another thread may alter the atom and trigger notifications. It is possible for the second thread's notifications to arrive before the first's.

compare_and_set (*oldval*, *newval*)

Given *oldval* and *newval*, sets the atom's value to *newval* if and only if *oldval* is the atom's current value. Returns *True* upon success, otherwise *False*.

Parameters

- **oldval** – The old expected value.
- **newval** – The new value which will be set if and only if *oldval*

equals the current value.

deref ()

Returns the value held.

reset (*newval*)

Resets the atom's value to *newval*, returning its old value.

Parameters **newval** – The new value to set.

swap (*fn*, **args*, ***kwargs*)

Given a mutator *fn*, calls *fn* with the atom's current state, *args*, and *kwargs*. The return value of this invocation becomes the new value of the atom. Returns the new value.

Parameters **fn** – A function which will be passed the current state. Should

return a new state. This absolutely MUST NOT mutate the reference to the current state! If it does, this function map loop indefinitely. :param *args: Arguments to be passed to *fn*. :param **kwargs: Keyword arguments to be passed to *fn*.

class atomos.atom.**ARef**

Ref object super type.

Refs may hold watches which can be notified when a value a ref holds changes. In effect, a watch is a callback which receives the key, object reference, oldval, and newval.

For example, a watch function could be constructed like this:

```
>>> def watch(k, ref, old, new):
...     print k, ref, old, new
>>> aref = ARef()
>>> aref.add_watch(watch)
```

However note that *ARef* should generally be subclassed, a la *Atom*, as it does not independently hold any value and functions merely as a container for the watch semantics.

add_watch (**args*, ***kwargs*)

Adds *key* to the watches dictionary with the value *fn*.

Parameters

- **key** – The key for this watch.
- **fn** – The value for this watch, should be a function. Note that

this function will be passed values which should not be mutated wihtout copying as other watches may in turn be passed the same reference!

get_watches ()

Returns the watches dictionary.

`notify_watches`(*oldval*, *newval*)
Passes *oldval* and *newval* to each *fn* in the watches dictionary, passing along its respective key and the reference to this object.

Parameters

- **`oldval`** – The old value which will be passed to the watch.
- **`newval`** – The new value which will be passed to the watch.

`remove_watch`(**args*, ***kwargs*)
Removes *key* from the watches dictionary.

Parameters **`key`** – The key of the watch to remove.

class `atomos.atomic.`**`AtomicReference`**(*value=None*)
A reference to an object which allows atomic manipulation semantics.

AtomicReferences are particularlly useful when an object cannot otherwise be manipulated atomically.

`compare_and_set`(*expect*, *update*)
Atomically sets the value to *update* if the current value is equal to *expect*.

Parameters

- **`expect`** – The expected current value.
- **`update`** – The value to set if and only if *expect* equals the

current value.

`get`()
Returns the value.

`get_and_set`(*value*)
Atomically sets the value to *value* and returns the old value.

Parameters **`value`** – The value to set.

`set`(*value*)
Atomically sets the value to *value*.

Parameters **`value`** – The value to set.

class `atomos.atomic.`**`AtomicBoolean`**(*value=False*)
A boolean value whichs allows atomic manipulation semantics.

`get`()
Returns the value.

class `atomos.atomic.`**`AtomicInteger`**(*value=0*)
An integer value which allows atomic manipulation semantics.

class `atomos.atomic.`**`AtomicLong`**(*value=0L*)
A long value which allows atomic manipulation semantics.

class `atomos.atomic.`**`AtomicFloat`**(*value=0.0*)
A float value which allows atomic manipulation semantics.

CHAPTER 4

API Multiprocessing

class `atomos.multiprocessing.atomic.`**`AtomicReference`**

class `atomos.multiprocessing.atomic.`**`AtomicBoolean`**`(`*value=False*`)`
A boolean value whichs allows atomic manipulation semantics.

`get``()`
Returns the value.

class `atomos.multiprocessing.atomic.`**`AtomicInteger`**`(`*value=0*`)`
An integer value which allows atomic manipulation semantics.

class `atomos.multiprocessing.atomic.`**`AtomicLong`**`(`*value=0L*`)`
A long value which allows atomic manipulation semantics.

class `atomos.multiprocessing.atomic.`**`AtomicFloat`**`(`*value=0.0*`)`
A float value which allows atomic manipulation semantics.

Index