fin Documentation

Release 2.1.8

Steve Stagg

Apr 19, 2017

Contents

1 About 1

2 Notable Modules 3

Python Module Index 13

CHAPTER 1

About

Fin is a small utility library for Python. It has several modules for doing common things, with a focus on keeping users' code clean and simple. With fin (*fin.contextlog*), producing the following is trivial:

Source

The above graphic was produced by the following code:

```python
import fin.contextlog
import functools

Log = functools.partial(fin.contextlog.Log, theme="mac")

...

def main():
    left = 10
    with Log("Processing Users"):
        for user in NAMES:
            with Log(user):
                with Log("Creating account"):
                    time.sleep(random.random()/4)
                with Log("Setting up homedir"):
                    time.sleep(random.random()/4)
                with Log("Creating primary key") as l:
                    time.sleep(random.random()/5)
                    left -= 1
                    if left == 0:
                        raise RuntimeError("Not enough entropy")
                    l.output(random.choice(keys).strip())
```

```python
if __name__ == '__main__':
    main()
```

CHAPTER 2

Notable Modules

fin.cache

cache is designed to make caching the results of slow method simple, and painless. It's very similar in nature to the common python @memoize pattern, but the results are stored on an object, rather than inside a closure, making caching of class-specific method results much simpler and easier.

An obvious example would be a database connection:

```
class DB(object):

    def __init__(self, dsn):
        self.dsn = dsn

    @fin.cache.depends("dsn")
    @fin.cache.property
    def connection(self):
        return dblibrary.connect(self.dsn)
```

In this example, the connection attribute is cached for each DB instance, allowing for very natural usage:

```
>>> db1 = DB("test")
>>> db1.connection
<dblibrary.Connection at 0x7f904012fb90>
>>> db1.connection
<dblibrary.Connection at 0x7f904012fb90>
>>> db2 = DB("test")
<dblibrary.Connection at 0x1a111111de55>
```

The `` `@fin.cache.depends` `` part means that if the dsn changes for a DB, the next time db.connection is requested, the class will create a *new* connection, using the new dsn, update the cache, and start using that instead. This allows code to be written in a very natural fashion, and not to worry about state management.

Code Docs

class `fin.cache.`**`property`**`(`*fun*, *wrapper=<function method>*`)`

This decorator behaves like the builtin `@property` decorator, but caches the results, similarly to `fin.cache.method`:

```
>>> class Example(object):

>>>     @fin.cache.property
>>>     def number(self):
>>>         time.sleep(1)
>>>         return 4

>>> e = Example()
>>> print "Slow:", e.number
Slow: 4  (1 second later)
>>> print "Fast:", e.number
Fast: 4  (immediately)
>>> E.number.reset(e)
>>> print "Slow:", e.number
Slow: 4  (1 second later)
```

`@fin.cache.property()` descriptors support assignment. The attribute can be assigned a callable, taking one argument, which will always be called on attribute access, and the result returned. This is best shown by an example, continuing from the previous:

```
>>> e = Example()
>>> f = Example()
>>> print(e.number, f.number)
4 4
>>> f.number = lambda e: 8
>>> print(e.number, f.number, f.number)
4 8 8
>>> e.number = lambda e: random.randint(1, 10)
>>> print(e.number, f.number, f.number)
4 5 9
```

`has_cached``(`*inst*`)`

Returns `True` if a result has been cached for the property on the specified instance:

```
>>> class Example(object):
>>>
>>>     @fin.cache.property
>>>     def one(self):
>>>         return 1
```

```
>>> e = Example()
>>> Example.one.has_cached(e)
False
>>> e.one
1
>>> Example.one.has_cached(e)
True
>>> Example.one.reset(e)
>>> Example.one.has_cached(e)
False
```

`reset``(`*inst*`)`

'Forgets' any cached value for instance `inst`. Use as shown in in the example above.

`fin.cache.`**`method`**`(`*fun*`)`

This is the core of `fin.cache`. Typically used as a decorator on class or instance methods. When a method is decorated with this function, repeatedly calling it, on the same object, with the same arguments*, will only cause the method to be called once. The result of that call is stored on the object, and is automatically returned subsequently.

An interesting (contrived) example from the tests:

```
class Factorial(object):

    #Try commenting out the @fin.cache.method line and see what happens..
    @fin.cache.method
    def factorial(self, num):
        if num <= 1:
            return 1
        return num * self.factorial(num - 1)

factorial = Factorial()
for i in range(2000):
    factorial.factorial(i)
```

*** NOTE: Arguments are tested by equality (`a==b` not `a is b`). This can, in a very few situations, lead to unexpected resu**

Also, the result value is cached by reference. If a cached method returns, for example, a `list`, then any modifications to that list will be shared amongst all return values, which can lead to some strange effects if mis-used:

```
>>> class Bad(object):

>>>     @classmethod
>>>     @fin.cache.method
>>>     def bad_range(self, n):
>>>         return list(range(n))

>>> nums = Bad.bad_range(1)
>>> print(Bad.bad_range(1), Bad.bad_range(2))
[0] [0, 1]
>>> nums.extend(Bad.bad_range(2))
>>> print(Bad.bad_range(1), Bad.bad_range(2))
[0, 0, 1] [0, 1]
```

Calling `reset(object)` on the descriptor will cause the cache to be cleared for that object, to continue the example:

```
>>> Bad.bad_range.reset(Bad)
>>> print(Bad.bad_range(1), Bad.bad_range(2))
[0] [0, 1]
```

When used on an instance method, rather than a classmethod, the object instance should be passed into reset.

`fin.cache.`**`depends`**`(`**attributes*`)`

Used in conjunction with *`fin.cache.property()`* or *`fin.cache.method()`*, this decorator tags a cached method as depending on the value of the specified named attribute on the method's object. Note, this does mean all dependant properties are evaluated every time the cached method is called.

As a naive example, to cache an object hash, where the hashing algorithm might change:

```
>>> class HashedValue(object):

>>>     def __init__(self, value):
>>>         self.value = value
>>>         self.hash_method = "sha1"

>>>     @fin.cache.property
>>>     @fin.cache.depends("value", "hash_method")
>>>     def hash(self):
>>>         print('** Calculating **')
>>>         return getattr(hashlib, self.hash_method)(self.value).hexdigest()

>>> val = HashedValue('hello')
>>> val.hash
** Calculating **
'aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d'
>>> val.hash
'aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d'
>>> val.hash_method = 'sha1'
>>> val.hash
'aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d'
>>> val.hash_method = 'sha512'
>>> val.hash
** Calculating **

 ↪'9b71d224bd62f3785d96d46ad3ea3d73319bfbc2890caadae2dff72519673ca72323c3d99ba5c11d7c7acc6e14b8c
 ↪'
```

In this case, `instance.hash` will always reflect the currently selected hashing method, and the current value, but will not re-hash the value needlessly.

`fin.cache.`**`uncached_property`**(*fun*)

Behaves like the builtin `@property` decorator, but supports the same assignment logic as `@fin.cache.property`. This method performs **no** caching, but is syntactic sugar.

`fin.cache.`**`invalidates`**(*other*)

Explicitly clears another cached method when this method is called, on the same object:

```
>>> class Dice(object):
>>>     @fin.cache.property
>>>     def current_value(self):
>>>         return random.randint(1, 7)
>>>
>>>     @fin.cache.invalidates(current_value)
>>>     def roll(self):
>>>         print("Rolling...")
```

```
>>> die = Dice()
>>> die.current_value
1
>>> die.current_value
1
>>> die.roll()
>>> die.current_value
5
```

`fin.cache.`**`generator`**(*fun*)

Use with care! This generator keeps a reference to all generated values for the lifetime of the cache (unless manually cleared). Given that generators are often used to handle larger volumes of data, this may cause memory issues if used incorrectly. This decorator is useful as a speed optimisation, but comes with a memory cost.

Acts like `@fin.cache.method` but for methods that return a generator (or uses :keyword:yield). Repeated calls to this method return an object that can be used to iterate over the generated values from the start:

```
>>> class Example(object):
>>>
>>>     @fin.cache.generator
>>>     def slow_range(self, num):
>>>         for i in range(num):
>>>             time.sleep(0.2)
>>>             yield i

>>> e = Example()
>>> e.slow_range(10)
<fin.cache.TeeGenerator at 0x7f9041062650> # Fast
>>> list(e.slow_range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]             # Slow
>>> list(e.slow_range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]             # Fast (already calculated)
>>> list(e.slow_range(5))
[0, 1, 2, 3, 4]                            # Slow (different arguments used)
>>> list(Example().slow_range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]             # Slow (different instance used)
>>> Example.slow_range.reset(e) # Free the memory..
>>> list(e.slow_range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]             # Slow (cache has been cleared)
```

The generator is evaluated on-demand, and lazily, so the following works:

```
>>> timeit(lambda: list(e.slow_range(3)), number=1)
# Takes 0.6 seconds to evaluate the list
0.60060    #  [0, 1, 2]
>>> e.slow_range.reset(e)
>>> timeit(lambda: zip(e.slow_range(10), e.slow_range(10)))
# Both generators are being evaluated at the same time, but the time stays the␣
↪same:
0.60075   #  [(0, 0), (1, 1), (2, 2)]
```

fin.color

Colorful console output not only makes output look prettier, it can significantly help with readability.

This module provides a simple wrapper that makes outputting colored text *really* simple:

```
C.red("this is red") + C.blue.bold("this is blue and bold")
```

Code Docs

`fin.color.`**C**

An instance of *Color* that is most appropriate for sys.stdout. Typically this will be a VtColor object, but when stdout is redirected to a file, or other non-terminal output then it will revert to NoColor.

This constant allows code to simply refer to, for example:

```
>>> fin.color.C.blue('hi')
'\x1b[34mhi\x1b[0m'
```

class `fin.color.`**`Color`**(*parts=()*)

Bases: `object`

A simple class for producing pretty terminal output. While *`Color`* is abstract, *`VtColor`* provides common VT-100 (xterm) compatible output. This is a very light, small library, and doesn't deal with curses or terminfo.

The module global `C` is created at import time. If standard out appears to support color output, then this will be an instance of *`VtColor`*, otherwise, *`NoColor`*.

Typical Usage:

```
c = fin.color.C
print c.blue + "I'm blue, da-ba-dee da-ba-dai" + c.reset
print c.red.bold("Color") + c.red.blue("Blind")
print c.green("In") + c.bg_green.black("verse") # Note assumes a white-on-black␣
↪color scheme.
```

class `fin.color.`**`NoColor`**(*parts=()*)

Bases: *`fin.color.Color`*

A color class for when no color information should be output. For example, if output can be redirected to a terminal OR a log file, NoColor can be transparently swapped in for the VtColor class to ensure that the data in the log file does not include a large number of escape codes.

class `fin.color.`**`VtColor`**(*parts=()*)

Bases: *`fin.color.Color`*

The color class for VT-compatible terminals (basically all non-windows terminals).

The color attributes may be used in two ways. Getting a color is a matter of referencing the correct attribute:

```
>>> C.red
<fin.color.VtColor at 0x7f904012fb90>
```

This may then be converted to a string, for printing:

```
>>> str(C.red)
'\x1b[34m'
>>> C.red + "foo" + C.reset
'\x1b[31mfoo\x1b[0m'
```

The alternate syntax is to *call* the color, passing in a string, which implicitly add the color code before the string, and adds a reset afterwards (NOTE: the reset will reset all color information, including any inherited):

```
>>> C.red("hello") + C.blue("world")
'\x1b[31mhello\x1b[0m\x1b[34mworld\x1b[0m'
```

Printing any of the above in a standard (non-windows) terminal, will result in the correct colored output.

The available colors are listed below. All colors may be prefixed with '**bg_**', and colors may be combined by further attribute access:

```
>>> C.bg_blue.white.bold("Bold, white-on-blue text")
'\x1b[44;37;1mBold, white-on-blue text\x1b[0m'
```

Two special attributes: 'bold', and 'reset' respectively turn the text bold (or use bright colors, depending on console) and reset all color attributes.

COLORS = ['black', 'red', 'green', 'yellow', 'blue', 'purple', 'cyan', 'white']
All colors that may be referenced

EXTRA = {'bold': 1, 'reset': 0}
Non-color attributes

fin.color.**auto_color**(*stream=<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>*)
This does some simple tests to determine if the output stream supports colors, returning the corect Color class for the stream.

The lookup is intentionally kept simple, as this has proved to capture 99% of cases without adding the burden of more complicated capabilities databases.

fin.contextlog

It's not uncommon for a python script that runs for a long time to produce hard-to-debug errors:

Let's assume there is a script that configures and starts a set of servers, and running it gives this error:

```
Traceback (most recent call last):
  File "test.py", line 31, in <module>
    main()
  File "test.py", line 28, in main
    setup_servers()
  File "test.py", line 12, in setup_servers
    start_server(server)
  File "test.py", line 16, in start_server
    configure_replication(server)
  File "test.py", line 24, in configure_replication
    subprocess.check_call(['replication_agent', machine_from, machine_to])
  File "/usr/lib/python2.7/subprocess.py", line 535, in check_call
    retcode = call(*popenargs, **kwargs)
  File "/usr/lib/python2.7/subprocess.py", line 522, in call
    return Popen(*popenargs, **kwargs).wait()
  File "/usr/lib/python2.7/subprocess.py", line 710, in __init__
    errread, errwrite)
  File "/usr/lib/python2.7/subprocess.py", line 1335, in _execute_child
    raise child_exception
OSError: [Errno 2] No such file or directory
```

We can see that calling a subprocess has failed. But for which server? Which replication was it configuring? There is a lot of critical information missing from this traceback.

The solution is typically to use logging to track progress, but this can also be confusing:

```
Starting database
Starting app_server
Booting Image
Configured database > app_server replication
Starting cache
Booting Image
Configured database > cache replication
Starting http_server
Booting Image
Traceback (most recent call last):
  File "test.py", line 31, in <module>
 ...
```

Was the app still running the 'starting http_server' step? what exact step did it actually fail on? Do you add logging before AND after every step in the process? This approach also produces a lot of noise that may not be relevant to debugging the problem.

fin.contextlog is designed to solve this in a nice, elegant way. It provides two context managers that log when the manager is entered, and exited, and can be used to pinpoint *exactly* where in the script the failure happened:

```
Setting-up servers:
| database:
| | Booting Image: OK
| | Configuring replication: OK
| `- OK
| app_server:
| | Booting Image: OK
| | Configuring replication:
| | | Configuring replication database > app_server: OK
| | `- OK
| `- OK
...
| http_server:
| | Booting Image: OK
| | Configuring replication:
| | | Configuring replication database > http_server: FAIL
| | `- FAIL
| `- FAIL
`- FAIL
Traceback (most recent call last):
  File "test.py", line 36, in <module>
    main()
...
  File "/usr/lib/python2.7/subprocess.py", line 1335, in _execute_child
    raise child_exception
OSError: [Errno 2] No such file or directory
```

This output also makes following the progress of a long-running script very easy.

It may be that you don't want all of this noisy output, and prefer your app to only produce output if there's an error to report. This can be done with the *fin.contextlog.CLog* class, which only outputs information related to failing steps (or steps that produce output):

```
Setting-up servers:
| http_server:
| | Configuring replication:
| | | Configuring replication database > http_server:
| | | `- FAIL
| | `- FAIL
| `- FAIL
`- FAIL
Traceback (most recent call last):
  File "test.py", line 36, in <module>
...
```

Usage

Usage is simple, import either the Log or CLog class, and use them as a context manager:

```
from fin.contextlog import Log
with Log("foo"):
    with Log("bar") as log:
        log.output('baz')
```

If a custom theme is required, or other customization (a specific stream etc.) it's common to wrap the Log or CLog constructors in a functools.partial call:

```
import fin.contextlog
import functools

Log = functools.partial(fin.contextlog.Log, theme="mac")

with Log('Even prettier output'):
    pass
```

Code Docs

class `fin.contextlog.`**`CLog`**(*message*, *ok_msg=None*, *fail_msg=None*, *theme='mac'*, *stream=<_io.FileIO name='<stderr>' mode='wb' closefd=False>*)

A logging context manager, similar to *`fin.contextlog.Log`*, that only produces output if an exception occurs or log.output()/log.format() is called.

This means that an app/script that uses CLog could run silently if there are no errors, but show all the context if/when an error does occur

class `fin.contextlog.`**`Log`**(*message*, *ok_msg=None*, *fail_msg=None*, *theme='mac'*, *stream=<_io.FileIO name='<stderr>' mode='wb' closefd=False>*)

A logging context manager that provides easy to understand, and useful console output.

Multiple logs may be nested (provided they use the same output stream) and the output reflects this, allowing for complex processing to be reflected simply to the user

Example

```
>>> from fin.contextlog import Log
```

```
>>> def do_stuff():
>>>     with Log("Doing stuff"):
>>>         pass
```

Parameters

- **`message`** – A string to be output when the context manager is entered
- **`ok_msg`** – Defaults to the theme-specific 'OK' message. The string that is printed if the Context manager exits without error
- **`fail_msg`** – Defaults to the theme-specific 'Fail' message. The string that is printed if the Context manager detects an exception
- **`theme`** – contextlib has several themes that control how the output is displayed, common ones are 'default', 'aa', and 'mac' Note, for performance reasons, themes cannot be mixed on the same stream.
- **`stream`** – A file-like object (default is stderr) that the context output is written to.

`format` (*msg*, **args*, ***kwargs*)

As Log.output, but provides prettier output. Using pformat, color-based substitution, and word wrapping.

Parameters

- **`msg`** – If msg is a string, any args or kwargs are used in percent subsitutions, with the substitution values output in a different color. log.format('user %s', user.name) might output '|+ user *bob*' (where bob is displayed in blue)
- **`msg`** – If msg is not a string, then pprint.pformat is called on it, and the result is output

`output` (*msg*)

Output *msg* to the stream, but correctly indented to fit nicely within the current contextlog output.

Parameters **`msg`** – String to be output to the stream

Python Module Index

f

fin.cache, 4
fin.color, 7
fin.contextlog, 11

Index

A

auto_color() (in module fin.color), 9

C

C (in module fin.color), 7
CLog (class in fin.contextlog), 11
Color (class in fin.color), 8
COLORS (fin.color.VtColor attribute), 8

D

depends() (in module fin.cache), 5

E

EXTRA (fin.color.VtColor attribute), 9

F

fin.cache (module), 4
fin.color (module), 7
fin.contextlog (module), 11
format() (fin.contextlog.Log method), 11

G

generator() (in module fin.cache), 6

H

has_cached() (fin.cache.property method), 4

I

invalidates() (in module fin.cache), 6

L

Log (class in fin.contextlog), 11

M

method() (in module fin.cache), 5

N

NoColor (class in fin.color), 8

O

output() (fin.contextlog.Log method), 12

P

property (class in fin.cache), 4

R

reset() (fin.cache.property method), 4

U

uncached_property() (in module fin.cache), 6

V

VtColor (class in fin.color), 8