Python Toolbox Documentation

Release 0.1.0

Ram Rachum

July 21, 2016

Contents

Contents:

CHAPTER 1

Topical guides to the Python Toolbox

(This section is still incomplete.)

These are topical guides about the various modules in the Python Toolbox.

It focuses on giving the motivation for each module in the Python Toolbox, explaining what it's good for and the basics of using it.

1.1 `abc_tools` - documentation not written

1.2 `address_tools` - documentation not written

1.3 `arguments_profile` - documentation not written

1.4 `binary_search` - documentation not written

1.5 `caching`

The `caching` modules provides tools related to caching:

1.5.1 `caching.cache()`

A caching decorator that understands arguments

The idea of a caching decorator is very cool. You decorate your function with a caching decorator:

```
>>> from python_toolbox import caching
>>>
>>> @caching.cache
... def f(x):
...     print('Calculating...')
...     return x ** x # Some long expensive computation
```

And then, every time you call it, it'll cache the results for next time:

```
>>> f(4)
Calculating...
256
>>> f(5)
Calculating...
3125
>>> f(5)
3125
>>> f(5)
3125
```

As you can see, after the first time we calculate `f(5)` the result gets saved to a cache and every time we'll call `f(5)` Python will return the result from the cache instead of calculating it again. This prevents making redundant performance-expensive calculations.

Now, depending on the function, there can be many different ways to make the same call. For example, if you have a function defined like this:

```
def g(a, b=2, **kwargs):
    return whatever
```

Then `g(1)`, `g(1, 2)`, `g(b=2, a=1)` and even `g(1, 2, **{})` are all equivalent. They give the exact same arguments, just in different ways. Most caching decorators out there don't understand that. If you call `g(1)` and then `g(1, 2)`, they will calculate the function again, because they don't understand that it's exactly the same call and they could use the cached result.

Enter `caching.cache()`:

```
>>> @caching.cache()
... def g(a, b=2, **kwargs):
...     print('Calculating')
...     return (a, b, kwargs)
...
>>> g(1)
Calculating
(1, 2, {})
>>> g(1, 2) # Look ma, no calculating:
(1, 2, {})
>>> g(b=2, a=1) # No calculating again:
(1, 2, {})
>>> g(1, 2, **{}) # No calculating here either:
(1, 2, {})
>>> g('something_else') # Now calculating for different arguments:
Calculating
('something_else', 2, {})
```

As you can see above, `caching.cache()` analyzes the function and understands that calls like `g(1)` and `g(1, 2)` are identical and therefore should be cached together.

Both limited and unlimited cache

By default, the cache size will be unlimited. If you want to limit the cache size, pass in the `max_size` argument:

```
>>> @caching.cache(max_size=7)
... def f(): pass
```

If and when the cache size reaches the limit (7 in this case), old values will get thrown away according to a LRU order.

Sleekrefs

`caching.cache()` arguments with sleekrefs. Sleekrefs are a more robust variation of weakrefs. They are basically a gracefully-degrading version of weakrefs, so you can use them on un-weakreff-able objects like `int`, and they will just use regular references.

The usage of sleekrefs prevents memory leaks when using potentially-heavy arguments.

1.5.2 `caching.CachedType`

A class that automatically caches its instances

Sometimes you define classes whose instances hold absolutely no state on them, and are completey determined by the arguments passed to them. In these cases using `caching.CachedType` as a metaclass would cache class instances, preventing more than one of them from being created:

```
>>> from python_toolbox import caching
>>>
>>> class A(object):
...       __metaclass__ = caching.CachedType
...       def __init__(self, a=1, b=2):
...           self.a = a
...           self.b = b
```

Now every time you create an instance, it'll be cached:

```
>>> my_instance = A(b=3)
```

And the next time you'll create an instance with the same arguments:

```
>>> another_instance = A(b=3)
```

No instance will be actually created; the same instance from before will be used:

```
>>> assert another_instance is my_instance
```

1.5.3 `caching.CachedProperty`

A cached property

Oftentimes you have a `property` on a class that never gets changed and needs to be calculated only once. This is a good situation to use `caching.CachedProperty` in order to have that property be calculated only one time per instance. Any future accesses to the property will use the cached value.

Example:

```
>>> import time
>>> from python_toolbox import caching
>>>
>>> class MyObject(object):
...     # ... Regular definitions here
...     def _get_personality(self):
...         print('Calculating personality...')
...         time.sleep(5) # Time consuming process...
...         return 'Nice person'
...     personality = caching.CachedProperty(_get_personality)
```

Now we create an object and calculate its "personality":

```
>>> my_object = MyObject()
>>> my_object.personality
'Nice person'
>>> # We had to wait 5 seconds for the calculation!
```

Consecutive calls will be instantaneous:

```
>>> my_object.personality
'Nice person'
>>> # That one was cached and therefore instantaneous!
```

1.6 change_tracker - documentation not written

1.7 cheat_hashing - documentation not written

1.8 color_tools - documentation not written

1.9 comparison_tools - documentation not written

1.10 context_managers - documentation not written

1.11 copy_mode - documentation not written

1.12 copy_tools - documentation not written

1.13 cute_inspect - documentation not written

1.14 cute_iter_tools - documentation not written

1.15 cute_profile - documentation not written

1.16 cute_testing - documentation not written

1.17 decorator_tools - documentation not written

1.18 dict_tools - documentation not written

1.19 emitters - documentation not written

1.20 exceptions - documentation not written

1.21 freezers - documentation not written

1.22 function_anchoring_type - documentation not written

1.23 gc_tools - documentation not written

1.24 identities - documentation not written

1.25 import_tools - documentation not written

1.26 infinity - documentation not written

1.27 introspection_tools - documentation not written

CHAPTER 2

Miscellaneous topics

2.1 Mailing Lists

There are three Python Toolbox groups, a.k.a. mailing lists:

- If you need help with Python Toolbox, post a message on the Python Toolbox Google Group.
- If you want to help on the development of Python Toolbox itself, come say hello on the python-toolbox-dev Google Group.
- If you want to be informed on new releases of the Python Toolbox, sign up for the low-traffic python-toolbox-announce Google Group.

This documentation is still incomplete. If you have any questions or feedback, say hello on the mailing list!

The Python Toolbox repository is at: https://github.com/cool-RR/python_toolbox

Feel free to fork and send pull requests!