Python Toolbox Documentation

Release 0.9.3

Ram Rachum

September 03, 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`

The problem that `address_tools` was originally designed to solve was getting the "address" of a class, and possibly shortening it to an equivalent but shorter string. But after I implemented that, I realized that this could be generalized into a pair of functions, `address_tools.describe()` and `address_tools.resolve()`, that can replace the built-in `repr()` and `eval()` functions.

So, Python has two built-in functions called `repr()` and `eval()`. You can say that they are opposites of each other: `repr()` "describes" a Python object as a string, and `eval()` evaluates a string into a Python object.

When is this useful? This is useful in various cases: For example when you have a GUI program that needs to show the user Python objects and let him manipulate them. As a more well-known example, Django uses something like `eval()` to let the user specify functions without importing them, both in `settings.py` and `urls.py`.

In some easy cases, `repr()` and `eval()` are the exact converses of each other:

```
>>> repr([1, 2, 'meow', {3: 4}])
"[1, 2, 'meow', {3: 4}]"
>>> eval(
...     repr(
...         [1, 2, 'meow', {3: 4}]
...     )
... )
[1, 2, 'meow', {3: 4}]
```

When you put a simple object like that in `repr()` and then put the resulting string in `eval()`, you get the original object again. That's really pretty, because then we have something like a one-to-one correspondence between objects and strings used to describe them.

In a happy-sunshine world, there would indeed be a perfect one-to-one mapping between Python objects and strings that describe them. You got a Python object? You can turn it into a string so a human could easily see it, and the string will be all the human will need to create the object again. But unfortunately some objects just can't be meaningfully described as a string in a reversible way:

```
>>> import threading
>>> lock = threading.Lock()
>>> repr(lock)
'<thread.lock object at 0x00ABF110>'
>>> eval(repr(lock))
Traceback (most recent call last):
  File "", line 1, in
invalid syntax: , line 1, pos 1
```

A lock object is used for synchronization between threads. You can't really describe a lock in a string in a reversible way; a lock is a breathing, living thing that threads in your program interact with, it's not a data-type like a list or a dict.

So when we call `repr()` on a lock object, we get something like `'<thread.lock object at 0x00ABF110>'`. Enveloping the text with pointy brackets is Python's way of saying, "you can't turn this string back into an object, sorry, but I'm still going to give you some valuable information about the object, in the hope that it'll be useful for you." This is good behavior on Python's part. We may not be able to use `eval()` on this string, but at least we got some info about the object, and introspection is a *very* useful ability.

So some objects, like lists, dicts and strings, can be easily described by `repr()` in a reversible way; some objects, like locks, queues, and file objects, simply cannot by their nature; and then there are the objects in between.

1.2.1 Classes, functions, methods, modules

What happens when we run `repr()` for a Python class?

```
>>> import decimal
>>> repr(decimal.Decimal)
"<class 'decimal.Decimal'>"
```

We get a pointy-bracketed un-`eval`-able string. How about a function?

```
>>> import re
>>> repr(re.match)
'<function match at 0x00E8B030>'
```

Same thing. We get a string that we can't put back in `eval()`. Is this really necessary? Why not return `'decimal.Decimal'` or `'re.match'` so we could `eval()` those later and get the original objects?

It *is* sometimes helpful that the `repr()` string `"<class 'decimal.Decimal'>"` informs us that this is a class; but sometimes you want a string that you can turn back into an object. Although... `eval()` might not be able to find it, because `decimal` might not be currently imported.

Enter `address_tools`:

1.2.2 `address_tools.describe()` and `address_tools.resolve()`

Let's play with `address_tools.describe()` and `address_tools.resolve()`:

```
>>> from python_toolbox import address_tools
>>> import decimal
>>> address_tools.describe(decimal.Decimal)
'decimal.Decimal'
```

That's a nice description string! We can put that back into `resolve` and get the original class:

```
>>> address_tools.resolve(address_tools.describe(decimal.Decimal)) is decimal.Decimal
True
```

We can use `resolve` to get this function, without `re` being imported, and it will import `re` by itself:

```
>>> address_tools.resolve('re.match')
<function match at 0x00B5E6B0>
```

This shtick also works on classes, functions, methods, modules, and possibly other kinds of objects.

1.3 `binary_search` - documentation not written

1.4 `caching`

The `caching` modules provides tools related to caching:

1.4.1 `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.

1.4.2 `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(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
```

1.4.3 `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!
```

1.5 change_tracker - documentation not written

1.6 cheat_hashing - documentation not written

1.7 color_tools - documentation not written

1.8 combi - Documentation on Combi site

Please go to Combi's documentation here.

1.9 comparison_tools - documentation not written

1.10 context_management

1.10.1 Context managers are awesome

I love context managers, and I love the `with` keyword. If you've never dealt with context managers or `with`, here's a practical guide which explains how to use them. You may also read the more official **PEP 343** which introduced these features to the language.

Using `with` and context managers in your code contributes a lot to making your code more beautiful and maintainable. Every time you replace a `try`-`finally` clause with a `with` clause, an angel gets a pair of wings.

Now, you don't *need* any official `ContextManager` class in order to use context managers or define them; you just need to define `__enter__()` and `__exit__()` methods in your class, and then you can use your class as a context manager. *But*, if you use the `ContextManager` class as a base class to your context manager class, you could enjoy a few more features that might make your code a bit more concise and elegant.

1.10.2 What does ContextManager add?

The `ContextManager` class allows using context managers as decorators (in addition to their normal use) and supports writing context managers in a new form called `manage_context()`. (As well as the original forms). First let's import:

```
>>> from python_toolbox import context_management
```

Now let's go over the features one by one.

The `ContextManager` class allows you to **define** context managers in new ways and to **use** context managers in new ways. I'll explain both of these; let's start with **defining** context managers.

1.10.3 Defining context managers

There are 3 different ways in which context managers can be defined, and each has their own advantages and disadvantages over the others.

- The classic way to define a context manager is to define a class with `__enter__()` and `__exit__()` methods. This is allowed, and if you do this you should still inherit from `ContextManager`. Example:

```
>>> class MyContextManager(context_management.ContextManager):
...     def __enter__(self):
...         pass # preparation
...     def __exit__(self, type_=None, value=None, traceback=None):
...         pass # cleanup
```

- As a decorated generator, like so:

```
>>> @context_management.ContextManagerType
... def MyContextManager():
...     # preparation
...     try:
...         yield
...     finally:
...         pass # cleanup
```

The advantage of this approach is its brevity, and it may be a good fit for relatively simple context managers that don't require defining an actual class. This usage is nothing new; it's also available when using the standard library's `contextlib.contextmanager()` decorator. One thing that is allowed here that `contextlib` doesn't allow is to yield the context manager itself by doing `yield context_management.SelfHook`.

- The third and novel way is by defining a class with a `manage_context()` method which returns a decorator. Example:

```
>>> class MyContextManager(ContextManager):
...     def manage_context(self):
...         do_some_preparation()
...         with other_context_manager:
...             yield self
```

This approach is sometimes cleaner than defining `__enter__()` and `__exit__()`; especially when using another context manager inside `manage_context()`. In our example we did `with other_context_manager` in our `manage_context()`, which is shorter, more idiomatic and less double-underscore-y than the equivalent classic definition:

```
>>> class MyContextManager(object):
...         def __enter__(self):
...             do_some_preparation()
...             other_context_manager.__enter__()
...             return self
...         def __exit__(self, *exc):
...             return other_context_manager.__exit__(*exc)
```

Another advantage of the `manage_context()` approach over `__enter__()` and `__exit__()` is that it's better at handling exceptions, since any exceptions would be raised inside `manage_context()` where we could `except` them, which is much more idiomatic than the way `__exit__()` handles exceptions, which is by receiving their type and returning whether to swallow them or not.

These were the different ways of defining a context manager. Now let's see the different ways of **using** a context manager:

1.10.4 Using context managers

There are 2 different ways in which context managers can be used:

- The plain old honest-to-Guido `with` keyword:

```
>>> with MyContextManager() as my_context_manager:
...     do_stuff()
```

- As a decorator to a function:

```
>>> @MyContextManager()
... def do_stuff():
...     pass # doing stuff
```

When the `do_stuff` function will be called, the context manager will be used. This functionality is also available in the standard library of Python 3.2+ by using `contextlib.ContextDecorator`, but here it is combined with all the other goodies given by `ContextManager`. Another advantage that `ContextManager` has over `contextlib.ContextDecorator` is that it uses Michele Simionato's excellent decorator module to preserve the decorated function's signature.

That's it. Inherit all your context managers from `ContextManager` (or decorate your generator functions with `ContextManagerType`) to enjoy all of these benefits.

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`

The `cute_profile` module allows you to profile your code (i.e. find out which parts make it slow) by giving a nicer interface to the `cProfile` library from Python's standard library.

1.15.1 What is "profiling"?

(Programmers experienced with profilers may skip this section.)

To "profile" a piece of code means to run it while checking how long it takes, and how long each function call inside the code takes. When you use a "profiler" to profile your program, you get a table of (a) all the functions calls that were made by the program, (b) how many times each function was called and (c) how long the function calls took.

A profiler is an indispensable programming tool, because it allows the programmer to understand which parts of his code take the longest. Usually, when using a profiler, you discover that only a few small parts of his code take most of the runtime of your program. And quite often, it's not the parts of code that you *thought* were the slow ones.

Once you realize which parts of the program cause slowness, you can focus your efforts on those problematic parts only, optimizing them or possibly redesigning the way they work so they're not slow anymore. Then the whole program becomes faster.

1.15.2 Profiling Python code with `cute_profile`

Python supplies a module called `cProfile` in its standard library. `cProfile` is a good profiler, but its interface can be inconvenient to work with. The `cute_profile` module has a more flexible interface, and it uses `cProfile` under the hood to do the actual profiling.

Let's profile an example program. Our example would be a function called `get_perfects`, which finds perfect numbers:

```
>>> def get_divisors(x):
...     '''Get all the integer divisors of `x`.'''
...     return [i for i in xrange(1, x) if (x % i == 0)]
...
>>> def is_perfect(x):
...     '''Is the number `x` perfect?'''
...     return sum(get_divisors(x)) == x
...
>>> def get_perfects(top):
...     '''Get all the perfect numbers up to the number `top`.'''
...     return [i for i in xrange(1, top) if is_perfect(i)]
>>> print(get_perfects(20000))
```

The result is `[6, 28, 496, 8128]`. However, this function takes a few seconds to run. That's fairly long. Let's use `cute_profile` to find out *why* this function is taking so long. We'll add the `cute_profile.profile_ready()` decorator around `get_perfects`:

```
>>> from python_toolbox import cute_profile
>>> @cute_profile.profile_ready()
... def get_perfects(top):
...     '''Get all the perfect numbers up to the number `top`.'''
...     return [i for i in xrange(1, top) if is_perfect(i)]
```

Now before we run `get_perfects`, we set it to profile:

```
>>> get_perfects.profiling_on = True
```

And now we run it:

```
>>> print(get_perfects(20000))
```

We still get the same result, but now a profiling table gets printed:

```
      60000 function calls in 7.997 seconds

Ordered by: cumulative time

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     1    0.000    0.000    7.997    7.997 <string>:1(<module>)
     1    0.020    0.020    7.997    7.997 <pyshell#1>:2(get_perfects)
 19999    0.058    0.000    7.977    0.000 <pyshell#0>:5(is_perfect)
 19999    7.898    0.000    7.898    0.000 <pyshell#0>:1(get_divisors)
 19999    0.021    0.000    0.021    0.000 {sum}
     1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

This table shows how long each function took. If you want to understand *exactly* what each number says in this table, see `cProfile.run()`.

The `tottime` column says how much time was spent inside this function, across all calls, and without counting the time that was spent in sub-functions. See how the `get_divisors` function in our example has a very high `tottime` of 7.898 seconds, which is about 100% of the entire run time. This means that `get_divisors` is what's causing our program to run slow, and if we'll want to optimize the program, we should try to come up with a smarter way of finding all of a number's divisors than going one-by-one over all numbers.

`profile_ready` has a bunch of other options. In brief:

- The `condition` argument is something like a "breakpoint condition" in an IDE: It can be a function, usually a lambda, that takes the decorated function and any arguments and returns whether or not to profile it this time.
- `off_after` means whether you want the function to stop being profiled after being profiled one time. Default is `True`.
- `sort` is an integer saying by which column the final results table should be sorted.

1.16 `cute_testing` - documentation not written

1.17 `decorator_tools` - documentation not written

1.18 `dict_tools` - documentation not written

1.19 `emitting` - documentation not written

1.20 `exceptions` - documentation not written

1.21 `file_tools` - documentation not written

1.22 `freezing` - documentation not written

1.23 `function_anchoring_type` - documentation not written

1.24 `gc_tools` - documentation not written

1.25 `human_names` - documentation not written

1.26 `identities` - documentation not written

1.27 `import_tools` - documentation not written

1.28 `introspection_tools` - documentation not written

1.29 `locking` - documentation not written

1.30 `logic_tools` - documentation not written

1.31 `math_tools` - documentation not written

1.32 `misc_tools` - documentation not written

1.33 `monkeypatching_tools` - documentation not written

1.34 `nifty_collections` - documentation not written

1.35 `os_tools` - documentation not written

1.36 `package_finder` - documentation not written

1.37 `path_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!

Python Toolbox on GitHub: https://github.com/cool-RR/python_toolbox

Python Toolbox on PyPI: https://pypi.python.org/pypi/python_toolbox

Feel free to fork and send pull requests!

The Python Toolbox was created by Ram Rachum. I provide Development services in Python and Django.

Index