tangled Documentation

Release 1.0a13.dev0

Wyatt Baldwin

Feb 14, 2018

Contents

Namespace for `tangled.*` projects and core utilities.

CHAPTER 1

Links

- Project Home Page
- Source Code (GitHub)

CHAPTER 2

Contents

2.1 API Documentation

2.1.1 Abstract Base Classes

Abstract base classes.

class tangled.abcs.**ACommand**(*parser*, *args*)
Abstract base class for tangled commands.

2.1.2 Decorators

class tangled.decorators.**cached_property**(**args*)
Similar to @property but caches value on first access.

When a cached property is first accessed, its value will be computed and cached in the instance's `__dict__`. Subsequent accesses will retrieve the cached value from the instance's `__dict__`.

Note: `__get__()` will always be called to retrieve the cached value since this is a so-called "data descriptor". This *might* be a performance issue in some scenarios due to extra lookups and method calls. To bypass the descriptor in cases where this might be a concern, one option is to store the cached value in a local variable.

The property can be set and deleted as usual. When the property is deleted, its value will be recomputed and reset on the next access.

It's safe to `del` a property that hasn't been set–this won't raise an attribute error as might be expected since a cached property can't really be deleted (since it will be recomputed the next time it's accessed).

A cached property can specify its dependencies (other cached properties) so that when its dependencies are set or deleted, the cached property will be cleared and recomputed on next access:

```
>>> class T:
...
...     @cached_property
...     def a(self):
...         return 'a'
...
...     @cached_property('a')
...     def b(self):
...         return '%s + b' % self.a
...
...
>>> t = T()
>>> t.a
'a'
>>> t.b
'a + b'
>>> t.a = 'A'
>>> t.b
'A + b'
```

When a property has been set directly (as opposed to via access), it won't be reset when its dependencies are set or deleted. If the property is later cleared, it will then be recomputed:

```
>>> t = T()
>>> t.b = 'B'  # set t.b directly
>>> t.b
'B'
>>> t.a = 'A'
>>> t.b  # t.b was set directly, so setting t.a doesn't affect it
'B'
>>> del t.b
>>> t.b  # t.b was cleared, so it's computed from t.a
'A + b'
```

classmethod reset_dependents_of(*obj*, *name*, ***, *_lock=<unlocked _thread.lock object>*, *_fake_props={}*)

Reset dependents of `obj.name`.

This is intended for use in overridden `__setattr__` and `__delattr__` methods for resetting cached properties that are dependent on regular attributes.

`tangled.decorators.`**fire_actions**(*where*, *tags=()*, *args=()*, *kwargs=None*, *_registry={}*)

Fire actions previously registered via *`register_action()`*.

`where` is typically a package or module. Only actions registered in that package or module will be fired.

`where` can also be some other type of object, such as a class, in which case only the actions registered on the class and its methods will be fired. Currently, this is considered non-standard usage, but it's useful for testing.

If no `tags` are specified, all registered actions under `where` will be fired.

`*args` and `**kwargs` will be passed to each action that is fired.

`tangled.decorators.`**per_instance_lru_cache**(*maxsize=128*, *typed=False*)

Least-recently-used cache decorator for methods and properties.

This is based on `functools.lru_cache()` in the Python standard library and mimics its API and behavior. The major difference is that this decorator creates a per-instance cache for the decorated method instead of a cache shared by all instances.

When `functools.lru_cache()` is used on a method, the cache for the method is shared between *all* instances. This means that clearing the LRU cache for a method clears it for all instances and that hit/miss info is an aggregate of calls from all instances.

This is intended for use with instance methods and properties. For class and static methods, `functools.lru_cache()` should work fine, since the issues noted above aren't applicable.

As with `functools.lru_cache()`, the arguments passed to wrapped methods must be hashable.

Args:

maxsize (int):

- If positive, LRU-caching will be enabled and the cache can grow up to the specified size, after which the least recently used item will be dropped.
- If `None`, the LRU functionality will be disabled and the cache can grow without bound.
- If 0, caching will be disabled and this will effectively just keep track of how many times a method is called per instance.
- A negative value is effectively the same as passing 1.

typed (bool): Whether arguments with different types will be cached separately. E.g., 1 and 1.0 both hash to 1, so `self.method(1)` and `self.method(1.0)` will result in the same key being generated by default.

Example:

```
>>> class C:
...
...     @per_instance_lru_cache()
...     def some_method(self, x, y, z):
...         return x + y + z
...
...     @property
...     @per_instance_lru_cache(1)
...     def some_property(self):
...         result = 2 ** 1000000
...         return result

>>> c = C()
>>> c.some_method(1, 2, 3)
6
>>> C.some_method.cache_info(c)
CacheInfo(hits=0, misses=1, maxsize=128, currsize=1)
>>> c.some_method(1, 2, 3)
6
>>> C.some_method.cache_info(c)
CacheInfo(hits=1, misses=1, maxsize=128, currsize=1)

>>> d = C()
>>> d.some_method(1, 2, 3)
6
>>> d.some_method(4, 5, 6)
15
>>> d.some_method(4, 5, 6)
15
>>> C.some_method.cache_info(d)
CacheInfo(hits=1, misses=2, maxsize=128, currsize=2)
>>> C.some_method.cache_clear(d)
>>> C.some_method.cache_info(d)
```

```
CacheInfo(hits=0, misses=0, maxsize=128, currsize=0)

>>> C.some_method.cache_info(c)  # Unaffected by instance d
CacheInfo(hits=1, misses=1, maxsize=128, currsize=1)

>>> c.some_property
9900...
>>> C.some_property.fget.cache_info(c)
CacheInfo(hits=0, misses=1, maxsize=1, currsize=1)
>>> c.some_property
9900...
>>> C.some_property.fget.cache_info(c)
CacheInfo(hits=1, misses=1, maxsize=1, currsize=1)
```

tangled.decorators.**register_action**(*wrapped*, *action*, *tag=None*, *_registry={}*)

Register a deferred decorator action.

The action will be performed later when *fire_actions()* is called with the specified `tag`.

This is used like so:

```
# mymodule.py

def my_decorator(wrapped):
    def action(some_arg):
        # do something with some_arg
    register_action(wrapped, action, tag='x')
    return wrapped  # <-- IMPORTANT

@my_decorator
def my_func():
    # do some stuff
```

Later, *fire_actions()* can be called to run `action`:

```
fire_actions(mymodule, tags='x', args=('some arg'))
```

2.1.3 Registry

class tangled.registry.**Registry**

A component registry.

get_all(*key*, *default=None*, *as_dict=False*)

Return all components for `key` in registration order.

2.1.4 Scripts

tangled.__main__.**main**(*argv=None*)

Entry point for running a tangled command.

Such commands are registered via the `tangled.scripts` entry point like so:

```
[tangled.scripts]
mycommand = mypackage.mymodule:MyCommand
```

The command can be run as `tangled mycommand ...`.

2.1.5 Settings

tangled.settings.**check_required**(*settings*, *required*)
Ensure `settings` contains the `required` keys.

tangled.settings.**get_type**(*name: str*)
Get the type corresponding to `name`.

tangled.settings.**parse_settings**(*settings*, *defaults={}*, *required=()*, *extra={}*, *prefix=None*, *strip_prefix=True*)
Convert settings values.

All settings values should be JSON-encoded strings. For example:

```
debug = true
factory:object = "tangled.web:Application"
something:package.module:SomeClass = "value"
```

Settings passed via `defaults` will be added if they're not already present in `settings`.

To convert only a subset of the settings, pass `prefix`; only the settings with a key matching `prefix` will be returned (see `get_items_with_key_prefix()` for details).

Required fields can be listed in `required`. If any required fields are missing, a `ValueError` will be raised.

For each setting:

- If the key specifies a type using `key:type` syntax, the specified type will be used to parse the value. The type can refer to any callable that accepts a single string.

  If the type is specified as `object`, *`load_object()`* will be used to parse the value.

  The `:type` will be stripped from the key.

- Otherwise, the value will be passed to `json.loads()`.

The original `settings` dict will not be modified.

tangled.settings.**parse_settings_file**(*path*, *section='app'*, *interpolation=None*, *meta_settings=True*, ***kwargs*)
Parse settings from the .ini file at `path`.

`path` can be a file system path or an asset path. `section` specifies which [section] to get settings from.

By default, some extra metadata will be added to the settings parsed from a file. These are the file name the settings were loaded from (__file__), the base file names of any extended files (__base__ and __bases__), and the environment indicated by the file's base name (env). Use `meta_settings=False` to disable this.

`kwargs` are the keyword args for *`parse_settings()`*.

2.1.6 Utilities

tangled.util.**NOT_SET = NOT_SET**
A `None`-ish constant for use where `None` may be a valid value.

tangled.util.**as_bool**(*value*)
Convert value to bool.

tangled.util.**filter_items**(*items*, *include=<function <lambda>>*, *exclude=<function <lambda>>*, *processors=()*)
Filter and optionally process `items`; yield pairs.

`items` can be any object with a `.items()` method that returns a sequence of pairs (e.g., a dict), or it can be a sequence of pairs (e.g., a list of 2-item tuples).

Each item will be passed to `include` and then to `exclude`; they must return `True` and `False` respectively for the item to be yielded.

If there are any `processors`, each included item will be passed to each processor in turn.

`tangled.util.`**`get_items_with_key_prefix`**(*items*, *prefix*, *strip_prefix=True*, *processors=()*)

Filter `items` to those with a key that starts with `prefix`.

`items` is typically a dict but can also be a sequence. See *`filter_items()`* for more on that.

`tangled.util.`**`load_object`**(*obj*, *obj_name=None*, *package=None*, *level=2*)

Load an object.

`obj` may be an object or a string that points to an object. If it's a string, the object will be loaded and returned from the specified path. If it's any other type of object, it will be returned as is.

The format of a path string is either 'package.module' to load a module or 'package.module:object' to load an object from a module.

The object name can be passed via `obj_name` instead of in the path (if the name is passed both ways, the name in the path will win).

Examples:

```
>>> load_object('tangled.util:load_object')
<function load_object at ...>
>>> load_object('tangled.util', 'load_object')
<function load_object at ...>
>>> load_object('tangled.util:load_object', 'IGNORED')
<function load_object at ...>
>>> load_object('.util:load_object', package='tangled')
<function load_object at ...>
>>> load_object('.:load_object', package='tangled.util')
<function load_object at ...>
>>> load_object(':load_object', package='tangled.util')
<function load_object at ...>
>>> load_object(load_object)
<function load_object at ...>
>>> load_object(load_object, 'IGNORED', 'IGNORED', 'IGNORED')
<function load_object at ...>
```

`tangled.util.`**`abs_path`**(*path*)

Get abs. path for `path`.

`path` may be a relative or absolute file system path or an asset path. If `path` is already an abs. path, it will be returned as is. Otherwise, it will be converted into a normalized abs. path.

`tangled.util.`**`asset_path`**(*path*, **rel_path*)

Get absolute path to asset in package.

`path` can be just a package name like 'tangled.web' or it can be a package name and a relative file system path like 'tangled.web:some/path'.

If `rel_path` is passed, it will be appended to the base rel. path in `path`.

Examples:

```
>>> asset_path('tangled.util')
'.../tangled/tangled'
>>> asset_path('tangled.util:')
```

```
'.../tangled/tangled'
>>> asset_path('tangled.util:x')
'.../tangled/tangled/x'
>>> asset_path('tangled.util', 'x')
'.../tangled/tangled/x'
>>> asset_path('tangled.util:x', 'y')
'.../tangled/tangled/x/y'
```

tangled.util.**fully_qualified_name**(*obj*)

Get the fully qualified name for an object.

Returns the object's module name joined with its qualified name. If the object is a module, its name is returned.

```
>>> fully_qualified_name(object)
'builtins.object'
>>> import tangled.util
>>> fully_qualified_name(tangled.util)
'tangled.util'
```

tangled.util.**is_asset_path**(*path*) → bool

Is `path` an asset path like `package.module:path`?

If `path` is absolute, it will not be considered an asset path. Otherwise, it will be considered an asset path if it contains a colon *and* the module path contains only valid Python identifiers. The file system path to the right of the colon can be empty or any string (it's ignored here).

Examples:

```
>>> is_asset_path('/some/abs/path')
False
>>> is_asset_path('rel/path')
False
>>> is_asset_path('package')
False
>>> is_asset_path('package:')
True
>>> is_asset_path('package.subpackage:rel/path')
True
>>> is_asset_path('package.subpackage:')
True
>>> is_asset_path('package.subpackage:rel/path')
True
>>> is_asset_path('base.ini')
False
```

tangled.util.**is_module_path**(*path*) → bool

Is `path` a module path like `package.module`?

Examples:

```
>>> is_module_path('package')
True
>>> is_module_path('package.module')
True
>>> is_module_path('.module')
True
>>> is_module_path('package.module:obj')
False
>>> is_module_path('a/b')
```

```
False
>>> is_module_path('/a/b')
False
```

tangled.util.**is_object_path**(*path*) → bool

Is `path` an object path like `package.module:obj.path`?

Examples:

```
>>> is_object_path('package.module:obj')
True
>>> is_object_path('.module:obj')
True
>>> is_object_path('package')
False
>>> is_object_path('package:')
False
>>> is_object_path('a/b')
False
>>> is_object_path('/a/b')
False
```

tangled.util.**constant_time_compare**(*a*, *b*)

Compare two bytes or str objects in constant time.

`a` and `b` must be either both bytes OR both strings w/ only ASCII chars.

Returns `False` if `a` and `b` have different lengths, if either is a string with non-ASCII characters, or their types don't match.

See `hmac.compare_digest()` for more details.

tangled.util.**random_bytes**(*n=16*, *as_hex=True*)

Return a random string of bytes.

By default, this will encode 16 random bytes as a 32-character byte string of hex digits (i.e., each byte is split into 4 bits and encoded as a hex digit).

In general, whenever `as_hex` is True, the number of bytes returned will be `2 * n`.

```
>>> len(random_bytes()) == 32
True
>>> len(random_bytes(10, as_hex=True)) == 20
True
>>> len(random_bytes(7, as_hex=False)) == 7
True
>>> random_bytes().__class__ is bytes
True
>>> random_bytes(as_hex=False).__class__ is bytes
True
```

tangled.util.**random_string**(*n=32*, *alphabet='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ012345*
_', *encoding='ascii'*) → str

Return a random string with length `n`.

By default, the string will contain 32 characters from the URL-safe base 64 alphabet.

`encoding` is used only if the `alphabet` is a byte string.

```
>>> len(random_string()) == 32
True
```

```
>>> len(random_string(8)) == 8
True
>>> len(random_string(7, ASCII_ALPHANUMERIC)) == 7
True
>>> random_string().__class__ is str
True
>>> random_string(alphabet=HEX).__class__ is str
True
>>> 'g' not in random_string(alphabet=HEX)
True
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index