pytool

Release 3.15.0

Oct 23, 2020

Contents

Pytool is available on PyPI: http://pypi.python.org/pypi/pytool/

Pytool is compatible with Python 2 (tested against 2.7) or Python 3.3 and higher. *Compatibility added August 4, 2015.*

Pytool's source is hosted on Github: http://github.com/shakefu/pytool

Any comments, issues or requests should be submitted via Github: https://github.com/shakefu/pytool/issues

CHAPTER 1

Contributors

- shakefu (creator, maintainer)
- dshen109
- abendig

1.1 pytool: Python helper package

This package contains a lot of helpful little methods and functions to make your life a little easier and your day a little better. Enjoy!

If you would like to see something added here, feel free to open up a GitHub issue.

Contents

        * hashed_singleton()
        * singleton()
        * unflatten()
    - pytool.text: *Text helpers*
        * wrap()
        * columns()
    - pytool.time: *Time and Date related helpers*
        * Timer
        * UTC
        * as_utc()
        * utcnow()
        * fromutctimestamp()
        * toutctimestamp()
        * is_dst()
        * trim_time()
        * floor_day()
        * floor_minute()
        * floor_month()
        * floor_week()
        * make_week_seconds()
        * week_seconds()
        * week_seconds_to_datetime()
        * week_start()
        * ago()
    - pytool.proxy: *DictProxy and ListProxy classes*
        * DictProxy
        * ListProxy

1.1.1 pytool.cmd: Command helpers

This module contains helpers related to writing scripts and creating command line utilities.

Command helpers

- pytool.cmd.Command

1.1.1.1 pytool.cmd.Command

class pytool.cmd.**Command**

Base class for creating commands that can be run easily as scripts. This class is designed to be used with the `console_scripts` entry point to create Python-based commands for your packages.

New in version 3.11.0: If the configargparse library is installed, Pytool will automatically use that as a drop-in replacement for the stdlib argparse module that is used by default.

You should use *`parser_opts()`* to give additional configuration arguments if you want to enable *configargparse* features like automatically using environment variables.

Hello world example:

```
# hello.py
from pytool.cmd import Command

class HelloWorld(Command):
    def run(self):
        print "Hello World."
```

The only thing that *must* be defined in the subclass is the *`run()`* method, which should contain the code to launch your application, all other methods are optional.

Example setup.py:

```
# setup.py
from setuptools import setup

setup(
    # ...
    entry_points={
        'console_scripts':[
            'helloworld = hello:HelloWorld.console_script',
            ],
        },
    )
```

When using an entry point script, the *`Command`* has a special *`console_script()`* method for launching the application.

Starting without an entry point script:

```
# hello.py [cont'd]
if __name__ == '__main__':
    import sys
    HelloWorld().start(sys.argv[1:])
```

The *`start()`* method always requires an argument - even if it's just an empty list.

More complex example:

```
from pytool.cmd import Command

class HelloAll(Command):
    def set_opts(self):
        self.opt('--world', default='World', help="use a different "
                 "world")
        self.opt('--verbose', '-v', action='store_true', help="use "
```

(continues on next page)

(continued from previous page)

```
                "more verbose output")

    def run(self):
        print "Hello", self.args.world

        if self.args.verbose:
            print "Hola", self.args.world
```

Whenever there are arguments for a command, they're made available for your use as `self.args`. This object is created by `argparse` so refer to that documentation for more information.

classmethod console_script()
Method used to start the command when launched from a distutils console script.

describe(*description*)
Describe the command in more detail. This will be displayed in addition to the argument help.

This automatically strips leading indentation but does not strip all formatting like the `ArgumentParser(description='')` keyword.

Example:

```
class MyCommand(Command):
    def set_opts(self):
        self.describe("""
            This is an example command. To use the example command,
            run it.""")

    def run(self):
        pass
```

opt(**args*, ***kwargs*)
Add an option to this command. This takes the same arguments as `ArgumentParser.add_argument()`.

parser_opts()
Subclasses should override this method to return a dictionary of additional arguments to the parser instance.

Example:

```
class MyCommand(Command):
    def parser_opts(self):
        return dict(
                description="Manual description for cmd.",
                add_env_var_help=True,
                auto_env_var_prefix='mycmd_',
                )
```

reload()
Reloads pyconfig if it is available.

Override this in your subclass if you wish to implement different reloading behavior.

run()
Subclasses should override this method to start the command process. In other words, this is where the magic happens.

Changed in version 3.15.0: By default, this will just print help and exit.

set_opts()
Subclasses should override this method to configure the command line arguments and options.

Example:

```
class MyCommand(Command):
    def set_opts(self):
        self.opt('--verbose', '-v', action='store_true',
                 help="be more verbose")

    def run(self):
        if self.args.verbose:
            print "I'm verbose."
```

start(*args*)
Starts a command and registers single handlers.

stop(**args*, ***kwargs*)
Exits the currently running process with status *0*.

Override this in your subclass if you wish to implement different SIGINT or SIGTERM handling for your process.

subcommand(*name*, *opt_func=None*, *run_func=None*, **args*, ***kwargs*)
Add a subcommand *name* with setup *opt_func* and main *run_func* to the argument parser.

Any additional positional or keyword arguments will be passed to the `ArgumentParser` instance created.

Changed in version 3.15.0: Either *opt_func* or *run_func* may be omitted, in which case a method with a name matching the subcommand name plus `_opts` will be bound to *opt_func* and a method matching the subcommand name will be bound to *run_func*.

For example, a subcommand `'write'` will bind the methods `write_opts()` and `write()`.

New in version 3.12.0.

Example:

```
class MyCommand(Command):
    def set_opts(self):
        self.subcommand('thing', self.thing, self.run_thing)

    def thing(self):
        # Set thing specific options
        self.opt('--thing', 't', help="Thing to do")

    def run_thing(self):
        # This runs if the thing subcommand is invoked
        pass

    def run(self):
        # This runs if there is no subcommand
```

Parameters

- **name** (*str*) – Subcommand name
- **opt_func** (*function*) – Options function to add
- **run_func** (*function*) – Run function to add

- **args** – Arguments to pass to the subparser constructor
- **kwargs** – Keyword arguments to pass to the subparser constructor

1.1.2 pytool.json: JSON helpers

This module contains helpers for working with JSON data.

Tries to use the *simplejson* module if it exists, otherwise falls back to the *json* module.

If the *bson* module exists, it allows *bson.ObjectId* objects to be decoded into JSON automatically.

JSON helpers

1.1.2.1 as_json()

pytool.json.**as_json**(*obj*, ***kwargs*)

Returns an object JSON encoded properly.

This method allows you to implement a hook method `for_json()` on your objects if you want to allow arbitrary objects to be encoded to JSON. A `for_json()` hook must return a basic JSON type (dict, list, int, float, string, unicode, float or None), or a basic JSON type which contains other objects which implement the `for_json()` hook.

If an object implements both `_asdict()` and `for_json()` the latter is given preference.

Also adds additional encoders for `datetime` and `bson.ObjectId`.

Parameters

- **obj** (*object*) – An object to encode.
- ****kwargs** – Any optional keyword arguments to pass to the JSONEncoder

Returns JSON encoded version of *obj*.

New in version 2.4: Objects which have an `_asdict()` method will have that method called as part of encoding to JSON, even when not using simplejson.

New in version 2.4: Objects which have a `for_json()` method will have that method called and the return value used for encoding instead.

Changed in version 3.0: simplejson (`>= 3.2.0`) is now required, and relied upon for the `_asdict()` and `for_json()` hooks. This change may break backwards compatibility in any code that uses these hooks.

1.1.2.2 from_json()

pytool.json.**from_json**(*value*)

Decodes a JSON string into an object.

Parameters **value** (*str*) – String to decode

Returns Decoded JSON object

1.1.3 pytool.lang: Language helpers

This module contains items that are "missing" from the Python standard library, that do miscelleneous things.

Language helpers

- Namespace
- UNSET
- classproperty()
- get_name()
- hashed_singleton()
- singleton()
- unflatten()

1.1.3.1 Namespace

class pytool.lang.**Namespace**(*obj=None*)

Namespace object used for creating arbitrary data spaces. This can be used to create nested namespace objects. It can represent itself as a dictionary of dot notation keys.

Basic usage:

```
>>> from pytool.lang import Namespace
>>> # Namespaces automatically nest
>>> myns = Namespace()
>>> myns.hello = 'world'
>>> myns.example.value = True
>>> # Namespaces can be converted to dictionaries
>>> myns.as_dict()
{'hello': 'world', 'example.value': True}
>>> # Namespaces have container syntax
>>> 'hello' in myns
True
>>> 'example.value' in myns
True
>>> 'example.banana' in myns
False
>>> 'example' in myns
True
>>> # Namespaces are iterable
>>> for name, value in myns:
...     print name, value
...
hello world
example.value True
>>> # Namespaces that are empty evaluate as False
>>> bool(Namespace())
False
>>> bool(myns.notset)
```

(continues on next page)

(continued from previous page)

```
False
>>> bool(myns)
True
>>> # Namespaces allow the __get__ portion of the descriptor protocol
>>> # to work on instances (normally they would not)
>>> class MyDescriptor(object):
...     def __get__(self, instance, owner):
...         return 'Hello World'
...
>>> myns.descriptor = MyDescriptor()
>>> myns.descriptor
'Hello World'
>>> # Namespaces can be created from dictionaries
>>> newns = Namespace({'foo': {'bar': 1}})
>>> newns.foo.bar
1
>>> # Namespaces will expand dot-notation dictionaries
>>> dotns = Namespace({'foo.bar': 2})
>>> dotns.foo.bar
2
>>> # Namespaces will coerce list-like dictionaries into lists
>>> listns = Namespace({'listish': {'0': 'zero', '1': 'one'}})
>>> listns.listish
['zero', 'one']
>>> # Namespaces can be deepcopied
>>> a = Namespace({'foo': [[1, 2], 3]}
>>> b = a.copy()
>>> b.foo[0][0] = 9
>>> a.foo
[[1, 2], 3]
>>> b.foo
[[9, 2], 3]
>>> # You can access keys using dict-like syntax, which is useful
>>> myns.foo.bar = True
>>> myns['foo'].bar
True
>>> # Dict-like access lets you traverse namespaces
>>> myns['foo.bar']
True
>>> # Dict-like access lets you traverse lists as well
>>> listns['listish.0']
'zero'
>>> listns['listish.1']
'one'
>>> # Dict-like access lets you traverse nested lists and namespaces
>>> nested = Namespace()
>>> nested.values = []
>>> nested.values.append(Namespace({'foo': 'bar'}))
>>> nested['values.0.foo']
'bar'
>>> # You can also call the traversal method if you need
>>> nested.traversal(['values', 0, 'foo'])
'bar'
```

Namespaces are useful!

New in version 3.5.0: Added the ability to create Namespace instances from dictionaries.

New in version 3.6.0: Added the ability to handle dot-notation keys and list-like dicts.

New in version 3.7.0: Added deepcopy capability to Namespaces.

New in version 3.8.0: Added dict-like access capability to Namespaces.

New in version 3.9.0: Added traversal by key/index arrays for nested Namespaces and lists

as_dict(*base_name=None*)
Return the current namespace as a dictionary.

Parameters **base_name** (*str*) – Base namespace (optional)

copy(**args*, ***kwargs*)
Return a copy of a Namespace by writing it to a dict and then writing back to a Namespace.

Arguments to this method are ignored.

from_dict(*obj*)
Populate this Namespace from the given *obj* dictionary.

Parameters **obj** (*dict*) – Dictionary object to merge into this Namespace

New in version 3.5.0.

items(*base_name=None*)
Return generator which returns `(key, value)` tuples.

Analagous to dict.items() behavior in Python3

Parameters **base_name** (*str*) – Base namespace (optional)

iteritems(*base_name=None*)
Return generator which returns `(key, value)` tuples.

Parameters **base_name** (*str*) – Base namespace (optional)

traverse(*path*)
Traverse the Namespace and any nested elements by following the elements in an iterable *path* and return the item found at the end of *path*.

Traversal is achieved using the __getitem__ method, allowing for traversal of nested structures such as arrays and dictionaries.

AttributeError is raised if one of the attributes in *path* does not exist at the expected depth.

Parameters **path** (*iterable*) – An iterable whose elements specify the keys to path over.

Example usage:

```
ns = Namespace({"foo":
                [Namespace({"name": "john"}),
                Namespace({"name": "jane"})]})
ns.traverse(["foo", 1, "name"])  # Returns "jane"
```

1.1.3.2 UNSET

class pytool.lang.**UNSET**
Special class that evaluates to `bool(False)`, but can be distinctly identified as seperate from `None` or `False`. This class can and should be used without instantiation.

```
>>> from pytool.lang import UNSET
>>> # Evaluates to False
>>> bool(UNSET)
False
>>> # Is a class-singleton (cannot become an instance)
>>> UNSET() is UNSET
True
>>> # Is good for checking default values
>>> if {}.get('example', UNSET) is UNSET:
...     print "Key is missing."
...
Key is missing.
>>> # Has no length
>>> len(UNSET)
0
>>> # Is iterable, but has no iterations
>>> list(UNSET)
[]
>>> # It has a repr() equal to itself
>>> UNSET
UNSET
```

1.1.3.3 classproperty()

pytool.lang.**classproperty**(*func*)

Makes a `@classmethod` style property (since `@property` only works on instances).

```
from pytool.lang import classproperty

class MyClass(object):
    _attr = 'Hello World'

    @classproperty
    def attr(cls):
        return cls._attr

MyClass.attr # 'Hello World'
MyClass().attr # Still 'Hello World'
```

1.1.3.4 get_name()

pytool.lang.**get_name**(*frame*)

Gets the name of the passed frame.

Warning It's very important to delete a stack frame after you're done using it, as it can cause circular references that prevents garbage collection.

Parameters **frame** – Stack frame to inspect.

Returns Name of the frame in the form *module.class.method*.

1.1.3.5 hashed_singleton()

pytool.lang.**hashed_singleton**(*klass*)

Wraps a class to create a hashed singleton version of it. A hashed singleton is like a singleton in that there will

be only a single instance of the class for each call signature.

The singleton is kept as a weak reference, so if your program ceases to reference the hashed singleton, you may get a new instance if the Python interpreter has garbage collected your original instance.

This will not work for classes that take arguments that are unhashable (e.g. dicts, sets).

Parameters **klass** – Class to decorate

New in version 2.1.

Changed in version 3.4.2: *@hashed_singleton* wrapped classes now preserve their *@staticmethod* functions on the class type as well as the instance.

Example usage:

```
# Make a class directly behave as a hashed singleton
@hashed_singleton
class Test(object):
    def __init__(self, *args, **kwargs):
        pass

# Make an imported class behave as a hashed singleton
Test = hashed_singleton(Test)

# The same arguments give you the same class instance back
test = Test('a', k='k')
test is Test('a', k='k') # True

# A different argument signature will give you a new instance
test is Test('b', k='k') # False
test is Test('a', k='j') # False

# Removing all references to a hashed singleton instance will allow
# it to be garbage collected like normal, because it's only kept
# as a weak reference
del test
test = Test('a', k='k') # If the Python interpreter has garbage
                        # collected, you will get a new instance
```

1.1.3.6 singleton()

pytool.lang.**singleton**(*klass*)

Wraps a class to create a singleton version of it.

Parameters **klass** – Class to decorate

Changed in version 3.4.2: *@singleton* wrapped classes now preserve their *@staticmethod* functions on the class type as well as the instance.

Example usage:

```
# Make a class directly behave as a singleton
@singleton
class Test(object):
    pass

# Make an imported class behave as a singleton
Test = singleton(Test)
```

1.1.3.7 unflatten()

pytool.lang.**unflatten**(*obj*)

Return *obj* with dot-notation keys unflattened into nested dictionaries, as well as list-like dictionaries converted into list instances.

Parameters **obj** – An arbitrary object, preferably a dict

1.1.4 pytool.text: Text helpers

This module contains text related things that make life easier.

Text helpers

- wrap()
- columns()

1.1.4.1 wrap()

pytool.text.**wrap**(*text*, *width=None*, *indent=''*)

Return *text* wrapped to *width* while trimming leading indentation and preserving paragraphs.

This function is handy for turning indented inline strings into unindented strings that preserve paragraphs, whitespace, and any indentation beyond the baseline.

The default wrap width is the number of columns available in the current console or TTY. If the columns can't be found, then the default wrap width is 79.

Parameters

- **text** (*str*) – Text to wrap
- **width** (*int*) – Width to wrap text at (default: text column width)
- **indent** (*str*) – String to indent text with (default: '')

```
>>> import pytool
>>> text = '''
        All this is indented by 8, but will be 0.
                This is indented by 16, and a really long long long
                line which is hard wrapped at a random character width,
                but will be wrapped appropriately at 70 chars
                afterwards.

        This is indented by 8 again.
    '''
>>> print pytool.text.wrap(text)
All this is indented by 8, but will be 0.
        This is indented by 16, and a really long long long line which
        is hard wrapped at a random character width, but will be
        wrapped appropriately at 70 chars afterwards.

This is indented by 8 again.
>>>
```

1.1.4.2 columns()

pytool.text.**columns**(*default=79*)
Return the number of text columns for the current console or TTY.

On certain systems it may be necessary to set the COLUMNS environment variable if you want this method to work correctly.

Uses *default* (79) if no value can be found.

1.1.5 pytool.time: Time and Date related helpers

This module contains time related things that make life easier.

Time and Date related helpers

- Timer
- UTC
- as_utc()
- utcnow()
- fromutctimestamp()
- toutctimestamp()
- is_dst()
- trim_time()
- floor_day()
- floor_minute()
- floor_month()
- floor_week()
- make_week_seconds()
- week_seconds()
- week_seconds_to_datetime()
- week_start()
- ago()

1.1.5.1 Timer

class pytool.time.**Timer**
This is a simple timer class.

```
timer = pytool.time.Timer()
for i in (1, 2, 3):
    sleep(i)
    print timer.mark(), "elapsed since last mark or start"
print timer.elapsed, "total elapsed"
```

elapsed
Return a `timedelta` of the time elapsed since the start.

mark()
Return a `timedelta` of the time elapsed since the last mark or start.

1.1.5.2 UTC

class pytool.time.**UTC**
UTC timezone. This is necessary since Python doesn't include any explicit timezone objects in the standard library. This can be used to create timezone-aware datetime objects, which are a pain to work with, but a necessary evil sometimes.

```
from datetime import datetime
from pytool.time import UTC

utc_now = datetime.now(UTC())
```

1.1.5.3 as_utc()

pytool.time.**as_utc**(*stamp*)
Converts any datetime (naive or aware) to UTC time.

Parameters **stamp** (*datetime*) – Datetime to convert

Returns *stamp* as UTC time

```
from datetime import datetime
from pytool.time import as_utc

utc_datetime = as_utc(datetime.now())
```

1.1.5.4 utcnow()

pytool.time.**utcnow**()
Return the current UTC time as a timezone-aware datetime.

Returns The current UTC time

1.1.5.5 fromutctimestamp()

pytool.time.**fromutctimestamp**(*stamp*)
Return a timezone-aware datetime object from a UTC unix timestamp.

Parameters **stamp** (*float*) – Unix timestamp in UTC

Returns UTC datetime object

```
import time
from pytool.time import fromutctimestamp

utc_datetime = fromutctimestamp(time.time())
```

1.1.5.6 toutctimestamp()

pytool.time.**toutctimestamp**(*stamp*)

Converts a naive datetime object to a UTC unix timestamp. This has an advantage over *time.mktime* in that it preserves the decimal portion of the timestamp when converting.

Parameters **stamp** (*datetime*) – Datetime to convert

Returns Unix timestamp as a `float`

```
from datetime import datetime
from pytool.time import toutctimestamp

utc_stamp = toutctimestamp(datetime.now())
```

1.1.5.7 is_dst()

pytool.time.**is_dst**(*stamp*)

Return `True` if *stamp* is daylight savings.

Parameters **stamp** (*datetime*) – Datetime

Returns `True` if *stamp* is daylight savings, otherwise `False`.

1.1.5.8 trim_time()

pytool.time.**trim_time**(*stamp*)

Trims the time portion off of *stamp*, leaving the date intact. Returns a datetime of the same date, set to 00:00:00 hours. Preserves timezone information.

Parameters **stamp** (*datetime*) – Timestamp to trim

Returns Trimmed timestamp

1.1.5.9 floor_day()

pytool.time.**floor_day**(*stamp=None*)

Return *stamp* floored to the current day. If no *stamp* is specified, the current time is used. This is similar to the `date()` method, but returns a `datetime` object, instead of a `date` object.

This is the same as *trim_time()*.

Changed in version 2.0: Preserves timezone information if it exists, and uses *pytool.time.utcnow()* instead of `datetime.datetime.now()` if *stamp* is not given.

Parameters **stamp** (*datetime*) – *datetime* object to floor (default: now)

Returns Datetime floored to the day

1.1.5.10 floor_minute()

pytool.time.**floor_minute**(*stamp=None*)

Return *stamp* floored to the current minute. If no *stamp* is specified, the current time is used. Preserves timezone information.

New in version 2.0.

Parameters **stamp** (*datetime*) – *datetime* object to floor (default: now)

Returns Datetime floored to the minute

1.1.5.11 floor_month()

pytool.time.**floor_month**(*stamp=None*)

Return *stamp* floored to the current month. If no *stamp* is specified, the current time is used.

Changed in version 2.0: Preserves timezone information if it exists, and uses *pytool.time.utcnow()* instead of `datetime.datetime.now()` if *stamp* is not given.

Parameters **stamp** (*datetime*) – *datetime* object to floor (default: now)

Returns Datetime floored to the month

1.1.5.12 floor_week()

pytool.time.**floor_week**(*stamp=None*)

Return *stamp* floored to the current week, at 00:00 Monday. If no *stamp* is specified, the current time is used. Preserves timezone information.

This is the same as *week_start()*

New in version 2.0.

Parameters **stamp** (*datetime*) – *datetime* object to floor (default now:)

Returns Datetime floored to the week

1.1.5.13 make_week_seconds()

pytool.time.**make_week_seconds**(*day*, *hour*, *minute=0*, *seconds=0*)

Return *week_seconds()* for the given *day* of the week, *hour* and *minute*.

Parameters

- **day** (*int*) – Zero-indexed day of the week
- **hour** (*int*) – Zero-indexed 24-hour
- **minute** (*int*) – Minute (default: 0)
- **seconds** (*int*) – Seconds (default: 0)

Returns Seconds since 00:00 Monday

1.1.5.14 week_seconds()

pytool.time.**week_seconds**(*stamp*)

Return *stamp* converted to seconds since 00:00 Monday.

Parameters **stamp** (*datetime*) – Timestamp to convert

Returns Seconds since 00:00 monday

1.1.5.15 week_seconds_to_datetime()

pytool.time.**week_seconds_to_datetime**(*seconds*)
Return the datetime that is *seconds* from the start of this week.

Parameters **seconds** (*int*) – Seconds

Returns Datetime for 00:00 Monday plus *seconds*

1.1.5.16 week_start()

pytool.time.**week_start**(*stamp*)
Return the start of the week containing *stamp*.

Changed in version 2.0: Preserves timezone information.

Parameters **stamp** (*datetime*) – Timestamp

Returns A datetime for 00:00 Monday of the given week

1.1.5.17 ago()

pytool.time.**ago**(*stamp=None*, ***kwargs*)
Return the current time as UTC minutes the specified timeframe.

This is a helper for simplifying the common pattern of *pytool.time.utcnow() - datetime.timedelta(mintues=15)*.

Parameters

- **stamp** – An optional timestamp instead of *utcnow()*
- **days** – Days previous
- **hours** – Hours previous
- **minutes** – Minutes previous
- **seconds** – Days previous

Returns UTC timestamp

If you like brevity in your arguments, you can use the shorter versions, *hrs=*, *mins=* and *secs=*.

Or if you really want a short signature, you can use *d=*, *h=*, *m=*, *s=*.

```
import pytool

yesterday = pytool.time.ago(days=1)

a_little_while_ago = pytool.time.ago(minutes=1)

a_little_before = pytool.time.ago(my_date, hours=1)

# Shorter arguments
shorter = pytool.time.ago(hrs=1, mins=1, secs=1)

# Shorthand argument names
short = pytool.time.ago(d=1, h=1, m=1, s=1)
```

1.1.6 pytool.proxy: DictProxy and ListProxy classes

This module contains implmentations of proxy-list and proxy-dictionary objects.

1.1.6.1 DictProxy

class pytool.proxy.**DictProxy**(*data*)

Proxies all methods for a dict instance.

This is useful when you want to modify a dictionary's behavior through subclassing without copying the dictionary or if you want to be able to modify the original dictionary.

Parameters **data** – A dict or dict-like object (implements all the `collections.MutableMapping` methods)

New in version 2.2.

Note If you intend to use a subclass which modifies the apparent keys or values of this class with *pytool.json.as_json()*, remember to override `for_json()` to produce the data you desire.

Example:

```
from pytool.proxy import DictProxy

class SquaredDict(DictProxy):
    def __getitem__(self, key):
        value = super(SquaredDict, self).__getitem__(key)
        if isinstance(value, int):
            value *= value
        return value

my_dict = {}
my_proxy = SquaredDict(my_dict)
my_proxy['val'] = 5

my_proxy['val'] # 25
my_dict['val'] # 5
```

clear() → None. Remove all items from D.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

1.1.6.2 ListProxy

class pytool.proxy.**ListProxy**(*data*)
Proxies all methods for a list instance. This is useful when you want to modify a list's behavior without copying the list.

Methods which do not mutate a list, and instead return a new list will return a *list* instance rather than a *ListProxy* instance.

Parameters **data** – A list or list-like object (implements all the `collections.MutableSequence` methods)

New in version 2.2.

Note If you intend to use a subclass which modifies the apparent indices or values of this class with *pytool.json.as_json()*, remember to override `for_json()` to produce the data you desire.

Example:

```
from pytool.proxy import ListProxy

class SquaredList(ListProxy):
    def __setitem__(self, index, value):
        if isinstance(value, int):
            value *= value
        super(SquaredList, self).__setitem__(index, value)

my_list = range(5)
my_proxy = SquaredList(my_list)
my_proxy[3] = 5

my_proxy[3] # 25
my_list[3] # 25
```

append(*item*)
S.append(object) – append object to the end of the sequence

count(*value*) → integer – return number of occurrences of value

extend(*other*)
S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*) → integer – return first index of value.
Raises ValueError if the value is not present.

insert(*i*, *item*)
S.insert(index, object) – insert object before index

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*item*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

1.2 Changelog

1.2.1 All releases

Release are now tracked **'on GitHub <https://github.com/shakefu/pytool/releases'_**. Please visit there to see changes from *3.4.2* and later.

1.2.2 Older releases

Here you'll find a record of the changes in each version of `pytool`.

1.2.2.1 3.4.2

- Preserves *@staticmethod* functions on `pytool.lang.singleton()` and `pytool.lang.hashed_singleton()` decorated classes.

Released March 19, 2018.

1.2.2.2 3.4.1

- Merges the tests and fix from PR #3. Thanks to abendig for the contribution.

Released August 4, 2015.

1.2.2.3 3.4.0

- Adds **Python 3** compatibility to Pytool! Hooray! Please submit an issue if you find any bugs in Python 3. Due to the dependency on *simplejson*, only Python 3.3 and later is supported.

Released August 4, 2015.

1.2.2.4 3.3.0

- Adds `pytool.time.ago()`, which is a convenient helper for getting times relative to a timestamp or the current time.

Released August 3, 2015.

1.2.2.5 3.2.0

- Adds *pytool.text* and *pytool.text.wrap()* which helps wrap text and remove or add indentation, and does so in a paragraph and whitespace aware fashion.
- Adds *pytool.cmd.Command.describe()* to make it easier to add verbose descriptions to your command's `--help`.

1.2.2.6 3.1.1

- Depend on canonical version of simplejson again instead of github fork.

1.2.2.7 3.1.0

- Add *pytool.time.Timer* for easy timing of things.

1.2.2.8 3.0.1

- Fix bug with setup.py which broke installs.

1.2.2.9 3.0.0

- Changed to depend on simplejson (`>=3.2.0`) for the `_asdict()` and `for_json()` hooks. This may break backwards compatability.

1.2.2.10 2.4.1

- Fix bug where `for_json()` hook was ignored on classes that subclass the basic types.
- Fix bug where *pytool.json.as_json()* would leave a trailing space on timestamps if there is no timezone associated with them.

1.2.2.11 2.4.0

- Improve documentation.
- Add `for_json()` hook in *pytool.json.as_json()*.
- Add `__repr__()` to *pytool.time.UTC* to make it prettier.
- Add support for `_asdict()` hook (implemented by `namedtuple`) even when not using `simplejson`.
- Fix *pytool.time.is_dst()* test.
- Add `for_json()` hook to *pytool.proxy.DictProxy* and *pytool.proxy.ListProxy*.

1.2.2.12 2.3.2

- Fix descriptor protocol in iteritems.

1.2.2.13 2.3.1

- Implement a instance-descriptor read-only protocol for *pytool.lang.Namespace* objects. This means you can assign descriptor instances to Namespace instances, and their values can be read, but not set.

  This differs from normal python descriptor behavior, where the descriptor instance must be present in the class rather than the instance.

1.2.2.14 2.3.0

- Make *pytool.lang.Namespace* instances evaluate as `False` when empty and cast as a `bool()`.

1.2.2.15 2.2.0

- Added *pytool.proxy.DictProxy* and *pytool.proxy.ListProxy*.

1.2.2.16 2.1.0

- Added *pytool.lang.hashed_singleton*.

1.2.2.17 2.0.1

- Update setup.py to include classifiers.

1.2.2.18 2.0.0

- Add *pytool.time.floor_minute()* and *pytool.time.floor_week()*.
- Change *pytool.time.floor_month()* and *pytool.time.floor_day()* to preserve timezone information.

1.2.2.19 Pre-2.0.0

Sorry, I was lazy and didn't keep a Changelog until 2.0. Apologies!

See the *Changelog* for a list of changes.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

R

S

T

U

V

W