tempvars Documentation

Release 1.0

Brian Skinn

Nov 14, 2018

Contents:

Streamlined temporary variable management in Jupyter Notebook, IPython, etc.

A frustrating aspect of working with Jupyter notebooks is debugging a worksheet for half an hour and discovering a carried-over variable name was hanging around in the notebook namespace and causing that cryptic misbehavior. Similarly, it's incredibly annoying to open a broken notebook that "worked fine" the last time it was used because of random variables lingering in the namespace.

The *TempVars* context manager helps to avoid these pitfalls by masking selected identifiers from the namespace for the duration of the with suite, then restoring them afterwards (or not, if desired). Further, any variables created within the managed context that match the criteria passed to *TempVars* are removed from the namespace upon exiting, ensuring these values do not spuriously contribute to following code. For convenience, all variables that were removed from the namespace at both entry and exit are stored with their values for later reference; see *Inspecting Masked Variables* and *Inspecting Discarded Temporary Variables*, respectively, in the usage instructions.

NOTE: Due to the way Python handles non-global variable scopes, *TempVars* can only be used at the global scope. *Any attempt to use TempVars in non-global contexts will result in a* `RuntimeError`. Viable use-cases include Jupyter notebooks, the IPython and basic Python REPLs, and the outermost scope of executed and imported modules. Preliminary testing indicates it also works with cauldron-notebook, though it may be less helpful there due to the step-local scoping paradigm used (shared values must be passed around via `cauldron.shared`).

NOTE ALSO that `tempvars` is *Python 3 only*.

Install with `pip install tempvars`, import as `from tempvars import TempVars`, and use as `with TempVars({pattern args}):`.

The project source is hosted on GitHub. Bug reports and feature requests are welcomed at the Issues page there. If you like the idea of an existing enhancement in the Issues list, please comment to say so; it'll help prioritization.

CHAPTER 1

tempvars Usage Examples

In all of these examples, it is assumed that `TempVars` has already been imported and that *foo* and *bar* have been defined as:

```
from tempvars import TempVars

foo = 1
bar = 2
```

The removal of a pre-existing variable from the namespace for the duration of a `with TempVars` context is termed **masking** here. Temporary variables created within the managed context that match one or more of *names*, *starts*, and/or *ends* are **discarded** (removed from the namespace) when exiting the context.

Note: The most common use case us anticipated to be via either *starts* or *ends*, where a common prefix or suffix, respectively (such as *t_* or *_t*), is used to mark all temporary variables within the managed context. See "*Masking Variables by Pattern*," below.

1.1 Table of Contents

1.1.1 Masking Specific Variables

The most basic usage is to supply individual variable names in the *names* argument:

```
>>> with TempVars(names=['foo', 'bar']):
...     print('foo' in dir())
...     print('bar' in dir())
...
False
False
>>> print(foo + bar)
3
```

Note: *names* must always be a list of strings, even when only one variable name is passed.

If a variable name passed to *names* doesn't exist in the namespace, *TempVars* silently ignores it when entering the with block. It **does**, however, still discard any matching variables from the namespace upon exiting the with block:

```
>>> with TempVars(names=['baz']):
...     print('foo' in dir())
...     print('bar' in dir())
...     print(2 * (foo + bar))
...     baz = 5
...     print(baz)
...
True
True
6
5
>>> print(2 * (foo + bar))
6
>>> 'baz' in dir()
False
```

1.1.2 Masking Variables by Pattern

Variables can also be masked by pattern matching. Currently, only 'starts with' and 'ends with' matching styles are supported:

```
>>> with TempVars(starts=['fo'], ends=['ar']):
...     print('foo' in dir())
...     print('bar' in dir())
...
False
False
>>> print(foo + bar)
3
```

Note: *starts* and *ends* must always be lists of strings, even when only one pattern is passed.

To avoid accidental masking of system variables, the *starts* argument cannot start with a double underscore:

```
>>> try:
...     with TempVars(starts=['__foo']):
...         pass
... except ValueError:
...     print('Argument rejected')
...
Argument rejected
```

Similarly, *ends* cannot end with a double underscore:

```
>>> try:
...     with TempVars(ends=['foo__']):
...         pass
... except ValueError:
...     print('Argument rejected')
...
Argument rejected
```

As well, neither *starts* nor *ends* can be a single underscore, since this also would mask Python system variables:

```
>>> try:
...     with TempVars(starts=['_']):
...         pass
... except ValueError:
...     print('Argument rejected')
...
Argument rejected
```

As with *names*, *starts* and *ends* also discard at exit any matching variables created within the with block, whether they existed previously or not:

```
>>> with TempVars(starts=['t_'], ends=['_t']):
...     t_foo = 6
...     bar_t = 7
...     print(t_foo * bar_t)
...
42
>>> 't_foo' in dir()
False
>>> 'bar_t' in dir()
False
```

1.1.3 Discarding Masked Variables

If desired, *TempVars* can be instructed not to restore any variables it masks from the original namespace, effectively discarding them permanently:

```
>>> with TempVars(names=['foo', 'bar'], restore=False):
...     pass
...
>>> 'foo' in dir()
False
>>> 'bar' in dir()
False
```

TempVars contexts can be freely nested to allow selective restore/discard behavior:

```
>>> with TempVars(names=['foo'], restore=False):
...     with TempVars(names=['bar']):
...         foo = 3
...         bar = 5
...         print(foo * bar)
...     print(foo * bar)
15
6
>>> print(bar)
2
>>> 'foo' in dir()
False
```

1.1.4 Binding TempVars Instances

TempVars is constructed so that each instance can be bound as part of the with statement, for later inspection within *and* after the managed context. The masking pattern arguments are stored without modification, but are duplicated from the input argument to avoid munging of mutable arguments:

```
>>> names_in = ['foo']
>>> with TempVars(names=names_in, starts=['baz', 'quux'],
...               ends=['ar']) as tv:
...     print(tv.starts)
...     print(tv.ends)
...     print(tv.names)
...     print('foo' in dir())
...     print('bar' in dir())
['baz', 'quux']
['ar']
['foo']
False
False
>>> names_in.append('quorz')
>>> print(tv.names)
['foo']
```

As shown above, these instance variables can also be examined after the end of the managed context.

1.1.5 Inspecting Masked Variables

TempVars provides a means to access the masked variables from within the managed context, via the *stored_nsvars* instance variable:

```
>>> with TempVars(names=['foo']) as tv:
...     print(list(tv.stored_nsvars.keys()))
...     print(tv.stored_nsvars['foo'])
...     print('foo' in dir())
['foo']
1
False
```

The masked variables remain available after the end of the managed context, even if they are not restored when the context exits:

```
>>> with TempVars(names=['foo']) as tv:
...     pass
>>> print(tv.stored_nsvars['foo'])
1
>>> with TempVars(names=['bar'], restore=False) as tv2:
...     pass
>>> print('bar' in dir())
False
>>> print(tv2.stored_nsvars['bar'])
2
```

A caveat: the masked variables are bound within *stored_nsvars* by simple assignment, which can have (possibly undesired) side effects when mutable objects are modified after being masked:

```
>>> baz = [1, 2, 3]
>>> with TempVars(names=['baz']) as tv:
...     tv.stored_nsvars['baz'].append(12)
>>> print(baz)
[1, 2, 3, 12]
>>> baz.remove(2)
>>> print(tv.stored_nsvars['baz'])
[1, 3, 12]
```

If `copy()` or `deepcopy()` behavior is of interest, please add a comment to that effect on the related GitHub issue.

1.1.6 Inspecting Discarded Temporary Variables

In an analogous fashion to *stored_nsvars*, the temporary variables discarded from the namespace at the exit of the managed context are stored in *retained_tempvars*:

```
>>> with TempVars(names=['foo']) as tv:
...     foo = 5
...     print(foo * bar)
10
>>> print(foo + tv.retained_tempvars['foo'])
6
```

Also as with *stored_nsvars*, at this time the values within *retained_tempvars* are bound by simple assignment, leading to similar possible side effects:

```
>>> baz = [1, 2]
>>> with TempVars(names=['baz']) as tv:
...     tv.stored_nsvars['baz'].append(3)
...     baz = tv.stored_nsvars['baz']
>>> tv.retained_tempvars['baz'].append(4)
>>> print(baz)
[1, 2, 3, 4]
```

As above, if `copy()` and/or `deepcopy()` behavior is of interest, please comment on the relevant GitHub issue.

CHAPTER 2

tempvars API

Base module of `tempvars` package.

class `tempvars.`**TempVars**(*names=None*, *starts=None*, *ends=None*, *restore=True*)

Context manager for handling temporary variables at the global scope.

WILL NOT WORK PROPERLY unless used as a context manager!!

CAN ONLY BE USED at global scopes (Python/IPython REPL, Jupyter notebook, etc.)

Parameters

- **names** – `list` of `str` - Variables will be treated as temporary if their names test equal to any of these items.
- **starts** – `list` of `str` - Variables will be treated as temporary if their names *start* with any of these patterns (tested with `.startswith(starts[i])`).
- **ends** – `list` of `str` - Variables will be treated as temporary if their names *end* with any of these patterns (tested with `.endswith(ends[i])`).
- **restore** – `bool` - If `True`, any variables hidden from the namespace upon entry into the with suite are restored to the namespace upon exit. If `False`, no variables are restored.

The *`TempVars`* instance can be bound in the with statement for access to stored variables, etc.:

```
>>> with TempVars(names=['abcd']) as tv:
...     pass
```

See the *usage examples* page for more information.

Class Members

These objects are accessible via the instance bound as part of the with statement (`tv` from the above snippet). All are constructed using `attr.ib()`.

names

`list` of `str` - All variable names passed to *names*.

starts
list of str - All passed .startswith matching patterns.

ends
list of str - All passed .endswith matching patterns.

restore
bool flag indicating whether to restore the prior namespace contents. **Can** be changed within the with suite.

stored_nsvars
dict container for preserving variables masked from the namespace, along with their associated values.

retained_tempvars
dict container for storing the temporary variables discarded from the namespace after exiting the with block.

Python Module Index

t

Index