undo-doc Documentation

Release 0.5.1

David Townshend

April 12, 2012

Contents

This is an undo/redo framework based on a functional approach which uses a undoable stack to track actions. Actions are the result of a function call and know how to undo and redo themselves, assuming that any objects they act on will always be in the same state before and after the action occurs respectively. The `Stack` tracks all actions which can be done or undone.

CHAPTER 1

Installation

undo can be installed using pip: `pip install undo`, or downloaded from http://bitbucket.org/aquavitae/undo and installed using `python setup.py install`. It has been tested with Python 2.7 and Python 3.2, and has no extra dependencies.

CHAPTER 2

Bug Reporting

Bug reports and feature requests can be made on the issue tracker.

CHAPTER 3

Basic Usage

The easiest way to use this framework is by defining actions using the `undoable` decorator on a generator object. This is a very similar syntax to that used by Pythons's `contextlib.contextmanager`. For example,

```
>>> @undoable
... def add(sequence, item):
...     # Do the action
...     sequence.append(item)
...     position = len(sequence) - 1
...     # Yield a string describing the action
...     yield "Add '{}' at psn '{}'".format(item, position)
...     # Undo the action
...     del sequence[position]
```

This defines a new action, *add*, which appends an item to a sequence. The resulting object is an factory which creates a new action instance and adds it to the stack.

```
>>> s = [1, 2, 3]
>>> add(s, 4)
>>> s
[1, 2, 3, 4]
>>> stack().undotext()
"Undo Add '4' at psn '3'"
>>> stack().undo()
>>> s
[1, 2, 3]
```

3.1 Return values and exceptions

The first call to the action may have a return value by adding it to the *yield* statement. However, it will be ignored in subsequent redos or undos.

```
>>> @undoable
... def process(obj):
...     obj[0] += 1
...     yield 'Process', obj
...     obj[0] -=1
...
>>> obj = [1, 2]
>>> process(obj)
[2, 2]
>>> print(obj)
[2, 2]
```

```
>>> stack().undo()
>>> print(obj)
[1, 2]
```

If an exception is raised during the action, it is not added to the stack and the exception is propagated. If an exception is raised during a redo or undo operation, the exception is propagated and the stack is cleared.

3.2 Nested actions

It is safe for actions to call each other. Only the top-most action is added to the stack.

```
>>> @undoable
... def add(seq, item):
...     seq.append(item)
...     yield 'Add'
...     pop(seq)
...
>>> @undoable
... def pop(seq):
...     value = seq.pop()
...     yield 'Pop'
...     add(seq, value)
...
>>> seq = [3, 6]
>>> add(seq, 4)
>>> seq
[3, 6, 4]
>>> stack().undo()
>>> seq
[3, 6]
>>> pop(seq)
>>> seq
[3]
>>> stack().undo()
>>> seq
[3, 6]
```

3.3 Clearing the stack

The stack may be cleared if, for example, the document is saved.

```
>>> stack().canundo()
True
>>> stack().clear()
>>> stack().canundo()
False
```

It is also possible to record a savepoint to check if there have been any changes.

```
>>> add(seq, 5)
>>> stack().haschanged()
True
>>> stack().savepoint()
>>> stack().haschanged()
False
>>> stack().undo()
>>> stack().haschanged()
True
```

3.4 Groups

A series of actions may be grouped into a single action using the `group` context manager.

```
>>> seq = []
>>> with group('Add many'):
...     for item in [4, 6, 8]:
...         add(seq, item)
>>> seq
[4, 6, 8]
>>> stack().undocount()
1
>>> stack().undo()
>>> seq
[]
```

CHAPTER 4

Advanced Usage

Actions can be created in a variety of ways. All that is required is that an action which has occurred has *do*, *undo* and *text* methods, none of which accept any arguments. The action must also be added to the stack manually using `Stack.append`. The simplest way of creating custom actions is to create a class which provides these methods and adds itself to the stack when created.

Members

undo.**undoable**()
Decorator which creates a new undoable action type.

This decorator should be used on a generator of the following format:

```
@undoable
def operation(*args):
    do_operation_code
    yield 'descriptive text'
    undo_operation_code
```

undo.**group**()
Return a context manager for grouping undoable actions. All actions which occur within the group will be undone by a single call of `Stack.undo`, e.g.

```
>>> @undoable
... def operation(n):
...     yield
...     print(n)
>>> with group('text'):
...     for n in range(3):
...         operation(n)
>>> operation(3)
>>> stack().undo()
3
>>> stack().undo()
2
1
0
```

undo.**stack**()
Returns the currently set `Stack` instance. If no stack has been set then a new instance is created and set.

undo.**setstack**(*stack*)
Set the `Stack` instance to use as the undo stack.

class undo.**Stack**
An undo stack. `stack` can usually be called instead of creating an instance iof this diectly.

The two key features are the `redo` and `undo` methods. If an exception occurs during doing or undoing a undoable, the undoable aborts and the stack is cleared to avoid any further data corruption.

The stack provides two properties for tracking actions: *docallback* and *undocallback*. Each of these allow a callback function to be set which is called when an action is done or undone repectively. By default, they do nothing.

```
>>> def done():
...     print('Can now undo: {}'.format(stack().undotext()))
>>> def undone():
...     print('Can now redo: {}'.format(stack().redotext()))
>>> stack().docallback = done
>>> stack().undocallback = undone
>>> @undoable
... def action():
...     yield 'An action'
>>> action()
Can now undo: Undo An action
>>> stack().undo()
Can now redo: Redo An action
>>> stack().redo()
Can now undo: Undo An action
```

Setting them back to `lambda: None` will stop any further actions.

```
>>> stack().docallback = stack().undocallback = lambda: None
>>> action()
>>> stack().undo()
```

It is possible to mark a point in the undo history when the document handled is saved. This allows the undo system to report whether a document has changed. The point is marked using `savepoint()` and `haschanged()` returns whether or not the state has changed (either by doing or undoing an action). Only one savepoint can be tracked, marking a new one removes the old one.

```
>>> stack().savepoint()
>>> stack().haschanged()
False
>>> action()
>>> stack().haschanged()
True
```

canundo()
Return `True` if undos are available.

canredo()
Return `True` if redos are available.

redo()
Redo the last undone action. This is only possible if no other actions have occurred since the last undo call.

undo()
Undo the last action.

clear()
Clear the undo list.

undocount()
Return the number of undos available.

redocount()
Return the number of redos available.

undotext()
Return a description of the next available undo.

setreceiver([*receiver=None*])
Set an object to receiver commands pushed onto the stack.

By default the receiver is an internally managed stack, but it can be set to any object with an *append()* method. This is used mainly for grouping actions.

resetreceiver()
Reset the receiver to the internal stack.

append(*action*)
Add an `undoable` action to the stack, using `receiver.append()`.

savepoint()
Set the current point in the undo/redo history as the savepoint. This makes it possible to check whether changes have been made.

haschanged()
Return `True` if the state has changed since the savepoint. This will always return `True` if the savepoint has not been set.

Python Module Index

u