Morris Documentation

Release 1.2

Zygmunt Krynicki

February 16, 2015

Contents

1 Morris - an announcement (signal/event) system for Python **3**
1.1 Features . . . 3

2 Installation **5**

3 Usage **7**
3.1 `morris` – announcement (signal/event) system for Python . . . 7
3.2 Reference . . . 13
3.3 Internals . . . 22

4 Contributing **23**
4.1 Types of Contributions . . . 23
4.2 Get Started! . . . 24
4.3 Pull Request Guidelines . . . 24
4.4 Tips . . . 25

5 Credits **27**
5.1 Development Lead . . . 27
5.2 Contributors . . . 27

6 History **29**
6.1 1.2 (2015-02-03) . . . 29
6.2 1.1 (2015-02-02) . . . 29
6.3 1.0 (2014-09-21) . . . 29
6.4 2012-2014 . . . 29

7 Indices and tables **31**

Python Module Index **33**

Contents:

CHAPTER 1

Morris - an announcement (signal/event) system for Python

1.1 Features

- Free software: LGPLv3 license
- Documentation: https://morris.readthedocs.org.
- Create signals with a simple decorator `morris.signal`
- Send signals by calling the decorated method or function
- Connect to and disconnect from signals with `morris.signal.connect()` and `morris.signal.disconnect()`.
- Test your code with `morris.SignalTestCase.watchSignal()`, `morris.SignalTestCase.assertSignalFired()`, `morris.SignalTestCase.assertSignalNotFired()` and `morris.SignalTestCase.assertSignalOrdering()`

CHAPTER 2

Installation

At the command line:

```
$ easy_install morris
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv morris
$ pip install morris
```

CHAPTER 3

Usage

3.1 `morris` – announcement (signal/event) system for Python

The morris module defines two main classes `signal` and `SignalTestCase`.

3.1.1 Defining Signals

Note: Since version 1.1 `Signal.define` and `signal` are identical

You can import the `signal` class and use idiomatic code like:

```
>>> from morris import signal

>>> # NOTE: classic python 2.x classes are not supported
>>> class Klass(object):
...     @signal
...     def on_foo(self):
...         pass

>>> @signal
... def on_bar():
...     pass
```

3.1.2 Connecting signal listeners

Connecting signals is equally easy, just call `signal.connect()`

```
>>> def handler():
...     print("handling signal")

>>> obj = Klass()
>>> obj.on_foo.connect(handler)
>>> on_bar.connect(handler)
```

3.1.3 Firing signals

To fire a signal simply *call* the signal object:

```
>>> obj.on_foo()
handling signal
>>> on_bar()
handling signal
```

Typically you will want to pass some additional arguments. Both positional and keyword arguments are supported:

```
>>> @signal
... def on_bar_with_args(arg1, arg2):
...     print("fired!")

>>> on_bar_with_args('foo', arg2='bar')
fired!
```

If you are working in a tight loop it is slightly faster to construct the list of positional arguments and the dictionary of keyword arguments and call the `Signal.fire()` method directly:

```
>>> args = ('foo',)
>>> kwargs = {'arg2': 'bar'}
>>> for i in range(3):
...     on_bar_with_args.fire(args, kwargs)
fired!
fired!
fired!
```

3.1.4 Passing additional meta-data to the signal listener

In some cases you may wish to use a generic signal handler that would benefit from knowing which signal has triggered it. To do that first make sure that your handler has a `signal` argument and then call `sig.connect(handler, pass_signal=True)`:

```
>>> def generic_handler(*args, **kwargs):
...     signal = kwargs.pop('signal')
...     print("Handling signal {}: {} {}".format(signal, args, kwargs))
```

Let's define two signals now:

```
>>> @signal
... def login(user, password):
...     pass
>>> @signal
... def logout(user):
...     pass
```

And connect both to the same handler:

```
>>> login.connect(generic_handler, pass_signal=True)
>>> logout.connect(generic_handler, pass_signal=True)
```

Now we can fire either one and see our handler work:

```
>>> login(str('user'), password=str('pass'))
Handling signal <signal name:'login'>: ('user',) {'password': 'pass'}
>>> logout(str('user'))
Handling signal <signal name:'logout'>: ('user',) {}
```

Note: The example uses `str(...)` to have identical output on Python 2.7 and 3.x but `str()` it is otherwise useless.

This also works with classes:

```
>>> class App(object):
...     def __repr__(self):
...         return "app"
...     @signal
...     def login(self, user, password):
...         pass
...     @signal
...     def logout(self, user):
...         pass
>>> app = App()
>>> app.login.connect(generic_handler, pass_signal=True)
>>> app.logout.connect(generic_handler, pass_signal=True)
```

We can now fire the signals, just as before:

```
>>> app.login(str('user'), password=str('pass'))
...
Handling signal <signal name:'...login' (specific to app)>:
    ('user',) {'password': 'pass'}
>>> app.logout(str('user'))
Handling signal <signal name:'...logout' (specific to app)>: ('user',) {}
```

3.1.5 Disconnecting signals

To disconnect a signal handler call `signal.disconnect()` with the same listener object that was used in `connect()`:

```
>>> obj.on_foo.disconnect(handler)
>>> on_bar.disconnect(handler)
```

3.1.6 Threading considerations

Morris doesn't do anything related to threads. Threading is diverse enough that for now it was better to just let uses handle it. There are two things that are worth mentioning though:

1. `signal.connect()` and `signal.disconnect()` should be safe to call concurrently with `signal.fire()` since fire() operates on a *copy* of the list of listeners
2. Event handlers are called from the thread calling `signal.fire()`, not from the thread that was used to connect to the signal handler. If you need special provisions for working with signals in a specific thread consider calling a thread-library-specific function that calls a callable in a specific thread context.

3.1.7 Support for writing unit tests

Morris ships with support for writing tests for signals. You can use `SignalTestCase`'s support methods such as `watchSignal()`, `assertSignalFired()`, `assertSignalNotFired()` and `assertSignalOrdering()` to simplify your tests.

Here's a simple example using all of the above:

```
>>> class App(object):
...     @signal
...     def on_login(self, user):
...         pass
...     @signal
...     def on_logout(self, user):
...         pass
...     def login(self, user):
...         self.on_login(user)
...     def logout(self, user):
...         self.on_logout(user)

>>> class AppTests(SignalTestCase):
...     def setUp(self):
...         self.app = App()
...         self.watchSignal(self.app.on_login)
...         self.watchSignal(self.app.on_logout)
...     def test_login(self):
...         # Log the user in, then out
...         self.app.login("user")
...         self.app.logout("user")
...         # Ensure that both login and logout signals were sent
...         event1 = self.assertSignalFired(self.app.on_login, 'user')
...         event2 = self.assertSignalFired(self.app.on_logout, 'user')
...         # Ensure that signals were fired in the right order
...         self.assertSignalOrdering(event1, event2)
...         # Ensure that we didn't login as admin
...         self.assertSignalNotFired(self.app.on_login, 'admin')

>>> import sys
>>> suite = unittest.TestLoader().loadTestsFromTestCase(AppTests)
>>> runner = unittest.TextTestRunner(stream=sys.stdout, verbosity=2)
>>> runner.run(suite)
test_login (morris.AppTests) ... ok

----------------------------------------------------------------------
Ran 1 test in ...s

OK
<unittest.runner.TextTestResult run=1 errors=0 failures=0>
```

3.1.8 Implementation notes

At some point in time one may need to peek under the cover and understand where the list of signal listeners is being stored and how signals interact with classes. First of all, the `signal` class can be used as a Python descriptor. Descriptors are objects that have methods such as `__get__`, `__set__` or `__delete__`.

You have most certainly used descriptors before, in fact the well-known `@property` decorator is nothing more than a class with methods such as listed above.

When used as a descriptor, a signal object will **create new signal objects each time it is being accessed on an instance of some class**. The instance of some class will be injected with a `__signals__` dictionary that contains signals that have been accessed.

Consider this example:

```
>>> class Foo(object):
...     @signal
...     def ping(self):
...         pass
```

Here `Foo.ping` is one instance of `signal`. When that instance is being accessed on a class it simply returns itself.

```
>>> Foo.ping
<signal name:'...ping'>
```

Note: While this looks similar to decorating a function it is functioning in a totally different way. Signals decorating plain functions (outside of a class definition body) are not using their descriptor nature.

Now, let's instantiate `Foo` and see what's inside:

```
>>> foo = Foo()
>>> foo.__dict__
{}
```

Nothing is inside, but there will be once we access `foo.ping`. Morris will create a new `signal` object associated with both the `foo` instance and the `foo.ping` method. It will look for `foo.__signals__` and not having found any will create one from an empty dictionary. Lastly morris will add the newly created signal object to the dictionary. This way each time we access `foo.ping` (on the particular `foo` object) we'll get exactly the same signal object in return.

```
>>> foo.ping
<signal name:'...ping' (specific to <...Foo object at ...>)>
>>> foo.__dict__
{'__signals__':
 {'...ping': <signal name:'...ping'
  (specific to <...Foo object at ...>)>}}
```

This all happens transparently the first time that code such as `foo.ping.connect(...)` is executed. When you connect a signal morris simply needs a place to store the list of listeners and that is in a signal object itself. We can now register a simple listener.

```
>>> def handler():
...     pass
>>> foo.ping.connect(handler)
```

Handlers are stored in the `signal.listeners()` attribute. They are stored as a list of `listenerinfo` tuples. Note that the first responder (the decorated function itself) is also present, here it is wrapped in the special (specific to morris) `boundmethod` class.

```
>>> foo.ping.listeners
[listenerinfo(listener=<...boundmethod object at ...>, pass_signal=False),
 listenerinfo(listener=<function handler at ...>, pass_signal=False)]
```

Now, let's compare this to using signals as a function decorator:

```
>>> @signal
... def standalone():
...     pass
```

The `standalone()` function is now *replaced* by the correspondingly-named signal object:

```
>>> standalone
<signal name:'standalone'>
```

The original function is connected as the first responder though:

```
>>> standalone.listeners
[listenerinfo(listener=<function ...standalone at ...>, pass_signal=False)]
```

Since there are no extra objects, there is no __dict__ and no __signals__ either.

3.1.9 Using @signal on class with __slots__

Since (having read the previous section) you already know that signal descriptors access the __signals__ attribute on objects of classes they belong to, to use signals on a class that uses __slots__ you need to reserve the __signals__ slot up-front.

```
>>> class Slotted(object):
...     __slots__ = ('__signals__')
...     @signal
...     def ping(self):
...         pass
>>> Slotted.ping
<signal name:'...ping'>
>>> slotted = Slotted()
>>> slotted.ping
<signal name:'...ping' (specific to <...Slotted object at ...>)>
>>> slotted.__signals__
{'...ping': <signal name:'...ping'
 (specific to <...Slotted object at ...>)>}
```

3.1.10 Creating signals explicitly

In all of the examples above we've been using signal as a decorator for existing methods or functions. This is fine for the vast majority of code but in some cases it may be beneficial to create signal objects explicitly. This may be of use in meta-programming, for example.

The `signal` class may be instantiated in the two following ways:

- with the signal name (and no listeners)
- with the first responder function (which becomes the first listener)

The second mode also has a special special case where the first responder. Let's examine than now. First, the plain signal object:

```
>>> signal(str("my-signal"))
<signal name:'my-signal'>
```

This is a normal signal object, we can call it to fire the signal, we can use the `signal.connect()` method to add listeners, etc. If you want to create standalone signals, this is the best way to do it.

Now let's examine the case where we pass a signal handler instead of the name:

```
>>> def my_signal2_handler():
...     pass
>>> signal(my_signal2_handler)
<signal name:'my_signal2_handler'>
```

Here the name of the signal is derived from the name of the handler function. We can customize the name, if desired, by passing the signal_name argument (preferably as a keyword argument to differentiate it from the `pass_signal` argument):

```
>>> signal(my_signal2_handler, signal_name='my-signal-2')
<signal name:'my-signal-2'>
```

Both examples that pass a handler are identical to what happens when decorating a regular function. There is nothing special about this mode either.

The last, and somewhat special, mode is where the handler is an instance of `boundmethod` (which is implemented inside morris). In the Python 2.x world, python had bound methods but they were removed. We still benefit from them, a little, hence they are back.

```
>>> class C(object):
...     def handler(self):
...         pass
>>> signal(boundmethod(C(), C.handler))
<signal name:'...handler' (specific to <...C object at ...>)>
```

Note: It is possible to remove boundmethod and rely `func.__self__` but this was not done, yet. Contributions are welcome!

To summarize this section, some simple rules:

- each signal object has a list of listeners
- signal objects act as descriptors and create per-instance signal objects
- signal object created this way are stored in per-instance `__signals__` attribute

3.2 Reference

class `morris.`**`signal`**(*name_or_first_responder*, *pass_signal=False*, *signal_name=None*)

Basic signal that supports arbitrary listeners.

While this class can be used directly it is best used with the helper decorator Signal.define on a function or method. See the documentation for the `morris` module for details.

Attr _name Name of the signal, typically accessed via `name()`.

Attr _listeners List of signal listeners. Each item is a tuple `(listener, pass_signal)` that encodes how to call the listener.

`__call__`(**args*, ***kwargs*)

Call fire() with all arguments forwarded transparently

This is provided for convenience so that a signal can be fired just by a simple method or function call and so that signals can be passed to other APIs that don't understand the `fire()` method.

`__get__`(*instance*, *owner*)

Descriptor __get__ method

This method is called when a signal-decorated method is being accessed via an object or a class. It is never called for decorated functions.

Parameters

- **instance** – Instance of the object the descriptor is being used on. This is None when the descriptor is accessed on a class.
- **owner** – The class that the descriptor is defined on.

Returns If `instance` is None we return ourselves, this is what descriptors typically do. If `instance` is not None we return a unique `Signal` instance that is specific to that object and signal. This is implemented by storing the signal inside the object's __signals__ attribute.

__init__(*name_or_first_responder*, *pass_signal=False*, *signal_name=None*)
Construct a signal with the given name

Parameters

- **name_or_first_responder** – Either the name of the signal to construct or a callable which will be the first responder. In the latter case the callable is used to obtain the name of the signal.
- **pass_signal** – An optional flag that instructs morris to pass the signal object itself to the first responder (as the `signal` argument). This is only used in the case where `name_or_first_responder` is a callable.
- **signal_name** – Optional name of the signal. This is meaningful only when the first argument `name_or_first_responder` is a callable. When that happens this argument is used and no guessing based on __qualname__ or __name__ is being used.

__repr__()
A representation of the signal.

There are two possible representations:

- a signal object created via a signal descriptor on an object
- a signal object acting as a descriptor or function decorator

__weakref__
list of weak references to the object (if defined)

connect(*listener*, *pass_signal=False*)
Connect a new listener to this signal

Parameters

- **listener** – The listener (callable) to add
- **pass_signal** – An optional argument that controls if the signal object is explicitly passed to this listener when it is being fired. If enabled, a `signal=` keyword argument is passed to the listener function.

Returns None

The listener will be called whenever `fire()` or `__call__()` are called. The listener is appended to the list of listeners. Duplicates are not checked and if a listener is added twice it gets called twice.

define
alias of `signal`

disconnect(*listener*, *pass_signal=False*)
Disconnect an existing listener from this signal

Parameters

- **listener** – The listener (callable) to remove
- **pass_signal** – An optional argument that controls if the signal object is explicitly passed to this listener when it is being fired. If enabled, a `signal=` keyword argument is passed to the listener function.

  Here, this argument simply aids in disconnecting the right listener. Make sure to pass the same value as was passed to `connect()`

Raises ValueError If the listener (with the same value of pass_signal) is not present

Returns None

fire(*args*, *kwargs*)
Fire this signal with the specified arguments and keyword arguments.

Typically this is used by using `__call__()` on this object which is more natural as it does all the argument packing/unpacking transparently.

first_responder
The first responder function.

This is the function that the `signal` may have been instantiated with. It is only relevant if the signal itself is used as a *descriptor* in a class (where it decorates a method).

For example, contrast the access of the signal on the class and on a class instance:

```
>>> class C(object):
...     @signal
...     def on_foo(self):
...         pass
```

Class access gives uses the descriptor protocol to expose the actual signal object.

```
>>> C.on_foo
<signal name:'...on_foo'>
```

Here we can use the `first_responder` property to see the actual function.

```
>>> C.on_foo.first_responder
<function ...on_foo at ...>
```

Object access is different as now the signal instance is specific to the object:

```
>>> C().on_foo
<signal name:'...on_foo' (specific to <morris.C object at ...)>
```

And now the first responder is gone (it is now buried inside the `listeners()` list):

```
>>> C().on_foo.first_responder
```

listeners
List of `listenerinfo` objects associated with this signal

The list of listeners is considered part of an implementation detail but is exposed for convenience. This is always the real list. Keep this in mind while connecting and disconnecting listeners. During the time `fire()` is called the list of listeners can be changed but won't take effect until after `fire()` returns.

name
Name of the signal

For signals constructed manually (i.e. by calling `Signal()`) the name is arbitrary. For signals constructed using either `Signal.define()` or `signal` the name is obtained from the decorated function.

On python 3.3+ the qualified name is used (see **PEP 3155**), on earlier versions the plain name is used (without the class name). The name is identical regardless of how the signal is being accessed:

```
>>> class C(object):
...     @signal
...     def on_meth(self):
...         pass
```

As a descriptor on a class:

```
>>> C.on_meth.name
'...on_meth'
```

As a descriptor on an object:

```
>>> C().on_meth.name
'...on_meth'
```

As a decorated function:

```
>>> @signal
... def on_func():
...     pass
>>> on_func.name
'on_func'
```

signal_name

Name of the signal

For signals constructed manually (i.e. by calling `Signal()`) the name is arbitrary. For signals constructed using either `Signal.define()` or `signal` the name is obtained from the decorated function.

On python 3.3+ the qualified name is used (see **PEP 3155**), on earlier versions the plain name is used (without the class name). The name is identical regardless of how the signal is being accessed:

```
>>> class C(object):
...     @signal
...     def on_meth(self):
...         pass
```

As a descriptor on a class:

```
>>> C.on_meth.name
'...on_meth'
```

As a descriptor on an object:

```
>>> C().on_meth.name
'...on_meth'
```

As a decorated function:

```
>>> @signal
... def on_func():
...     pass
>>> on_func.name
'on_func'
```

class `morris.`**SignalInterceptorMixIn**

A mix-in class for TestCase-like classes that adds extra methods for working with and testing signals. This class may be of use if the base TestCase class is not the standard `unittest.TestCase` class but the user still wants to take advantage of the extra methods provided here.

assertSignalFired(*signal*, **args*, ***kwargs*)

Assert that a signal was fired with appropriate arguments.

Parameters

- **signal** – The `Signal` that should have been fired. Typically this is `SomeClass.on_some_signal` reference

- **args** – List of positional arguments passed to the signal handler
- **kwargs** – List of keyword arguments passed to the signal handler

Returns A 3-tuple (signal, args, kwargs) that describes that event

assertSignalNotFired(*signal*, **args*, ***kwargs*)
Assert that a signal was fired with appropriate arguments.

Parameters

- **signal** – The `Signal` that should not have been fired. Typically this is `SomeClass.on_some_signal` reference
- **args** – List of positional arguments passed to the signal handler
- **kwargs** – List of keyword arguments passed to the signal handler

assertSignalOrdering(**expected_events*)
Assert that a signals were fired in a specific sequence.

Parameters **expected_events** – A (varadic) list of events describing the signals that were fired Each element is a 3-tuple (signal, args, kwargs) that describes the event.

Note: If you are using `assertSignalFired()` then the return value of that method is a single event that can be passed to this method

---

watchSignal(*signal*)
Setup provisions to watch a specified signal

Parameters **signal** – The `Signal` to watch for.

After calling this method you can use `assertSignalFired()` and `assertSignalNotFired()` with the same signal.

class `morris.`**SignalTestCase**(*methodName='runTest'*)
Bases: `unittest.case.TestCase`, `morris.SignalInterceptorMixIn`

A `unittest.TestCase` subclass that simplifies testing uses of the Morris signals. It provides three assertion methods and one utility helper method for observing signal events.

addCleanup(*function*, **args*, ***kwargs*)
Add a function, with arguments, to be called when the test is completed. Functions added are called on a LIFO basis and are called after tearDown on test failure or success.

Cleanup items are called even if setUp fails (unlike tearDown).

addTypeEqualityFunc(*typeobj*, *function*)
Add a type specific assertEqual style function to compare a type.

This method is for use by TestCase subclasses that need to register their own type equality functions to provide nicer error messages.

Args:

typeobj: The data type to call this function on when both values are of the same type in assertEqual().

function: The callable taking two arguments and an optional msg= argument that raises self.failureException with a useful error message when the two arguments are not equal.

assertAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal

places (default 7) and comparing to zero, or by comparing that the between the two objects is more than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

If the two objects compare equal then they will automatically compare almost equal.

assertCountEqual (*first*, *second*, *msg=None*)
An unordered sequence comparison asserting that the same elements, regardless of order. If the same element occurs more than once, it verifies that the elements occur the same number of times.

self.assertEqual(Counter(list(first)), Counter(list(second)))

Example:

- [0, 1, 1] and [1, 0, 1] compare equal.
- [0, 0, 1] and [0, 1] compare unequal.

assertDictContainsSubset (*subset*, *dictionary*, *msg=None*)
Checks whether dictionary is a superset of subset.

assertEqual (*first*, *second*, *msg=None*)
Fail if the two objects are unequal as determined by the '==' operator.

assertFalse (*expr*, *msg=None*)
Check that the expression is false.

assertGreater (*a*, *b*, *msg=None*)
Just like self.assertTrue(a > b), but with a nicer default message.

assertGreaterEqual (*a*, *b*, *msg=None*)
Just like self.assertTrue(a >= b), but with a nicer default message.

assertIn (*member*, *container*, *msg=None*)
Just like self.assertTrue(a in b), but with a nicer default message.

assertIs (*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is b), but with a nicer default message.

assertIsInstance (*obj*, *cls*, *msg=None*)
Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.

assertIsNone (*obj*, *msg=None*)
Same as self.assertTrue(obj is None), with a nicer default message.

assertIsNot (*expr1*, *expr2*, *msg=None*)
Just like self.assertTrue(a is not b), but with a nicer default message.

assertIsNotNone (*obj*, *msg=None*)
Included for symmetry with assertIsNone.

assertLess (*a*, *b*, *msg=None*)
Just like self.assertTrue(a < b), but with a nicer default message.

assertLessEqual (*a*, *b*, *msg=None*)
Just like self.assertTrue(a <= b), but with a nicer default message.

assertListEqual (*list1*, *list2*, *msg=None*)
A list-specific equality assertion.

Args: list1: The first list to compare. list2: The second list to compare. msg: Optional message to use on failure instead of a list of

differences.

assertLogs(*logger=None*, *level=None*)
Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children. If omitted, *level* defaults to INFO and *logger* defaults to the root logger.

This method must be used as a context manager, and will yield a recording object with two attributes: *output* and *records*. At the end of the context manager, the *output* attribute will be a list of the matching formatted log messages and the *records* attribute will be a list of the corresponding LogRecord objects.

Example:

```
with self.assertLogs('foo', level='INFO') as cm:
    logging.getLogger('foo').info('first message')
    logging.getLogger('foo.bar').error('second message')
self.assertEqual(cm.output, ['INFO:foo:first message',
                             'ERROR:foo.bar:second message'])
```

assertMultiLineEqual(*first*, *second*, *msg=None*)
Assert that two multi-line strings are equal.

assertNotAlmostEqual(*first*, *second*, *places=None*, *msg=None*, *delta=None*)
Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the between the two objects is less than the given delta.

Note that decimal places (from zero) are usually not the same as significant digits (measured from the most signficant digit).

Objects that are equal automatically fail.

assertNotEqual(*first*, *second*, *msg=None*)
Fail if the two objects are equal as determined by the '!=' operator.

assertNotIn(*member*, *container*, *msg=None*)
Just like self.assertTrue(a not in b), but with a nicer default message.

assertNotIsInstance(*obj*, *cls*, *msg=None*)
Included for symmetry with assertIsInstance.

assertNotRegex(*text*, *unexpected_regex*, *msg=None*)
Fail the test if the text matches the regular expression.

assertRaises(*excClass*, *callableObj=None*, **args*, ***kwargs*)
Fail unless an exception of class excClass is raised by callableObj when invoked with arguments args and keyword arguments kwargs. If a different type of exception is raised, it will not be caught, and the test case will be deemed to have suffered an error, exactly as for an unexpected exception.

If called with callableObj omitted or None, will return a context object used like this:

```
with self.assertRaises(SomeException):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertRaises is used as a context object.

The context manager keeps a reference to the exception as the 'exception' attribute. This allows you to inspect the exception after the assertion:

```
with self.assertRaises(SomeException) as cm:
    do_something()
the_exception = cm.exception
self.assertEqual(the_exception.error_code, 3)
```

assertRaisesRegex(*expected_exception*, *expected_regex*, *callable_obj=None*, **args*, ***kwargs*)
Asserts that the message in a raised exception matches a regex.

Args: expected_exception: Exception class expected to be raised. expected_regex: Regex (re pattern object or string) expected

to be found in error message.

callable_obj: Function to be called. msg: Optional message used in case of failure. Can only be used

when assertRaisesRegex is used as a context manager.

args: Extra args. kwargs: Extra kwargs.

assertRegex(*text*, *expected_regex*, *msg=None*)
Fail the test unless the text matches the regular expression.

assertSequenceEqual(*seq1*, *seq2*, *msg=None*, *seq_type=None*)
An equality assertion for ordered sequences (like lists and tuples).

For the purposes of this function, a valid ordered sequence type is one which can be indexed, has a length, and has an equality operator.

Args: seq1: The first sequence to compare. seq2: The second sequence to compare. seq_type: The expected datatype of the sequences, or None if no

datatype should be enforced.

msg: Optional message to use on failure instead of a list of differences.

assertSetEqual(*set1*, *set2*, *msg=None*)
A set-specific equality assertion.

Args: set1: The first set to compare. set2: The second set to compare. msg: Optional message to use on failure instead of a list of

differences.

assertSetEqual uses ducktyping to support different types of sets, and is optimized for sets specifically (parameters must support a difference method).

assertSignalFired(*signal*, **args*, ***kwargs*)
Assert that a signal was fired with appropriate arguments.

Parameters

- **signal** – The `Signal` that should have been fired. Typically this is `SomeClass.on_some_signal` reference
- **args** – List of positional arguments passed to the signal handler
- **kwargs** – List of keyword arguments passed to the signal handler

Returns A 3-tuple (signal, args, kwargs) that describes that event

assertSignalNotFired(*signal*, **args*, ***kwargs*)
Assert that a signal was fired with appropriate arguments.

Parameters

- **signal** – The `Signal` that should not have been fired. Typically this is `SomeClass.on_some_signal` reference
- **args** – List of positional arguments passed to the signal handler
- **kwargs** – List of keyword arguments passed to the signal handler

assertSignalOrdering(**expected_events*)
Assert that a signals were fired in a specific sequence.

Parameters **expected_events** – A (varadic) list of events describing the signals that were fired Each element is a 3-tuple (signal, args, kwargs) that describes the event.

Note: If you are using `assertSignalFired()` then the return value of that method is a single event that can be passed to this method

---

assertTrue(*expr*, *msg=None*)
Check that the expression is true.

assertTupleEqual(*tuple1*, *tuple2*, *msg=None*)
A tuple-specific equality assertion.

Args: tuple1: The first tuple to compare. tuple2: The second tuple to compare. msg: Optional message to use on failure instead of a list of

differences.

assertWarns(*expected_warning*, *callable_obj=None*, **args*, ***kwargs*)
Fail unless a warning of class warnClass is triggered by callable_obj when invoked with arguments args and keyword arguments kwargs. If a different type of warning is triggered, it will not be handled: depending on the other warning filtering rules in effect, it might be silenced, printed out, or raised as an exception.

If called with callable_obj omitted or None, will return a context object used like this:

```
with self.assertWarns(SomeWarning):
    do_something()
```

An optional keyword argument 'msg' can be provided when assertWarns is used as a context object.

The context manager keeps a reference to the first matching warning as the 'warning' attribute; similarly, the 'filename' and 'lineno' attributes give you information about the line of Python code from which the warning was triggered. This allows you to inspect the warning after the assertion:

```
with self.assertWarns(SomeWarning) as cm:
    do_something()
the_warning = cm.warning
self.assertEqual(the_warning.some_attribute, 147)
```

assertWarnsRegex(*expected_warning*, *expected_regex*, *callable_obj=None*, **args*, ***kwargs*)
Asserts that the message in a triggered warning matches a regexp. Basic functioning is similar to assertWarns() with the addition that only warnings whose messages also match the regular expression are considered successful matches.

Args: expected_warning: Warning class expected to be triggered. expected_regex: Regex (re pattern object or string) expected

to be found in error message.

callable_obj: Function to be called. msg: Optional message used in case of failure. Can only be used

when assertWarnsRegex is used as a context manager.

args: Extra args. kwargs: Extra kwargs.

debug()
Run the test without collecting errors in a TestResult

doCleanups()
Execute all cleanup functions. Normally called for you after tearDown.

fail(*msg=None*)
Fail immediately, with the given message.

failureException
alias of `AssertionError`

setUp()
Hook method for setting up the test fixture before exercising it.

setUpClass()
Hook method for setting up class fixture before running tests in the class.

shortDescription()
Returns a one-line description of the test, or None if no description has been provided.

The default implementation of this method returns the first line of the specified test method's docstring.

skipTest(*reason*)
Skip this test.

subTest(*msg=None*, ***params*)
Return a context manager that will return the enclosed block of code in a subtest identified by the optional message and keyword parameters. A failure in the subtest marks the test case as failed but resumes execution at the end of the enclosed block, allowing further test code to be executed.

tearDown()
Hook method for deconstructing the test fixture after testing it.

tearDownClass()
Hook method for deconstructing the class fixture after running all tests in the class.

watchSignal(*signal*)
Setup provisions to watch a specified signal

Parameters **signal** – The `Signal` to watch for.

After calling this method you can use `assertSignalFired()` and `assertSignalNotFired()` with the same signal.

3.3 Internals

class `morris.`**listenerinfo**
listenerinfo(listener, pass_signal)

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

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

listener
Alias for field number 0

pass_signal
Alias for field number 1

class `morris.`**boundmethod**(*instance*, *func*)
A helper class that allows us to emulate a bound method

This class emulates a bond method by storing an object `instance`, function `func` and calling `instance.``func``()` whenever the boundmethod object itself is called.

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/zyga/morris/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

4.1.4 Write Documentation

Morris could always use more documentation, whether as part of the official Morris docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/zyga/morris/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *morris* for local development.

1. Fork the *morris* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/morris.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv morris
   $ cd morris/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ flake8 morris
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.2, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/zyga/morris/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ python -m unittest morris.tests
```

CHAPTER 5

Credits

5.1 Development Lead

- Zygmunt Krynicki <zygmunt.krynicki@canonical.com>

While under development as a part of the Plainbox project. Sylvain Pineau has contributed a number of improvements. Thanks Sylvain!

5.2 Contributors

None yet. Why not be the first?

CHAPTER 6

History

6.1 1.2 (2015-02-03)

- Merge backwards compatibility features for Plainbox migration. (signal_name, SignalInterceptorMixIn)
- Fix a bug in signal.__repr__()
- Document internals better

6.2 1.1 (2015-02-02)

- Merge `Signal` and `signal` into one class.
- Make `Signal` an alias of `signal`.
- Make `Signal.define` an alias of `signal`.
- Fix signal support on standalone functions (https://github.com/zyga/morris/issues/1)
- Add more documentation and tests
- Enable travis-ci.org integration

6.3 1.0 (2014-09-21)

- First release on PyPI.

6.4 2012-2014

- Released on PyPI as a part of plainbox as `plainbox.impl.signal`

CHAPTER 7

Indices and tables

- *genindex*
- *modindex*
- *search*

m

morris, 7

Symbols

__call__() (morris.signal method), 13
__get__() (morris.signal method), 13
__init__() (morris.signal method), 14
__repr__() (morris.signal method), 14
__weakref__ (morris.signal attribute), 14

A

addCleanup() (morris.SignalTestCase method), 17
addTypeEqualityFunc() (morris.SignalTestCase method), 17
assertAlmostEqual() (morris.SignalTestCase method), 17
assertCountEqual() (morris.SignalTestCase method), 18
assertDictContainsSubset() (morris.SignalTestCase method), 18
assertEqual() (morris.SignalTestCase method), 18
assertFalse() (morris.SignalTestCase method), 18
assertGreater() (morris.SignalTestCase method), 18
assertGreaterEqual() (morris.SignalTestCase method), 18
assertIn() (morris.SignalTestCase method), 18
assertIs() (morris.SignalTestCase method), 18
assertIsInstance() (morris.SignalTestCase method), 18
assertIsNone() (morris.SignalTestCase method), 18
assertIsNot() (morris.SignalTestCase method), 18
assertIsNotNone() (morris.SignalTestCase method), 18
assertLess() (morris.SignalTestCase method), 18
assertLessEqual() (morris.SignalTestCase method), 18
assertListEqual() (morris.SignalTestCase method), 18
assertLogs() (morris.SignalTestCase method), 19
assertMultiLineEqual() (morris.SignalTestCase method), 19
assertNotAlmostEqual() (morris.SignalTestCase method), 19
assertNotEqual() (morris.SignalTestCase method), 19
assertNotIn() (morris.SignalTestCase method), 19
assertNotIsInstance() (morris.SignalTestCase method), 19
assertNotRegex() (morris.SignalTestCase method), 19
assertRaises() (morris.SignalTestCase method), 19
assertRaisesRegex() (morris.SignalTestCase method), 19
assertRegex() (morris.SignalTestCase method), 20
assertSequenceEqual() (morris.SignalTestCase method), 20
assertSetEqual() (morris.SignalTestCase method), 20
assertSignalFired() (morris.SignalInterceptorMixIn method), 16
assertSignalFired() (morris.SignalTestCase method), 20
assertSignalNotFired() (morris.SignalInterceptorMixIn method), 17
assertSignalNotFired() (morris.SignalTestCase method), 20
assertSignalOrdering() (morris.SignalInterceptorMixIn method), 17
assertSignalOrdering() (morris.SignalTestCase method), 20
assertTrue() (morris.SignalTestCase method), 21
assertTupleEqual() (morris.SignalTestCase method), 21
assertWarns() (morris.SignalTestCase method), 21
assertWarnsRegex() (morris.SignalTestCase method), 21

B

boundmethod (class in morris), 22

C

connect() (morris.signal method), 14
count() (morris.listenerinfo method), 22

D

debug() (morris.SignalTestCase method), 21
define (morris.signal attribute), 14
disconnect() (morris.signal method), 14
doCleanups() (morris.SignalTestCase method), 21

F

fail() (morris.SignalTestCase method), 21
failureException (morris.SignalTestCase attribute), 22
fire() (morris.signal method), 15
first_responder (morris.signal attribute), 15

I

index() (morris.listenerinfo method), 22

L

listener (morris.listenerinfo attribute), 22
listenerinfo (class in morris), 22
listeners (morris.signal attribute), 15

M

morris (module), 7

N

name (morris.signal attribute), 15

P

pass_signal (morris.listenerinfo attribute), 22
Python Enhancement Proposals
    PEP 3155, 15, 16

S

setUp() (morris.SignalTestCase method), 22
setUpClass() (morris.SignalTestCase method), 22
shortDescription() (morris.SignalTestCase method), 22
signal (class in morris), 13
signal_name (morris.signal attribute), 16
SignalInterceptorMixIn (class in morris), 16
SignalTestCase (class in morris), 17
skipTest() (morris.SignalTestCase method), 22
subTest() (morris.SignalTestCase method), 22

T

tearDown() (morris.SignalTestCase method), 22
tearDownClass() (morris.SignalTestCase method), 22

W

watchSignal() (morris.SignalInterceptorMixIn method), 17
watchSignal() (morris.SignalTestCase method), 22