signalslot Documentation

Release 0.1.1

Numergy

January 26, 2016

Contents

This package provides a simple and stupid implementation of the Signal/Slot pattern for Python. Wikipedia has a nice introduction:

> Signals and slots is a language construct introduced in Qt for communication between objects[1] which makes it easy to implement the Observer pattern while avoiding boilerplate code.

Rationale against Signal/Slot is detailed in the "Pattern" section of the documentation.

CHAPTER 1

Install

Install latest stable version:

```
pip install signalslot
```

Install development version:

```
pip install -e git+https://github.com/Numergy/signalslot
```

CHAPTER 2

Upgrade

Upgrade to the last stable version:

```
pip install -U signalslot
```

CHAPTER 3

Uninstall

```
pip uninstall signalslot
```

3.1 Signal/Slot design pattern

3.1.1 Introduction

Signal/Slot is a pattern that allows loose coupling various components of a software without having to introduce boilerplate code. Loose coupling of components allows better modularity in software code which has the nice side effect of making it easier to test because less dependencies means less mocking and monkey patching.

Signal/Slot is a widely used pattern, many frameworks have it built-in including Django, Qt and probably many others. If you have a standalone project then you probably don't want to add a big dependency like PyQt or Django just for a Signal/Slot framework. There are a couple of standalone libraries which allow to acheive a similar result, like Circuits or PyPubSub, which has way more features than `signalslots`, like messaging over the network and is a quite complicated and has weird (non-PyPi hosted) dependencies and is not PEP8 compliant ...

`signalslot` has the vocation of being a light and simple implementation of the well known Signal/Slot design pattern provided as a classic quality Python package.

3.1.2 Tight coupling

Consider such a code in `your_client.py`:

```
import your_service
import your_dirty_hack  # WTH is that doing here ? huh ?

class YourClient(object):
    def something_happens(self, some_argument):
        your_service.something_happens(some_argument)
        your_dirty_hack.something_happens(some_argument)
```

The problem with that code is that it ties `your_client` with `your_service` and `your_dirty_hack` which you really didn't want to put there, but had to, "until you find a better place for it".

Tight coupling makes code harder to test because it takes more mocking and harder to maintain because it has more dependencies.

An improvement would be to acheive the same while keeping components loosely coupled.

3.1.3 Observer pattern

You could implement an Observer pattern in `YourClient` by adding boilerplate code:

```
class YourClient(object):
    def __init__(self):
        self.observers = []

    def register_observer(self, observer):
        self.observers.append(observer)

    def something_happens(self, some_argument):
        for observer in self.observers:
            observer.something_happens(some_argument)
```

This implementation is a bit dumb, it doesn't check the compatibility of observers for example, also it's additionnal code you'd have to test, and it's "boilerplate".

This would work if you have control on instanciation of `YourClient`, ie.:

```
your_client = YourClient()
your_client.register_observer(your_service)
your_client.register_observer(your_dirty_hack)
```

If `YourClient` is used by a framework with IoC then it might become harder:

```
service = some_framework.Service.create(
    client='your_client.YourClient')

service._client.register_observer(your_service)
service._client.register_observer(your_dirty_hack)
```

In this example, we're accessing a private python variable `_client` and that's never very good because it's not safe against forward compatibility.

3.1.4 With Signal/Slot

Using the Signal/Slot pattern, the same result could be achieved with total component decoupling. It would organise as such:

- `YourClient` defines a `something_happens` signal,
- `your_service` connects its own callback to the `something_happens`,
- so does `your_dirty_hack`,
- `YourClient.something_happens()` "emits" a signal, which in turn calls all connected callbacks.

Note that a connected callback is called a "slot" in the "Signal/Slot" pattern.

See Usage for example code.

3.2 Usage

3.2.1 `signalslot.Signal` objects

Module defining the Signal class.

class signalslot.signal.**BaseSlot**
Slot abstract class for type resolution purposes.

class signalslot.signal.**DummyLock**
Class that implements a no-op instead of a re-entrant lock.

class signalslot.signal.**Signal**(*args=None*, *name=None*, *threadsafe=False*)
Define a signal by instanciating a *Signal* object, ie.:

```
>>> conf_pre_load = Signal()
```

Optionaly, you can declare a list of argument names for this signal, ie.:

```
>>> conf_pre_load = Signal(args=['conf'])
```

Any callable can be connected to a Signal, it **must** accept keywords (**kwargs), ie.:

```
>>> def yourmodule_conf(conf, **kwargs):
...     conf['yourmodule_option'] = 'foo'
...
```

Connect your function to the signal using *connect()*:

```
>>> conf_pre_load.connect(yourmodule_conf)
```

Emit the signal to call all connected callbacks using *emit()*:

```
>>> conf = {}
>>> conf_pre_load.emit(conf=conf)
>>> conf
{'yourmodule_option': 'foo'}
```

Note that you may disconnect a callback from a signal if it is already connected:

```
>>> conf_pre_load.is_connected(yourmodule_conf)
True
>>> conf_pre_load.disconnect(yourmodule_conf)
>>> conf_pre_load.is_connected(yourmodule_conf)
False
```

connect(*slot*)
Connect a callback slot to this signal.

disconnect(*slot*)
Disconnect a slot from a signal if it is connected else do nothing.

emit(***kwargs*)
Emit this signal which will execute every connected callback slot, passing keyword arguments.

If a slot returns anything other than None, then *emit()* will return that value preventing any other slot from being called.

```
>>> need_something = Signal()
>>> def get_something(**kwargs):
...     return 'got something'
...
>>> def make_something(**kwargs):
...     print('I will not be called')
...
>>> need_something.connect(get_something)
>>> need_something.connect(make_something)
```

```
>>> need_something.emit()
'got something'
```

is_connected(*slot*)
Check if a callback `slot` is connected to this signal.

slots
Return a list of slots for this signal.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

S

B

C

D

E

I

S