Loose Coupling (Python recipe)

The "broadcaster" and "broker" modules enable loose coupling between objects in a running application.

```
######################################################################
# broadcaster.py
#

__all__ = ['Register', 'Broadcast',
            'CurrentSource', 'CurrentTitle', 'CurrentData']

listeners = {}
currentSources = []
currentTitles = []
currentData = []

def Register(listener, arguments=(), source=None, title=None):
    if not listeners.has_key((source, title)):
        listeners[(source, title)] = []

    listeners[(source, title)].append((listener, arguments))

def Broadcast(source, title, data={}):
    currentSources.append(source)
    currentTitles.append(title)
    currentData.append(data)

    listenerList = listeners.get((source, title), [])[:]
    if source != None:
        listenerList += listeners.get((None, title), [])
    if title != None:
        listenerList += listeners.get((source, None), [])

    for listener, arguments in listenerList:
        apply(listener, arguments)

    currentSources.pop()
    currentTitles.pop()
    currentData.pop()

def CurrentSource():
    return currentSources[-1]

def CurrentTitle():
    return currentTitles[-1]

def CurrentData():
    return currentData[-1]


######################################################################
# broker.py
#

__all__ = ['Register', 'Request',
            'CurrentTitle', 'CurrentData']

providers = {}
```

```python
currentTitles = []
currentData = []

def Register(title, provider, arguments=()):
    assert not providers.has_key(title)
    providers[title] = (provider, arguments)

def Request(title, data={}):
    currentTitles.append(title)
    currentData.append(data)

    result = apply(apply, providers.get(title))

    currentTitles.pop()
    currentData.pop()

    return result

def CurrentTitle():
    return currentTitles[-1]

def CurrentData():
    return currentData[-1]


######################################################################
# sample.py
#

from __future__ import nested_scopes

import broadcaster
import broker

class UserSettings:
    def __init__(self):
        self.preferredLanguage = 'English'
        # The use of lambda here provides a simple wrapper around
        # the value being provided. Every time the value is requested
        # the variable will be reevaluated by the lambda function.
        # Note the dependence on nested scopes.
        broker.Register('Preferred Language', lambda: self.preferredLanguage)

        self.preferredSkin = 'Cool Blue Skin'
        broker.Register('Preferred Skin', lambda: self.preferredSkin)

    def ChangePreferredSkinTo(self, preferredSkin):
        self.preferredSkin = preferredSkin
        broadcaster.Broadcast('Preferred Skin', 'Changed')

    def ChangePreferredLanguageTo(self, preferredLanguage):
        self.preferredLanguage = preferredLanguage
        broadcaster.Broadcast('Preferred Language', 'Changed')

def ChangeSkin():
    print 'Changing to', broker.Request('Preferred Skin')

def ChangeLanguage():
    print 'Changing to', broker.Request('Preferred Language')

broadcaster.Register(ChangeSkin, source='Preferred Skin', title='Changed')
broadcaster.Register(ChangeLanguage, source='Preferred Language', title='Changed')
```

```
userSettings = UserSettings()
userSettings.ChangePreferredSkinTo('Bright Green Skin')
userSettings.ChangePreferredSkinTo('French')
```

These modules are particularly useful in GUI applications where they help to shield application logic from UI changes.

Broadcasting is essentially equivalent to multiplexed function calls where the caller does not need to know the interface of the called function(s). A broadcaster can optionally supply data for the listeners to consume. For example if an application is about to exit, it can broadcast a message to that effect and any interested objects can do their finalization. Another example is a UI control which can broadcast a message whenever its state changes so that other objects can respond appropriately.

The broker enables the retrieval of named data even when the source of the data is not known. For example a UI control (such as an edit box) can register itself as a data provider with the broker and then any code in the application can retrieve the control's value with no knowledge of how or where the value is stored. This avoids two potential pitfalls – (1) storing data in multiple locations and requiring extra logic to keep those locations in sync and (2) proliferating the dependency upon the control's API.

The broker and broadcaster can work together quite nicely. Take, for example, an edit box that is used for entering a date. Whenever its value changes, it can broadcast a message indicating that the entered date has changed. Anything depending upon that date can respond to that message by asking the broker for the current value. Later the edit box can be replaced by a calendar control. As long as the new control broadcasts the same messages and provides the same data through the broker, no other code should need to be changed.

This idiom is thread hostile." Even if access to the module level variables was properly controlled, this style of programming is tailor made for deadlocks / race conditions. Please consider the impact carefully before using this from multiple threads.

It would be interesting to extend this idiom for use in inter-application communication (or even for use across a network).