Dependency Injection The Python Way (Python recipe)

```
#############################################################################
##
## Feature Broker
##
#############################################################################

class FeatureBroker:
   def __init__(self, allowReplace=False):
      self.providers = {}
      self.allowReplace = allowReplace
   def Provide(self, feature, provider, *args, **kwargs):
      if not self.allowReplace:
         assert not self.providers.has_key(feature), "Duplicate feature: %r" % feature
      if callable(provider):
         def call(): return provider(*args, **kwargs)
      else:
         def call(): return provider
      self.providers[feature] = call
   def __getitem__(self, feature):
      try:
         provider = self.providers[feature]
      except KeyError:
         raise KeyError, "Unknown feature named %r" % feature
      return provider()

features = FeatureBroker()

#############################################################################
##
## Representation of Required Features and Feature Assertions
##
#############################################################################

#
# Some basic assertions to test the suitability of injected features
#

def NoAssertion(obj): return True

def IsInstanceOf(*classes):
   def test(obj): return isinstance(obj, classes)
   return test

def HasAttributes(*attributes):
   def test(obj):
      for each in attributes:
         if not hasattr(obj, each): return False
      return True
   return test

def HasMethods(*methods):
   def test(obj):
      for each in methods:
         try:
            attr = getattr(obj, each)
         except AttributeError:
            return False
```

```
            if not callable(attr): return False
        return True
    return test

#
# An attribute descriptor to "declare" required features
#

class RequiredFeature(object):
    def __init__(self, feature, assertion=NoAssertion):
        self.feature = feature
        self.assertion = assertion
    def __get__(self, obj, T):
        return self.result # <-- will request the feature upon first call
    def __getattr__(self, name):
        assert name == 'result', "Unexpected attribute request other then 'result'"
        self.result = self.Request()
        return self.result
    def Request(self):
        obj = features[self.feature]
        assert self.assertion(obj), \
               "The value %r of %r does not match the specified criteria" \
               % (obj, self.feature)
        return obj

class Component(object):
    "Symbolic base class for components"

########################################################################
##
## DEMO
##
########################################################################

# ----------------------------------------------------------------------------
# Some python module defines a Bar component and states the dependencies
# We will assume that
# - Console denotes an object with a method WriteLine(string)
# - AppTitle denotes a string that represents the current application name
# - CurrentUser denotes a string that represents the current user name
#

class Bar(Component):
    con   = RequiredFeature('Console', HasMethods('WriteLine'))
    title = RequiredFeature('AppTitle', IsInstanceOf(str))
    user  = RequiredFeature('CurrentUser', IsInstanceOf(str))
    def __init__(self):
        self.X = 0
    def PrintYourself(self):
        self.con.WriteLine('-- Bar instance --')
        self.con.WriteLine('Title: %s' % self.title)
        self.con.WriteLine('User: %s' % self.user)
        self.con.WriteLine('X: %d' % self.X)

# ----------------------------------------------------------------------------
# Some other python module defines a basic Console component
#

class SimpleConsole(Component):
    def WriteLine(self, s):
        print s
```

```
# -----------------------------------------------------------------------------
# Yet another python module defines a better Console component
#

class BetterConsole(Component):
   def __init__(self, prefix=''):
      self.prefix = prefix
   def WriteLine(self, s):
      lines = s.split('\n')
      for line in lines:
         if line:
            print self.prefix, line
         else:
            print

# -----------------------------------------------------------------------------
# Some third python module knows how to discover the current user's name
#

def GetCurrentUser():
   return os.getenv('USERNAME') or 'Some User' # USERNAME is platform-specific

# -----------------------------------------------------------------------------
# Finally, the main python script specifies the application name,
# decides which components/values to use for what feature,
# and creates an instance of Bar to work with
#
if __name__ == '__main__':
   print '\n*** IoC Demo ***'
   features.Provide('AppTitle', 'Inversion of Control ...\n\n... The Python Way')
   features.Provide('CurrentUser', GetCurrentUser)
   features.Provide('Console', BetterConsole, prefix='-->') # <-- transient lifestyle
   ##features.Provide('Console', BetterConsole(prefix='-->')) # <-- singleton
lifestyle

   bar = Bar()
   bar.PrintYourself()

#
# Evidently, none of the used components needed to know about each other
# => Loose coupling goal achieved
# -----------------------------------------------------------------------------
```

Inversion of Control (IoC) Containers and the Dependency Injection pattern have drawn some attention in the Java world, and they are increasingly spreading over to .NET, too. (Perhaps we are facing a sort of "Infection OUT of Control" - IooC? ;)

IoC is all about loose coupling between components of an application, about cutting off explicit, direct dependencies, plus some goodies (most of which are useful in statically typed languages only, like automatic type/interface matching). A thorough discussion on the subject can be found at http://www.martinfowler.com/articles/injection.html
In statically typed languages, an IoC container is quite a challenge. But at the heart of it, there are only few key concepts behind it.

1. Components do not know each other directly

2. Components specify external dependencies using some sort of a key.
3. Dependencies are resolved late, preferably just before they are used (JIT dependency resolution).
4. Dependencies are resolved once for each component.

You guessed it - it should not be *such* a big deal to do this in python!

And indeed, a combination of a broker, descriptors and lazy attributes brings about pretty much the same core result as those IoC containers - effectively in little more then 50 lines (not counting demo code, comments and empy lines).

So what does the code do?

- It offers a mechanism to register provided "features".
- It offers a mechanism to "declare" required features in a readable way as attributes.
- The required features are resolved (injected) as late as possible - at access time.
- It provides for reasonable verification of injected dependencies.

The supported injection type is "Setter Injection", which basically means that dependencies are expressed through attributes. There is another type of injection, the "Constructor Injection", but that one builds heavily on static typing and can therefore not be employed in python (or at least I could not think of any elegant way to do it).