Roles Documentation

Release 0.9

Arjan Molenaar

Jul 12, 2023

CONTENTS

Library for Role based development

`Roles` provides a Pythonic implementation of the DCI (Data Context Interaction) paradigm (http://www.artima.com/articles/dci_vision.html).

Roles allow you to assign and revoke behaviour on a per-instance basis. This defines the big difference with mixins, which are assigned at class level.

A role has a special meaning in a context (imagine you want to do a money transfer: in this context you'll need 2 accounts, a source and a destination account). The roles module provides a simple implementation for defining contexts.

Roles can be assigned and revoked. Multiple roles can be applied to an instance. Revocation can happen in any particular order.

Homepage:
http://amolenaar.github.com/roles

Sources:
http://github.com/amolenaar/roles

Downloads:
http://pypi.python.org/pypi/roles

Contents:

CHAPTER

ONE

USAGE

1.1 Using Roles

```
>>> from roles import RoleType
```

As a basic example, consider a data class:

```
>>> class Person:
...     def __init__(self, name):
...         self.name = name
>>> person = Person("John")
```

The instance should participate in a collaboration in which it fulfills a particular role:

```
>>> class Carpenter(metaclass=RoleType):
...     def chop(self):
...         return "chop, chop"
```

Assign the role to the person:

```
>>> Carpenter(person)
<__main__.Person+Carpenter object at 0x...>
>>> person
<__main__.Person+Carpenter object at 0x...>
```

The person is still a Person:

```
>>> isinstance(person, Person)
True
```

… and can do carpenter things:

```
>>> person.chop()
'chop, chop'
```

To revoke a role from an instance do

```
>>> Carpenter.revoke(person)
<__main__.Person object at 0x...>
```

It is much more convenient, though, to apply roles only in a specific context:

```
>>> from roles.context import context
```

```
>>> class WoodChopping:
...     def __init__(self, person):
...         self.person = person
...         self.chopping_context = context(self, person=Carpenter)
...
...     def __call__(self):
...         with self.chopping_context:
...             return self.person.chop()
...
>>> ctx = WoodChopping(person)
>>> ctx()
'chop, chop'
```

1.2 Using Context

Roles make a lot of sense when used in a context. A classic example is the money transfer example. Here two accounts are used and an amount of money is transfered from one account to the other. So, one account playes the role of source account and the other plays the role of target account.

```
>>> from roles.context import context
```

Say we have this simple account class:

```
>>> class Account:
...
...     def __init__(self, amount):
...         self.balance = amount
...         super(Account, self).__init__()
...
...     def withdraw(self, amount):
...         self.balance -= amount
...
...     def deposit(self, amount):
...         self.balance += amount
```

If you want to transfer money from one account to another, we're normally calling one the source account and the other the destination account. Let's make that clear in code:

```
>>> class MoneySource(metaclass=RoleType):
...
...     def transfer(self, amount):
...         if self.balance >= amount:
...             self.withdraw(amount)
...             context.sink.receive(amount)
...
>>> class MoneySink(metaclass=RoleType):
...
...     def receive(self, amount):
...         self.deposit(amount)
```

During the act of a money transfer (what do you think about when money is transfered?), some amount is transfered from one account to the other. That is made explicit in a context.

A context contains the objects that are required for some action and assign them the roles they will play during the enactment. Roles have no meaning outside a context, since the logic executed is specific to this use case. Advantage of using a context is that it is not required to pass role objects around. Also contexts can be used as intermediate data store (for the duration of the context): note how the money source is finding its destination through the context.

```
>>> class TransferMoney:
...
...     def __init__(self, source, sink):
...         self.source = source
...         self.sink = sink
...         self.transfer_context = context(self,
...                     source=MoneySource,
...                     sink=MoneySink)
...
...     def transfer_money(self, amount):
...         """
...         The interaction.
...         """
...         with self.transfer_context as ctx:
...             ctx.source.transfer(amount)
```

Another way is to make the method itself act as context (well, the first argument is considered the context object:

```
>>> from roles.context import in_context
```

```
>>> class TransferMoney:
...
...     def __init__(self, source, sink):
...         self.source = source
...         self.sink = sink
...
...     @in_context
...     def transfer_money(self, amount):
...         """
...         The interaction.
...         """
...         with MoneySource.played_by(self.source),\
...                     MoneySink.played_by(self.sink):
...             self.source.transfer(amount)
```

CHAPTER

TWO

USER API

2.1 RoleType metaclass

class roles.RoleType

RoleType is a metaclass that provides role support to classes. The initialization process has been altered to provide addition and removal of roles.

It starts with a normal class:

```
>>> class Person:
...     def __init__(self, name): self.name = name
...     def am(self): print(self.name, 'is')
```

Apart from that a few roles can be defined. Simple objects with a default `__init__()` (no arguments) and the `RoleType` as metaclass:

```
>>> class Carpenter(metaclass=RoleType):
...     def chop(self): print(self.name, 'chops')
>>> class Biker(metaclass=RoleType):
...     def bike(self): print(self.name, 'bikes')
```

Now, by default an object has no roles (in this case our person).

```
>>> person = Person('Joe')
```

Roles can be added by calling the `assign()` method:

```
>>> Carpenter.assign(person)
<roles.role.Person+Carpenter object at 0x...>
```

Or by calling the role on the subject:

```
>>> Carpenter(person)
<roles.role.Person+Carpenter object at 0x...>
```

The persons methods can be invoked:

```
>>> person.am()
Joe is
```

As well as the role's methods:

```
>>> person.chop()
Joe chops
```

The default behaviour is to apply the role directly to the instance.

```
>>> person
<roles.role.Person+Carpenter object at 0x...>
```

The module contains a function `clone()` that can be provided to the `asign()` method to create proxy instances (the default function is called `instance()` and can also be found in this module):

```
>>> biker = Biker.assign(person, method=clone)
>>> biker
<roles.role.Person+Carpenter+Biker object at 0x...>
>>> biker is person
False
```

Objects can contain multiple roles:

```
>>> biker = Biker.assign(person)
>>> biker
<roles.role.Person+Carpenter+Biker object at 0x...>
>>> biker.__class__.__bases__
(<class 'roles.role.Person'>, <class 'roles.role.Carpenter'>, <class 'roles.role.
↪Biker'>)
```

Note that a new class is assigned, with the roles applied (roles first).

Roles can be revoked:

```
>>> Carpenter.revoke(biker)
<roles.role.Person+Biker object at 0x...>
>>> biker.__class__.__bases__
(<class 'roles.role.Person'>, <class 'roles.role.Biker'>)
```

Revoking a non-existant role has no effect:

```
>>> Carpenter.revoke(biker)
<roles.role.Person+Biker object at 0x...>
```

Roles do not allow for overriding methods.

```
>>> class Incognito(metaclass=RoleType):
...     def am(self): return 'under cover'
>>> Incognito(Person)
Traceback (most recent call last):
  ...
TypeError: Can not apply role when overriding methods: am
```

Caching

One more thing: role classes are cached. This means that if I want to assign a role to a different instance, the same role class is applied:

```
>>> person = Person('Joe')
>>> someone = Person('Jane')
>>> Biker(someone).__class__ is Biker(person).__class__
True
```

Changing role application

If for some reason the role should not be directly applied to the instance, another application method can be assigned.

Here is an example that uses the `clone` method:

```
>>> person = Person('Joe')
>>> person.__class__
<class 'roles.role.Person'>
>>> biker = Biker(person, method=clone)
>>> biker
<roles.role.Person+Biker object at 0x...>
>>> person.__class__
<class 'roles.role.Person'>
>>> biker.bike()
Joe bikes
```

assign(*subj: ~roles.role.T, method: ~typing.Callable[[~typing.Type[~roles.role.R], ~roles.role.T], ~roles.role.R] = <function instance>*) → Union[T, R]

Call is invoked when a role should be assigned to an object.

played_by(*subj: T*) → Iterator[Union[T, R]]

Shorthand for using roles in with statements.

```
>>> class Biker(metaclass=RoleType):
...     def bike(self): return 'bike, bike'
>>> class Person:
...     pass
>>> john = Person()
>>> with Biker.played_by(john):
...     john.bike()
'bike, bike'
```

revoke(*subj: ~roles.role.R, method: ~typing.Callable[[~typing.Type[~roles.role.T], ~roles.role.R], ~roles.role.T] = <function instance>*) → T

Retract the role from subj.

By default the `instance` strategy is used.

2.2 Using roles in a context

Roles are played in a context. The `roles.context` module provides a means to access the context from within your roles. Use this to make your role's code simpler and more readable.

`roles.context.`**`context`**`(`*ctxobj*, ***bindings*`)`

The default application wide context stack.

Put a new context class on the context stack. This functionality should be called with the context class as first argument.

```
>>> class SomeContext:
...     pass # define some methods, define some roles
...     def execute(self):
...         with context(self):
...             pass # do something
```

Roles can be fetched from the context by calling `context.name`. Just like that.

You can provide additional bindings to be performed:

```
>>> from roles.role import RoleType
```

```
>>> class SomeRole(metaclass=RoleType):
...     pass
```

```
>>> class SomeContext:
...     def __init__(self, data_object):
...         self.data_object = data_object
...     def execute(self):
...         with context(self, data_object=SomeRole):
...             pass # do something
```

Those bindings are applied when the context is entered (in this case immediately).

`roles.context.`**`in_context`**`(`*func*`)`

Decorator for running methods in context.

The context is the object (self).

2.3 Ways to assign roles

There are basically 3 ways to assign a role to an instance. The first one is to manipulate the instance's class (this is the default) and the second is to proxy the object by referencing the same instance dict (Borg pattern). The this (last resort) is to create an adapter on top of the data instance.

2.4 Generic roles

As an (non-DCI) extension it is possible to create role implementations tailored for specific classes.

Although this may clutter the clear and readable ways provided by DCI, for specific tasks it may help. Use it wisely.

CHAPTER

THREE

DJANGO SUPPORT

Django is a popular web framework written in Python. Using DCI (or just roles) in Django is quite easy, but since Django is using some meta classes of its own, a few things have to be taken into account.

First of all, roles that need to be applied to model classes (which is most obvious) should use `roles.django.ModelRoleType` as metaclass.

```
>>> from roles.django import ModelRoleType
```

Then roles can be applied in the regular way.

```
>>> class MoneySource(metaclass=ModelRoleType):
...
...     def transfer(self, amount):
...         if self.balance >= amount:
...             self.withdraw(amount)
...             context.sink.receive(amount)
```

Since the roles module changes the class names (it adds the roles that are applied at a certain time), those can not be used directly for storing. For now, saving objects should be done outside the role context.

```
>>> class TransferMoney:
...
...     def __init__(self, source, sink):
...         self.source = source
...         self.sink = sink
...
...     def transfer_money(self, amount):
...         """
...         The interaction.
...         """
...         with context(self,
...                      source=MoneySource,
...                      sink=MoneySink):
...             # Let roles interact, in context
...             self.source.transfer(amount)
...         # Do storage when roles are removed
...         self.source.save()
...         self.sink.save()
```

CHAPTER

FOUR

PRIVATE API

For those interested in (non-public) API.

4.1 Utility classes

roles.role.**class_fields**(*cls: Type*, *exclude: Sequence[str] = ('__doc__', '__module__', '__dict__', '__weakref__', '__slots__')*) → Set[str]

Get all fields declared in a class, including superclasses.

Don't forget to clear the cache if fields are added to a class or role!

4.2 Context internals

class roles.context.**CurrentContextManager**

The default application wide context stack.

Put a new context class on the context stack. This functionality should be called with the context class as first argument.

```
>>> class SomeContext:
...     pass # define some methods, define some roles
...     def execute(self):
...         with context(self):
...             pass # do something
```

Roles can be fetched from the context by calling `context.name`. Just like that.

You can provide additional bindings to be performed:

```
>>> from roles.role import RoleType
```

```
>>> class SomeRole(metaclass=RoleType):
...     pass
```

```
>>> class SomeContext:
...     def __init__(self, data_object):
...         self.data_object = data_object
...     def execute(self):
```

(continues on next page)

(continued from previous page)

```
...         with context(self, data_object=SomeRole):
...             pass # do something
```

Those bindings are applied when the context is entered (in this case immediately).

CHAPTER

FIVE

CHANGE HISTORY

1.0.0

- Python 3.8+ only
- Removed `roles.factory`, use *functools.singledispatch*` instead
- Added type annotations
- Publish docs on readthedocs.io
- Build with Poetry
- CI on Github Actions

0.10

- Allow saving domain instances with roles applied. Thanks to Ben Scherrey and Chokchai Phatharamalai
- Deal with `__slots__`
- Can assign roles bindings to be used in the Context

0.9

- It is no longer allowed to let roles override methods.
- Added module roles.django
- Added Django application example (django_dci module)
- context is thread safe
- added adapter method for role assignment

0.8

- Removed @*rolecontext*. Seems not such a good idea.
- separated code in multiple modules
- Added DCI *context* management classes.
- Added warnings for using the factory functionality in a DCI context

0.7.0

- Added @*rolecontext* decorator to ensure roles are applied on function invocation.

0.6.0

- RoleType.played_by for easy use with *with* statement.
- removed roles function and psyco optimizations.

- bug fixes and performance updates

0.5.0

- Support for contexts (*with* statement).
- revoke on factories now works as expected.

0.4.0

- Make the way a role is applied to an object pluggable. This means you can either apply the role to the original instance or create a clone, using the original instance dict.

0.3.0

- Module works for Python 2.6 as well as Python 3.x. Doctests still run under 2.6.

0.2.0

- Added psyco_optimize() for optimizing code with psyco.

0.1.0

- Initial release: roles.py

CHAPTER

SIX

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

INDEX