Dectate Documentation

Release 0.7

Martijn Faassen

April 11, 2016

Contents

Dectate is a Python library that lets you construct a decorator-based configuration system for frameworks. Configuration is associated with class objects. It supports configuration inheritance and overrides as well as conflict detection.

CHAPTER 1

Using Dectate

1.1 Introduction

Dectate is a configuration system that can help you construct Python frameworks. A framework needs to record some information about the functions and classes that the user supplies. We call this process *configuration*.

Imagine for instance a framework that supports a certain kind of plugins. The user registers each plugin with a decorator:

```
from framework import plugin

@plugin(name="foo")
def foo_plugin(...):
    ...
```

Here the framework registers as a plugin the function `foo_plugin` under the name `foo`.

You can implement the `plugin` decorator as follows:

```
plugins = {}

class plugin(name):
    def __init__(self, name):
        self.name = name

    def __call__(self, f):
       plugins[self.name] = f
```

In the user application the user makes sure to import all modules that use the `plugin` decorator. As a result, the `plugins` dict contains the names as keys and the functions as values. Your framework can then use this information to do whatever you need to do.

There are a lot of examples of code configuration in frameworks. In a web framework for instance the user can declare routes and assemble middleware.

You may be okay constructing a framework with the simple decorator technique described above. But advanced frameworks need a lot more that the basic decorator system described above cannot offer. You may for instance want to allow the user to reuse configuration, override it, do more advanced error checking, and execute configuration in a particular order.

Dectate supports such advanced use cases. It was extracted from the Morepath web framework.

1.2 Features

Here are some features of Dectate:

- Decorator-based configuration – users declare things by using Python decorators on functions and classes: we call these decorators *directives*, which issue configuration *actions*.
- Dectate detects conflicts between configuration actions in user code and reports what pieces of code are in conflict.
- Users can easily reuse and extend configuration: it's just Python class inheritance.
- Users can easily override configurations in subclasses.
- You can compose configuration actions from other, simpler ones.
- You can control the order in which configuration actions are executed. This is unrelated to where the user uses the directives in code. You do this by declaring *dependencies* between types of configuration actions, and by *grouping* configuration actions together.
- You can declare exactly what objects are used by a type of configuration action to register the configuration – different types of actions can use different registries.
- Unlike normal decorators, configuration actions aren't performed immediately when a module is imported. Instead configuration actions are executed only when the user explicitly *commits* the configuration. This way, all configuration actions are known when they are performed.
- Dectate-based decorators always return the function or class object that is decorated unchanged, which makes the code more predictable for a Python programmer – the user can use the decorated function or class directly in their Python code, just like any other.
- Dectate-based configuration systems are themselves easily extensible with new directives and registries.
- Dectate-based configuration systems can be queried. Dectate also provides the infrastructure to easily construct command-line tools for querying configuration.

1.3 App classes

Configuration in Dectate is associated with special *classes* which derive from `dectate.App`:

```
import dectate

class MyApp(dectate.App):
    pass
```

1.4 Creating a directive

We can now use the `dectate.App.directive()` decorator to declare a *directive* which executes a special configuration action. Let's replicate the simple *plugins* example above using Dectate:

```
@MyApp.directive('plugin')
class PluginAction(dectate.Action):
    config = {
       'plugins': dict
    }
    def __init__(self, name):
```

```
        self.name = name

    def identifier(self, plugins):
        return self.name

    def perform(self, obj, plugins):
        plugins[self.name] = obj
```

Let's use it now:

```
@MyApp.plugin('a')
def f():
    pass # do something interesting

@MyApp.plugin('b')
def g():
    pass # something else interesting
```

We have registered the function `f` on `MyApp`. The `name` argument is `'a'`. We've registered `g` under `'b'`.

We can now commit the configuration for `MyApp`:

```
dectate.commit(MyApp)
```

Once the commit has successfully completed, we can take a look at the configuration:

```
>>> sorted(MyApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>)]
```

What are the changes between this and the simple plugins example?

The main difference is that `plugin` decorator is associated with a class and so its the resulting configuration. The other difference is that we provide an `identifier` method in the action definition. These differences support configuration *reuse*, *conflicts*, *extension*, *overrides* and *isolation*.

1.4.1 Reuse

You can reuse configuration by simply subclassing `MyApp`:

```
class SubApp(MyApp):
   pass
```

We commit both classes:

```
dectate.commit(MyApp, SubApp)
```

`SubClass` now contains all the configuration declared for `MyApp`:

```
>>> sorted(SubApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>)]
```

So class inheritance lets us reuse configuration, which allows *extension* and *overrides*, which we discuss below.

1.4.2 Conflicts

Consider this example:

```
class ConflictingApp(MyApp):
    pass

@ConflictingApp.plugin('foo')
def f():
    pass

@ConflictingApp.plugin('foo')
def g():
    pass
```

Which function should be registered for `foo`, `f` or `g`? We should refuse to guess and instead raise an error that the configuration is in conflict. This is exactly what Dectate does:

```
>>> dectate.commit(ConflictingApp)
Traceback (most recent call last):
  ...
ConflictError: Conflict between:
 File "...", line 4
   @ConflictingApp.plugin('foo')
 File "...", line 8
   @ConflictingApp.plugin('foo')
```

As you can see, Dectate reports the lines in which the conflicting configurations occurs.

How does Dectate know that these configurations are in conflict? This is what the `identifier` method in our action definition did:

```
def identifier(self, plugins):
    return self.name
```

We say here that the configuration is uniquely identified by its `name` attribute. If two configurations exist with the same name, the configuration is considered to be in conflict.

1.4.3 Extension

When you subclass configuration, you can also *extend* `SubApp` with additional configuration actions:

```
@SubApp.plugin('c')
def h():
    pass # do something interesting

dectate.commit(MyApp, SubApp)
```

`SubApp` now has the additional plugin `c`:

```
>>> sorted(SubApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>), ('c', <function h at ...>)]
```

But `MyApp` is unaffected:

```
>>> sorted(MyApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>)]
```

1.4.4 Overrides

What if you wanted to override a piece of configuration? You can do this in `SubApp` by simply reusing the same `name`:

```
@SubApp.plugin('a')
def x():
    pass

dectate.commit(MyApp, SubApp)
```

In `SubApp` we now have changed the configuration for `a` to register the function `x` instead of `f`. If we had done this for `MyApp` this would have been a conflict, but doing so in a subclass lets you override configuration instead:

```
>>> sorted(SubApp.config.plugins.items())
[('a', <function x at ...>), ('b', <function g at ...>), ('c', <function h at ...>)]
```

But `MyApp` still uses `f`:

```
>>> sorted(MyApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>)]
```

1.4.5 Isolation

We have already seen in the inheritance and override examples that `MyApp` is isolated from configuration extension and overrides done for `SubApp`. We can in fact entirely isolate configuration from each other.

We first set up a new base class with a directive, independently from everything before:

```
class BaseApp(dectate.App):
    pass

@BaseApp.directive('plugin')
class PluginAction2(dectate.Action):
    config = {
       'plugins': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, plugins):
        return self.name

    def perform(self, obj, plugins):
        plugins[self.name] = obj
```

We don't set up any configuration for `BaseApp`; it's intended to be part of our framework. Now we create two subclasses:

```
class OneApp(BaseApp):
    pass

class TwoApp(BaseApp):
    pass
```

As you can see `OneApp` and `TwoApp` are completely isolated from each other; the only thing they share is a common `BaseApp`.

We register a plugin for `OneApp`:

```
@OneApp.plugin('a')
def f():
    pass
```

This won't affect `TwoApp` in any way:

```
dectate.commit(OneApp, TwoApp)
```

```
>>> sorted(OneApp.config.plugins.items())
[('a', <function f at ...>)]
>>> sorted(TwoApp.config.plugins.items())
[]
```

`OneApp` and `TwoApp` are isolated, so configurations are independent, and cannot conflict or override.

1.5 The Anatomy of a Directive

Let's consider the directive registration again in detail:

```
@MyApp.directive('plugin')
class PluginAction(dectate.Action):
    config = {
       'plugins': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, plugins):
        return self.name

    def perform(self, obj, plugins):
        plugins[self.name] = obj
```

What is going on here?

- We create a new directive called `plugin` on `MyApp`. It also exists for its subclasses.
- The directive is implemented with a custom class called `PluginAction` that inherits from *`dectate.Action`*.
- `config` (*`dectate.Action.config`*) specifies that this directive has a configuration effect on `plugins`. We declare that `plugins` is created using the `dict` factory, so our registry is a plain dictionary. You provide any factory function you like here.
- `_init_` specifies the parameters the directive should take and how to store them on the action object. You can use default parameters and such, but otherwise `__init__` should be very simple and not do any registration or validation. That logic should be in `perform`.
- `identifier` (*`dectate.Action.identifier()`*) takes the configuration objects specified by `config` as keyword arguments. It returns an immutable that is unique for this action. This is used to detect conflicts and determine how configurations override each other.
- `perform` (*`dectate.Action.perform()`*) takes `obj`, which is the function or class that the decorator is used on, and the arguments specified in `config`. It should use `obj` and the information on `self` to configure the configuration objects. In this case we store `obj` under the key `self.name` in the `plugins` dict.

Once we have declared the directive for our framework we can tell programmers to use it.

Directives have absolutely no effect until *commit* is called, which we do with `dectate.commit`. This performs the actions and we can then find the result `MyApp.config`.

The results are in `MyApp.config.plugins` as we set this up with `config` in our `PluginAction`.

1.6 Depends

In some cases you want to make sure that one type of directive has been executed before the other – the configuration of the second type of directive depends on the former. You can make sure this happens by using the `depends` (*dectate.Action.depends*) class attribute.

First we set up a `foo` directive that registers into a `foos` dict:

```
class DependsApp(dectate.App):
    pass

@DependsApp.directive('foo')
class FooAction(dectate.Action):
    config = {
       'foos': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos[self.name] = obj
```

Now we create a `bar` directive that depends on `FooDirective` and uses information in the `foos` dict:

```
@DependsApp.directive('bar')
class BarAction(dectate.Action):
   depends = [FooAction]

   config = {
      'foos': dict,  # also use the foos dict
      'bars': list
   }
   def __init__(self, name):
       self.name = name

   def identifier(self, foos, bars):
       return self.name

   def perform(self, obj, foos, bars):
       in_foo = self.name in foos
       bars.append((self.name, obj, in_foo))
```

We have now ensured that `BarAction` actions are performed after `FooAction` action, no matter what order we use them:

```
@DependsApp.bar('a')
def f():
    pass

@DependsApp.bar('b')
```

```
def g():
    pass

@DependsApp.foo('a')
def x():
    pass

dectate.commit(DependsApp)
```

We expect `in_foo` to be `True` for `a` but to be `False` for `b`:

```
>>> DependsApp.config.bars
[('a', <function f at ...>, True), ('b', <function g at ...>, False)]
```

1.7 config dependencies

In the example above, the items in `bars` depend on the items in `foos` and we've implemented this dependency in the `perform` of `BarDirective`.

We can instead make the configuration object for the `BarDirective` depend on `foos`. This way `BarDirective` does not need to know about `foos`. You can declare a dependency between config objects with the `factory_arguments` attribute of the config factory. Any config object that is created in earlier dependencies of this action, or in the action itself, can be listed in `factory_arguments`. The key and value in `factory_arguments` have to match the key and value in `config` of that earlier action.

First we create an app with a `FooAction` that sets up a `foos` config item as before:

```
class ConfigDependsApp(dectate.App):
    pass

@ConfigDependsApp.directive('foo')
class FooAction(dectate.Action):
    config = {
       'foos': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos[self.name] = obj
```

Now we create a `Bar` class that also depends on the `foos` dict by listing it in `factory_arguments`:

```
class Bar(object):
    factory_arguments = {
        'foos': dict
    }

    def __init__(self, foos):
        self.foos = foos
        self.l = []

    def add(self, name, obj):
```

```
        in_foo = name in self.foos
        self.l.append((name, obj, in_foo))
```

We create a `BarAction` that depends on the `FooAction` (so that `foos` is created first) and that uses the `Bar` factory:

```
@ConfigDependsApp.directive('bar')
class BarAction(dectate.Action):
   depends = [FooAction]

   config = {
      'bar': Bar
   }

   def __init__(self, name):
       self.name = name

   def identifier(self, bar):
       return self.name

   def perform(self, obj, bar):
       bar.add(self.name, obj)
```

When we use our directives:

```
@ConfigDependsApp.bar('a')
def f():
    pass

@ConfigDependsApp.bar('b')
def g():
    pass

@ConfigDependsApp.foo('a')
def x():
    pass

dectate.commit(ConfigDependsApp)
```

we get the same result as before:

```
>>> ConfigDependsApp.config.bar.l
[('a', <function f at ...>, True), ('b', <function g at ...>, False)]
```

1.8 before and after

It can be useful to do some additional setup just before all actions of a certain type are performed, or just afterwards. You can do this using `before` (*`dectate.Action.before()`*) and `after` (*`dectate.Action.after()`*) static methods on the Action class:

```
class BeforeAfterApp(dectate.App):
    pass

@BeforeAfterApp.directive('foo')
class FooAction(dectate.Action):
    config = {
       'foos': list
```

```
    }
    def __init__(self, name):
        self.name = name

    @staticmethod
    def before(foos):
        print "before:", foos

    @staticmethod
    def after(foos):
        print "after:", foos

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos.append((self.name, obj))

@BeforeAfterApp.foo('a')
def f():
    pass

@BeforeAfterApp.foo('b')
def g():
    pass
```

This executes `before` just before `a` and `b` are configured, and then executes `after`:

```
>>> dectate.commit(BeforeAfterApp)
before: []
after: [('a', <function f at ...>), ('b', <function g at ...>)]
```

1.9 grouping actions

Different actions normally don't conflict with each other. It can be useful to group different actions together in a group so that they do affect each other. You can do this with the `group_class` (*dectate.Action.group_class*) class attribute. Grouped classes share their `config` and their `before` and `after` methods.

```
class GroupApp(dectate.App):
    pass

@GroupApp.directive('foo')
class FooAction(dectate.Action):
    config = {
       'foos': list
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos.append((self.name, obj))
```

We now create a `BarDirective` that groups with `FooAction`:

```
@GroupApp.directive('bar')
class BarAction(dectate.Action):
   group_class = FooAction

   def __init__(self, name):
       self.name = name

   def identifier(self, foos):
       return self.name

   def perform(self, obj, foos):
       foos.append((self.name, obj))
```

It reuses the `config` from `FooAction`. This means that `foo` and `bar` can be in conflict:

```
class GroupConflictApp(GroupApp):
    pass

@GroupConflictApp.foo('a')
def f():
    pass

@GroupConflictApp.bar('a')
def g():
    pass
```

```
>>> dectate.commit(GroupConflictApp)
Traceback (most recent call last):
  ...
ConflictError: Conflict between:
  File "...", line 8
    @GroupConflictApp.bar('a')
```

1.10 Additional discriminators

In some cases an action should conflict with *multiple* other actions all at once. You can take care of this with the `discriminators` (*dectate.Action.discriminators()*) method on your action:

```
class DiscriminatorsApp(dectate.App):
    pass

@DiscriminatorsApp.directive('foo')
class FooAction(dectate.Action):
    config = {
       'foos': dict
    }
    def __init__(self, name, extras):
        self.name = name
        self.extras = extras

    def identifier(self, foos):
        return self.name

    def discriminators(self, foos):
        return self.extras
```

```
    def perform(self, obj, foos):
        foos[self.name] = obj
```

An action now conflicts with an action of the same name *and* with any action that is in the `extra` list:

```
# example
@DiscriminatorsApp.foo('a', ['b', 'c'])
def f():
    pass

@DiscriminatorsApp.foo('b', [])
def g():
    pass
```

And then:

```
>>> dectate.commit(DiscriminatorsApp)
Traceback (most recent call last):
  ...
ConflictError: Conflict between:
  File "...", line 2:
    @DiscriminatorsApp.foo('a', ['b', 'c'])
  File "...", line 6
    @DiscriminatorsApp.foo('b', [])
```

1.11 Composite actions

When you can define an action entirely in terms of other actions, you can subclass *dectate.Composite*.

First we define a normal `sub` directive to use in the composite action later:

```
class CompositeApp(dectate.App):
    pass

@CompositeApp.directive('sub')
class SubAction(dectate.Action):
    config = {
        'my': list
    }

    def __init__(self, name):
        self.name = name

    def identifier(self, my):
        return self.name

    def perform(self, obj, my):
        my.append((self.name, obj))
```

Now we can define a special *dectate.Composite* subclass that uses `SubAction` in an `actions` (*dectate.Composite.actions()*) method:

```
@CompositeApp.directive('composite')
class CompositeAction(dectate.Composite):
    def __init__(self, names):
        self.names = names
```

```
    def actions(self, obj):
        return [(SubAction(name), obj) for name in self.names]
```

We can now use it:

```
@CompositeApp.composite(['a', 'b', 'c'])
def f():
    pass

dectate.commit(CompositeApp)
```

And `SubAction` is performed three times as a result:

```
>>> CompositeApp.config.my
[('a', <function f at ...>), ('b', <function f at ...>), ('c', <function f at ...>)]
```

1.12 `with` statement

Sometimes you want to issue a lot of similar actions at once. You can use the `with` statement to do so with less repetition:

```
class WithApp(dectate.App):
    pass

@WithApp.directive('foo')
class SubAction(dectate.Action):
    config = {
        'my': list
    }

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def identifier(self, my):
        return (self.a, self.b)

    def perform(self, obj, my):
        my.append((self.a, self.b, obj))
```

Instead of this:

```
class VerboseWithApp(WithApp):
    pass

@VerboseWithApp.foo('a', 'x')
def f():
   pass

@VerboseWithApp.foo('a', 'y')
def g():
   pass

@VerboseWithApp.foo('a', 'z')
def h():
   pass
```

You can instead write:

```
class SuccinctWithApp(WithApp):
    pass

with SuccinctWithApp.foo('a') as foo:
    @foo('x')
    def f():
        pass

    @foo('y')
    def g():
        pass

    @foo('z')
    def h():
        pass
```

And this has the same configuration effect:

```
>>> dectate.commit(VerboseWithApp, SuccinctWithApp)
>>> VerboseWithApp.config.my
[('a', 'x', <function f at ...>), ('a', 'y', <function g at ...>), ('a', 'z', <function h at ...>)]
>>> SuccinctWithApp.config.my
[('a', 'x', <function f at ...>), ('a', 'y', <function g at ...>), ('a', 'z', <function h at ...>)]
```

1.13 importing recursively

When you use dectate-based decorators across a package, it can be useful to just import *all* modules in it at once. This way the user cannot forget to import a module with decorators in it.

Dectate itself does not offer this facility, but you can use the importscan library to do this recursive import. Simply do something like:

```
import my_package

importscan.scan(my_package, ignore=['.tests'])
```

This imports every module in `my_package`, except for the `tests` sub package.

1.14 logging

Dectate logs information about the performed actions as debug log messages. By default this goes to the `dectate.directive.<directive_name>` log. You can use the standard Python `logging` module function to make this information go to a log file.

If you want to override the name of the log you can set `logger_name` (*`dectate.App.logger_name`*) on the app class:

```
class MorepathApp(dectate.App):
   logger_name = 'morepath.directive'
```

1.15 querying

Dectate keeps a database of committed actions that can be queried by using *dectate.Query*.

Here is an example of a query for all the plugin actions on `MyApp`:

```
q = dectate.Query('plugin')
```

We can now run the query:

```
>>> list(q(MyApp))
[(<PluginAction ...>, <function f ...>),
 (<PluginAction ...>, <function g ...>)]
```

We can also filter the query for attributes of the action:

```
>>> list(q.filter(name='a')(MyApp))
[(<PluginAction object ...>, <function f ...>)]
```

Sometimes the attribute on the action is not the same as the name you may want to use in the filter. You can use *dectate.Action.filter_name* to create a mapping to the correct attribute.

By default the filter does an equality comparison. You can define your own comparison function for an attribute using *dectate.Action.filter_compare*.

If you want to allow a query on a `Composite` action you need to give it some help by defining *dectate.Composite.query_classes*.

1.16 query tool

Dectate also includes a command-line tool that lets you issue queries. You need to configure it for your application. For instance, in the module `main.py` of your project:

```
import dectate

def query_tool():
    dectate.commit(SomeApp)
    dectate.query_tool([SomeApp])
```

In this function you should commit any *dectate.App* subclasses your application normally uses, and then provide a list of them to *dectate.query_tool()*. This is the list of applications that is queried by default if you don't specify `--app`.

Then in `setup.py` of your project:

```
entry_points={
    'console_scripts': [
        'decq = query.main:query_tool',
    ]
},
```

When you re-install this project you have a command-line tool called `decq` that lets you issues queries. For instance, this query returns all uses of directive `foo` in the apps you provided to `query_tool`:

```
$ decq foo
App: <class 'query.a.App'>
  File ".../query/b.py", line 4
  @App.foo(name='alpha')
```

```
  File ".../query/b.py", line 9
  @App.foo(name='beta')

  File ".../query/b.py", line 14
  @App.foo(name='gamma')

  File ".../query/c.py", line 4
  @App.foo(name='lah')

App: <class 'query.a.Other'>
  File ".../query/b.py", line 19
  @Other.foo(name='alpha')
```

And this query filters by `name`:

```
$ decq foo name=alpha
App: <class 'query.a.App'>
  File ".../query/b.py", line 4
  @App.foo(name='alpha')

App: <class 'query.a.Other'>
  File ".../query/b.py", line 19
  @Other.foo(name='alpha')
```

You can also explicit provide the app classes to query with the `--app` option; the default list of app classes is ignored in this case:

```
$ bin/decq --app query.a.App foo name=alpha
App: <class 'query.a.App'>
  File ".../query/b.py", line 4
  @App.foo(name='alpha')
```

You need to give `--app` a dotted name of the *`dectate.App`* subclass to query. You can repeat the `--app` option to query multiple apps.

Not all things you would wish to query on are string attributes. You can provide a conversion function that takes the string input and converts it to the underlying object you want to compare to using *`dectate.Action.filter_convert`*.

A working example is in `scenarios/query` of the Dectate project.

1.17 Sphinx Extension

If you use Sphinx to document your project and you use the `sphinx.ext.autodoc` extension to document your API, you need to install a Sphinx extension so that directives are documented properly. In your Sphinx `conf.py` add `'dectate.sphinxext'` to the `extensions` list.

1.18 `__main__` and conflicts

Import-time side effects are evil

This scenario is based on the one described in Application programmers don't control the module-scope codepath in the Pyramid design defense document. If you're curious, look under `scenarios/main_module` in the Dectate project for a Dectate version.
Dectate makes a different compromise than Venusian – it reports an error if a directive is executed because of a double import, so it won't get you into trouble. But since Dectate's directives cause registrations to happen immediately (but defer configuration), you can dynamically generate them inside Python function, which won't work with with Venusian.

In certain scenarios where you run your code like this:

```
$ python app.py
```

and you use `__name__ == '__main__'` to determine whether the module should run:

```
if __name__ == '__main__':
    import another_module
    dectate.commit(App)
```

you might get a `ConflictError` from Dectate that looks somewhat like this:

```
Traceback (most recent call last):
 ...
dectate.error.ConflictError: Conflict between:
  File "/path/to/app.py", line 6
    @App.foo(name='a')
  File "app.py", line 6
    @App.foo(name='a')
```

The same line shows up on *both* sides of the configuration conflict, but the path is absolute on one side and relative on the other.

This happens because in some scenarios involving `__main__`, Python imports a module *twice* (more about this). Dectate refuses to operate in this case until you change your imports so that this doesn't happen anymore.

How to avoid this scenario? If you use setuptools automatic script creation this problem is avoided entirely.

Fooling Dectate after all

It *is* possible to fool Dectate into accepting a double import without conflicts, but you'd need to work hard. You need to use a global variable that gets modified during import time and then use it as a directive argument. If you want to dynamically generate directives then don't do that in module-scope – do it in a function.

If you want to use the `if __name__ == '__main__'` system, keep your main module tiny and just import the main function you want to run from elsewhere.

So, Dectate warns you if you do it wrong, so don't worry about it.

API

`dectate.commit(*apps)`

Commit one or more app classes

A commit causes the configuration actions to be performed. The resulting configuration information is stored under the `.config` class attribute of each `App` subclass supplied.

This function may safely be invoked multiple times – each time the known configuration is recommitted.

Parameters `*apps` – one or more `App` subclasses to perform configuration actions on.

`dectate.autocommit()`

Automatically commit all `App` subclasses.

Dectate keeps track of all `App` subclasses that have been imported. You can automatically commit configuration for all of them.

class `dectate.App`

A configurable application object.

Subclass this in your framework and add directives using the `App.directive()` decorator.

Set the `logger_name` class attribute to the logging prefix that Dectate should log to. By default it is `"dectate.directive"`.

classmethod `directive(name)`

Decorator to register a new directive with this application class.

You use this as a class decorator for a `dectate.Action` or a `dectate.Composite` subclass:

```
@MyApp.directive('my_directive')
class FooAction(dectate.Action):
    ...
```

This needs to be executed *before* the directive is used and thus might introduce import dependency issues unlike normal Dectate configuration, so beware! An easy way to make sure that all directives are installed before you use them is to make sure you define them in the same module as where you define the `App` subclass that has them.

classmethod `private_action_class(action_class)`

Register a private action class.

In some cases action classes can be an implementation detail, for instance in the implementation of a Composite action.

In this case you don’t want the action class to be known but not have a directive.

This function may be used as a decorator like this:

```
@App.private_action_class
class MyActionClass(dectate.Action):
    ...
```

logger_name = 'dectate.directive'
The prefix to use for directive debug logging.

class dectate.**Action**
A configuration action.

Base class of configuration actions.

A configuration action is performed for an object (typically a function or a class object) and affects one or more configuration objects.

Actions can conflict with each other based on their identifier and discriminators. Actions can override each other based on their identifier. Actions can only be in conflict with actions of the same action class or actions with the same `action_group`.

static after(***kw*)
Do setup just after actions in a group are performed.

Can be implemented as a static method by the *Action* subclass.

Parameters ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

static before(***kw*)
Do setup just before actions in a group are performed.

Can be implemented as a static method by the *Action* subclass.

Parameters ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

discriminators(***kw*)
Returns a list of immutables to detect conflicts.

Can be implemented by the *Action* subclass.

Used for additional configuration conflict detection.

Parameters ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

filter_get_value(*name*)
A function to get the filter value.

Takes two arguments, action and name. Should return the value on the filter.

This function is called if the name cannot be determined by looking for the attribute directly using filter_name.

The function should return *NOT_FOUND* if no value with that name can be found.

For example if the filter values are stored on `key_dict`:

```
def filter_get_value(self, name):
    return self.key_dict.get(name, dectate.NOT_FOUND)
```

get_value_for_filter(*name*)
Get value, takes into account filter_name, filter_get_value.

Used by the query system. You can override it if your action has a different way storing values altogether.

identifier(***kw*)

Returns an immutable that uniquely identifies this config.

Needs to be implemented by the `Action` subclass.

Used for overrides and conflict detection.

If two actions in the same group have the same identifier in the same configurable, those two actions are in conflict and a `ConflictError` is raised during `commit()`.

If an action in an extending configurable has the same identifier as the configurable being extended, that action overrides the original one in the extending configurable.

Parameters ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

perform(*obj*, ***kw*)

Do whatever configuration is needed for `obj`.

Needs to be implemented by the `Action` subclass.

Raise a `DirectiveError` to indicate that the action cannot be performed due to incorrect configuration.

Parameters

- **obj** – the object that the action should be performed for. Typically a function or a class object.
- ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

code_info

Info about where in the source code the action was invoked.

Is an instance of `CodeInfo`.

Can be `None` if action does not have an associated directive but was created manually.

config = {}

Describe configuration.

A dict mapping configuration names to factory functions. The resulting configuration objects are passed into `Action.identifier()`, `Action.discriminators()`, `Action.perform()`, and `Action.before()` and `Action.after()`.

After commit completes, the configured objects are found as attributes on `App.config`.

depends = []

List of other action classes to be executed before this one.

The `depends` class attribute contains a list of other action classes that need to be executed before this one is. Actions which depend on another will be executed after those actions are executed.

Omit if you don't care about the order.

filter_compare = {}

Map of names used in query filter to comparison functions.

If for instance you want to be able check whether the value of `model` on the action is a subclass of the value provided in the filter, you can provide it here:

```
filter_compare = {
    'model': issubclass
}
```

The default filter compare is an equality comparison.

filter_convert = {}
Map of names to convert functions.

The query tool that can be generated for a Dectate-based application uses this information to parse filter input into actual objects. If omitted it defaults to passing through the string unchanged.

A conversion function takes a string as input and outputs a Python object. The conversion function may raise `ValueError` if the conversion failed.

A useful conversion function is provided that can be used to refer to an object in a module using a dotted name: `convert_dotted_name()`.

filter_name = {}
Map of names used in query filter to attribute names.

If for instance you want to be able to filter the attribute `_foo` using `foo` in the query, you can map `foo` to `_foo`:

```
filter_name = {
   'foo': '_foo'
}
```

If a filter name is omitted the filter name is assumed to be the same as the attribute name.

group_class = None
Action class to group with.

This class attribute can be supplied with the class of another action that this action should be grouped with. Only actions in the same group can be in conflict. Actions in the same group share the `config` and `before` and `after` of the action class indicated by `group_class`.

By default an action only groups with others of its same class.

class `dectate.`**Composite**
A composite configuration action.

Base class of composite actions.

Composite actions are very simple: implement the `action` method and return a iterable of actions in there.

actions(*obj*)
Specify a iterable of actions to perform for `obj`.

The iteratable should yield `action, obj` tuples, where `action` is an instance of class `Action` or `Composite` and `obj` is the object to perform the action with.

Needs to be implemented by the `Composite` subclass.

code_info
Info about where in the source code the action was invoked.

Is an instance of `CodeInfo`.

Can be `None` if action does not have an associated directive but was created manually.

filter_convert = {}
Map of names to convert functions.

The query tool that can be generated for a Dectate-based application uses this information to parse filter input into actual objects. If omitted it defaults to passing through the string unchanged.

A conversion function takes a string as input and outputs a Python object. The conversion function may raise `ValueError` if the conversion failed.

A useful conversion function is provided that can be used to refer to an object in a module using a dotted name: `convert_dotted_name()`.

query_classes = []

A list of actual action classes that this composite can generate.

This is to allow the querying of composites. If the list if empty (the default) the query system refuses to query the composite. Note that if actions of the same action class can also be generated in another way they are in the same query result.

class dectate.**Query**(**action_classes*)

An object representing a query.

A query can be chained with `Query.filter()`, `Query.attrs()`, `Query.obj()`.

Param `*action_classes`: one or more action classes to query for. Can be instances of `Action` or `Composite`. Can also be strings indicating directive names, in which case they are looked up on the app class before execution.

attrs(**names*)

Extract attributes from resulting actions.

The list of attribute names indicates which keys to include in the dictionary. Obeys `Action.filter_name` and `Action.filter_get_value`.

Param `*names`: list of names to extract.

Returns iterable of dictionaries.

filter(***kw*)

Filter this query by keyword arguments.

The keyword arguments are matched with attributes on the action. `Action.filter_name` is used to map keyword name to attribute name, by default they are the same. `Action.filter_get_value()` can also be implemented for more complicated attribute access as a fallback.

By default the keyword argument values are matched by equality, but you can override this using `Action.filter_compare`.

Can be chained again with a new `filter`.

Parameters **`**kw`** – keyword arguments to match against.

Returns iterable of `(action, obj)`.

obj()

Get objects from results.

Throws away actions in the results and return an iterable of objects.

Returns iterable of decorated objects.

dectate.**query_tool**(*app_classes*)

Command-line query tool for dectate.

Uses command-line arguments to do the query and prints the results.

usage: decq [-h] [–app APP] directive <filter>

Query all directives named `foo` in given app classes:

```
$ decq foo
```

Query directives `foo` with `name` attribute set to `alpha`:

```
$ decq foo name=alpha
```

Query directives `foo` specifically in given app:

```
$ decq --app=myproject.App foo
```

Parameters **app_classes** – a list of *App* subclasses to query by default.

`dectate.`**query_app**(*app_class*, *directive*, ***filters*)

Query a single app with raw filters.

This function is especially useful for writing unit tests that test the conversion behavior.

Parameters

- **app_class** – a *App* subclass to query.
- **directive** – name of directive to query.
- ****filters** – raw (unconverted) filter values.

Returns iterable of `action, obj` tuples.

`dectate.`**convert_dotted_name**(*s*)

Convert input string to an object in a module.

Takes a dotted name: `pkg.module.attr` gets `attr` from module `module` which is in package `pkg`.

To refer to builtin objects, such as `int` or `object`, prefix with `__builtin__.`, so `__builtin__.int` or `__builtin__.object`.

Raises `ValueError` if it cannot be imported.

`dectate.`**convert_bool**(*s*)

Convert input string to boolean.

Input string must either be `True` or `False`.

`dectate.`**NOT_FOUND = <dectate.config.NotFound object>**

Sentinel value returned if filter value cannot be found on action.

class `dectate.`**CodeInfo**(*path*, *lineno*, *sourceline*)

Information about where code was invoked.

The `path` attribute gives the path to the Python module that the code was invoked in.

The `lineno` attribute gives the linenumber in that file.

The `sourceline` attribute contains the actual source line that did the invocation.

exception `dectate.`**ConfigError**

Raised when configuration is bad.

exception `dectate.`**ConflictError**(*actions*)

Bases: `dectate.error.ConfigError`

Raised when there is a conflict in configuration.

Describes where in the code directives are in conflict.

exception `dectate.`**DirectiveError**

Bases: `dectate.error.ConfigError`

Can be raised by user when there directive cannot be performed.

Raise it in *Action.perform()* with a message describing what the problem is:

```
    raise DirectiveError("name should be a string, not None")
```

This is automatically converted by Dectate to a *DirectiveReportError*.

exception dectate.**DirectiveReportError**(*message*, *code_info*)

Bases: dectate.error.ConfigError

Raised when there's a problem with a directive.

Describes where in the code the problem occurred.

CHAPTER 3

Developing Dectate

3.1 Install Dectate for development

First make sure you have virtualenv installed for Python 2.7.

Now create a new virtualenv somewhere for Dectate's development:

```
$ virtualenv /path/to/ve_dectate
```

The goal of this is to isolate you from any globally installed versions of setuptools, which may be incompatible with the requirements of the buildout tool. You should also be able to recycle an existing virtualenv, but this method guarantees a clean one.

Clone Dectate from github (https://github.com/morepath/dectate) and go to the dectate directory:

```
$ git clone git@github.com:morepath/dectate.git
$ cd dectate
```

Now we need to run `bootstrap-buildout.py` to set up buildout, using the Python from the virtualenv we've created before:

```
$ /path/to/ve_dectate/bin/python bootstrap-buildout.py
```

This installs buildout, which can now set up the rest of the development environment:

```
$ bin/buildout
```

This will download and install various dependencies and tools.

3.2 Running the tests

You can run the tests using py.test. Buildout will have installed it for you in the `bin` subdirectory of your project:

```
$ bin/py.test dectate
```

To generate test coverage information as HTML do:

```
$ bin/py.test --cov dectate --cov-report html
```

You can then point your web browser to the `htmlcov/index.html` file in the project directory and click on modules to see detailed coverage information.

3.3 Running the documentation tests

The documentation contains code. To check these code snippets, you can run this code using this command:

```
$ bin/sphinxpython bin/sphinx-build  -b doctest doc out
```

3.4 Building the HTML documentation

To build the HTML documentation (output in `doc/build/html`), run:

```
$ bin/sphinxbuilder
```

3.5 Various checking tools

The buildout will also have installed flake8, which is a tool that can do various checks for common Python mistakes using pyflakes, check for PEP8 style compliance and can do cyclomatic complexity checking. To do pyflakes and pep8 checking do:

```
$ bin/flake8 dectate
```

To also show cyclomatic complexity, use this command:

```
$ bin/flake8 --max-complexity=10 dectate
```

CHAPTER 4

History of Dectate

Dectate was extracted from Morepath and then extensively refactored and cleaned up. It is authored by me, Martijn Faassen.

In the beginning (around 2001) there was zope.configuration, part of the Zope 3 project. It features declarative XML configuration with conflict detection and overrides to assemble pieces of Python code.

In 2006, I helped create the Grok project. This did away with the XML based configuration and instead used Python code. This in turn then drove *zope.configuration*. Grok did not use Python decorators but instead used specially annotated Python classes, which were recursively scanned from modules. Grok's configuration system was spun off as the Martian library.

Chris McDonough was then inspired by Martian to create Venusian, a deferred decorator execution system. It is like Martian in that it imports Python modules recursively in order to find configuration.

I created the Morepath web framework, which uses decorators for configuration throughout and used Venusian. Morepath grew a configuration subsystem where configuration is associated with classes, and uses class inheritance to power configuration reuse and overrides. This configuration subsystem started to get a bit messy as requirements grew.

So in 2016 I extracted the configuration system from Morepath into its own library, Dectate. This allowed me to extensively refactor the code for clarity and features. Dectate does not use Venusian for configuration. Dectate still defers the execution of configuration actions to an explicit commit phase, so that conflict detection and overrides and such can take place.

CHAPTER 5

CHANGES

5.1 0.7 (2016-04-11)

- Fix a few documentation issues.
- Expose `convert_dotted_name` and document it.
- Implement new `convert_bool`.
- Allow use of directive name instead of Action subclass as argument to Query.
- A `query_app` function which is especially helpful when writing tests for the query tool – it takes unconverted filter arguments.
- Use newer version of `with_metaclass` from six.
- Expose `NOT_FOUND` and document it.
- Introduce a new `filter_get_value` method you can implement if the normal attribute getting and `filter_name` are not enough.

5.2 0.6 (2016-04-06)

- Introduce a query system for actions and a command-line tool that lets you query actions.

5.3 0.5 (2016-04-04)

- **Breaking change** The signature of `commit` has changed. Just pass in one or more arguments you want to commit instead of a list. See #8.

5.4 0.4 (2016-04-01)

- Expose `code_info` attribute for action. The `path` in particular can be useful in implementing a directive such as Morepath's `template_directory`. Expose it for composite too.
- Report a few more errors; you cannot use `config`, `before` or `after` after in an action class if `group_class` is set.
- Raise a DirectiveReportError if a DirectiveError is raised in a composite `actions` method.

5.5 0.3 (2016-03-30)

- Document `importscan` package that can be used in combination with this one.
- Introduced `factory_arguments` feature on `config` factories, which can be used to create dependency relationships between configuration.
- Fix a bug where config items were not always properly reused. Now only the first one in the action class dependency order is used, and it is not recreated.

5.6 0.2 (2016-03-29)

- Remove clear_autocommit as it was useless during testing anyway. In tests just use explicit commit.
- Add a `dectate.sphinxext` module that can be plugged into Sphinx so that directives are documented properly.
- Document how Dectate deals with double imports.

5.7 0.1 (2016-03-29)

- Initial public release.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

d