Demystifying Mixins

with Django
I work at Potato
Mixins are a controlled way of adding functionality to classes.
Mixins are **not** special language constructs.
In fact, mixins are ordinary Python classes.

```python
1 class SomeMixin(object):
2     """My smart mixin"""
3
4     def test_method(self):
5         pass
```
Why use mixins?

to improve modularity
When to use mixins?

want to reuse a particular feature in a lot of different classes
Properties

• single responsibility
Properties

• single responsibility
• not meant to be extended
Properties

- single responsibility
- not meant to be extended
- not meant to be instantiated
In Python the concept of mixins is implemented using multiple inheritance.
Order matters
class Foo(BaseFoo, SomeMixIn):
    pass
class Foo(BaseFoo, SomeMixin):
    pass
```python
1 class Foo(BaseFoo, SomeMixin):
2     pass
```
class Foo(BaseFoo, SomeMixin):
    pass
1   class Foo(SomeMixin, BaseFoo):
2       pass
class Foo(SomeMixin, BaseFoo):
    pass
# some_app/views.py

from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
# some_app/views.py

```python
from django.views.generic import TemplateView

class AboutView(SomeMixin, TemplateView):
    template_name = "about.html"
```
# some_app/views.py

from django.views.generic import TemplateView

class AboutView(SomeMixin, TemplateView):
    template_name = "about.html"
My first mixin
class LoginRequiredMixin(object):
class LoginRequiredMixin(object):

def dispatch(self, request, *args, **kwargs):
from django.core.exceptions import PermissionDenied

class LoginRequiredMixin(object):
    
def dispatch(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            raise PermissionDenied
# some_app/views.py

```python
from django.core.exceptions import PermissionDenied

class LoginRequiredMixin(object):
    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_authenticated():
            raise PermissionDenied

        return super(LoginRequiredMixin, self).dispatch(request, *args, **kwargs)
```

# some_app/views.py

```python
from django.views.generic import TemplateView

class AboutView(LoginRequiredMixin, TemplateView):
    template_name = "about.html"
```
LoginRequiredMixin

DetailView

AboutView
dispatch()
dispatch()

check if user is logged in, has permission
dispatch()

get_context_data()

check if user is logged in, has permission
dispatch() check if user is logged in, has permission

get_context_data() add new data to the context
check if user is logged in, has permission

add new data to the context
dispatch()

get_context_data()

get_template_names()

check if user is logged in, has permission

add new data to the context

add more flexibility to the template names
from django.views.generic import TemplateView

class AboutView(TemplateView):
    template_name = "about.html"
dispatch()

get_context_data()

get_template_names()

check if user is logged in, has permission

add new data to the context

add more flexibility to the template names
django-braces
django-braces

Access
Mixins
django-braces

Access Mixins
Form Mixins
Decorators
Decorators

login_required()
Decorators

login_required()
user_passes_test()
Decorators

login_required()
user_passes_test()
permission_required()
Good news everyone!
Django 1.9
Django 1.9

LoginRequiredMixin
Django 1.9

LoginRequiredMixin
UserPassesTestMixin
Django 1.9

LoginRequiredMixin
UserPassesTestMixin
PermissionRequiredMixin
class CuteMixin:

def be_cute(self):
    print "{} ✿♥‿♥✿".format(self.name)
class CuteMixin:
    def be_cute(self):
        print "{} ✿♥‿♥✿".format(self.name)

class Mascot:
    def __init__(self, name):
        self.name = name
```python
class CuteMixin:
    def be_cute(self):
        print "{} ✿♥‿♥✿ " .format(self.name)

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

if __name__ == '__main__':
    domo = Mascot("Domo-kun")
    Mascot.__bases__ += (CuteMixin, )
    domo.be_cute()
```
```python
class CuteMixin:
    def be_cute(self):
        print "{} ✿♥‿♥✿".format(self.name)

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

if __name__ == '__main__':
    domo = Mascot("Domo-kun")
    Mascot.__bases__ += (CuteMixin, )
    domo.be_cute()
```

Domo-kun  🌸❤️😊❤️🌸
class CuteMixin:
    def be_cute(self):
        print "{{} ❥❤️❤️".format(self.name)

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

if __name__ == '__main__':
    domo = Mascot("Domo-kun")
    Mascot.__bases__ += (CuteMixin, )
    domo.be_cute()
```python
class CuteMixin:
    def be_cute(self):
        print(f"{} ♡‿♡".format(self.name))

class Mascot:
    def __init__(self, name):
        self.name = name
```

The code defines two classes, `CuteMixin` and `Mascot`. The `CuteMixin` class has a method `be_cute` which prints a cute message. The `Mascot` class has an `__init__` method which sets the `name` attribute of the instance.
```python
class CuteMixin:
    def be_cute(self):
        print "{{} ✿♥‿♥✿}.format(self.name)

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

if __name__ == '__main__':
    kumamon = Mascot("Kumamon")
    kumamon.__class__ = type('SomeNewType',
        (Mascot, CuteMixin), {})
    kumamon.be_cute()
```

py3
class CuteMixin:
    def be_cute(self):
        print "{} ✿♥‿♥✿".format(self.name)

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

if __name__ == '__main__':
    kumamon = Mascot("Kumamon")
    kumamon.__class__ = type('SomeNewType', (Mascot, CuteMixin), {})
    kumamon.be_cute()
Kumamon ♡ ❤ 😊❤ ♡
With great power comes great responsibility
Recap
Recap

- single responsibility
Recap

• single responsibility
• plug-in functionality
Recap

• single responsibility
• plug-in functionality
• isn’t creating a subtyping relation
Go back to your views and start writing mixins to clean up the code.
Go back to your views and start writing mixins to clean up the code.