Model Inheritance

Model inheritance in Django works almost identically to the way normal class inheritance works in Python. The base class should subclass django.db.models.Model.

- Abstract Base Classes
- Multi-table Inheritance
- Proxy Models

Abstract Base Classes

Abstract base classes are useful when you want to put some common information into a number of other models.

You write your base class and put *abstract=True* in the *Meta* class.

This model will then not be used to create any database table. Instead, when it is used as a base class for other models, its fields will be added to those of the child class.

It does not generate a database table or have a manager, and cannot be instantiated or saved directly.

Fields inherited from abstract base classes can be overridden with another field or value, or be removed with None.

Abstract Base Classes

```
from django.db import models
class CommonInfo(models.Model):
    name = models.CharField(max_length=70)
    age = models.IntegerField()
    class Meta:
        abstract = True


class Student(CommonInfo):
    fees = models.IntegerField()


class Teacher(CommonInfo):
    salary = models.IntegerField()
```

Abstract Base Classes

Meta Inheritance - When an abstract base class is created, Django makes any Meta inner class you declared in the base class available as an attribute.

If a child class does not declare its own Meta class, it will inherit the parent's Meta.

If the child wants to extend the parent's Meta class, it can subclass it.

Django does make one adjustment to the Meta class of an abstract base class: before installing the Meta attribute, it sets abstract=False.

This means that children of abstract base classes don't automatically become abstract classes themselves.

You can make an abstract base class that inherits from another abstract base class. You just need to remember to explicitly set abstract=True each time.

Abstract Base Classes

```
from django.db import models
class CommonInfo(models.Model):
    name = models.CharField(max_length=70)
    age = models.IntegerField()
    class Meta:
        abstract = True


class Student(CommonInfo):
    fees = models.IntegerField()
    class Meta:
        abstract = False
```

Abstract Base Classes

When you are using related_name or related_query_name in an abstract base class (only), part of the value should contain '%(app_label)s' and '%(class)s'.

- '%(class)s' is replaced by the lowercased name of the child class that the field is used in.
- '%(app_label)s' is replaced by the lowercased name of the app the child class is contained within. Each installed application name must be unique and the model class names within each app must also be unique, therefore the resulting name will end up being different.

```
from django.db import models

class Base(models.Model):
    m2m = models.ManyToManyField(
        OtherModel,
        related_name="%(app_label)s_%(class)s_related",
        related_query_name="%(app_label)s_%(class)ss",
    )
    class Meta:
        abstract = True
```

```
class ChildA(Base):
    pass

class ChildB(Base):
    pass
```

Multi-table Inheritance

In this inheritance each model have their own database table, which means base class and child class will have their own table.

The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created OneToOneField).

```
from django.db import models

class ExamCenter(models.Model):
  cname = models.CharField(max_length=70)
  city = models.CharField(max_length=70)


class Student(ExamCenter):
 name = models.CharField(max_length=70)
 roll = models.IntegerField()
```

All of the fields of ExamCenter will also be available in Student, although the data will reside in a different database table.

Proxy Model

Sometimes, however, you only want to change the Python behavior of a model – perhaps to change the default manager, or add a new method.

This is what proxy model inheritance is for: creating a proxy for the original model. You can create, delete and update instances of the proxy model and all the data will be saved as if you were using the original (non-proxied) model. The difference is that you can change things like the default model ordering or the default manager in the proxy, without having to alter the original.

Proxy models are declared like normal models. You tell Django that it’s a proxy model by setting the proxy attribute of the Meta class to True.

Proxy Model

```
from django.db import models
class ExamCenter(models.Model):
  cname = models.CharField(max_length=70)
  city = models.CharField(max_length=70)



class MyExamCenter(ExamCenter):
 class Meta:
   proxy = True
   ordering = ['city']
 def do_something(self):
    pass
```

```
class MySome(ExamCenter):
    objects = NewManager()

    class Meta:
        proxy = True
```

Proxy Model

- A proxy model must inherit from exactly one non-abstract model class.
- You can’t inherit from multiple non-abstract models as the proxy model doesn’t provide any connection between the rows in the different database tables.
- A proxy model can inherit from any number of abstract model classes, providing they do not define any model fields.
- A proxy model may also inherit from any number of proxy models that share a common non-abstract parent class.
- If you don’t specify any model managers on a proxy model, it inherits the managers from its model parents.
- If you define a manager on the proxy model, it will become the default, although any managers defined on the parent classes will still be available.