Django

Documentation

Models

A model is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.

The basics:

- Each model is a Python class that subclasses **`django.db.models.Model`**.
- Each attribute of the model represents a database field.
- With all of this, Django gives you an automatically-generated database-access API; see Making queries.

Quick example

This example model defines a **`Person`**, which has a **`first_name`** and **`last_name`**:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

`first_name` and **`last_name`** are fields of the model. Each field is specified as a class attribute, and each attribute maps to a database column.

The above **`Person`** model would create a database table like this:

```
CREATE TABLE myapp_person (
    "id" serial NOT NULL PRIMARY KEY,
    "first_name" varchar(30) NOT NULL,
    "last_name" varchar(30) NOT NULL
);
```

Some technical notes:

- The name of the table, **`myapp_person`**, is automatically derived from some model metadata but can be overridden. See Table names for more details.
- An **`id`** field is added automatically, but this behavior can be overridden. See Automatic primary key fields.
- The **`CREATE TABLE`** SQL in this example is formatted using PostgreSQL syntax, but it's worth noting Django uses SQL tailored to the database backend specified in your settings file.

Using models

Once you have defined your models, you need to tell Django you're going to *use* those models. Do this by editing your settings file and changing the **`INSTALLED_APPS`** setting to add the name of the module that contains your **`models.py`**.

For example, if the models for your application live in the module **`myapp.models`** (the package structure that is created for an application by the **`manage.py startapp`** script), **`INSTALLED_APPS`** should read, in part:

```
INSTALLED_APPS = [
    #...
    'myapp',
    #...
]
```

When you add new apps to **`INSTALLED_APPS`**, be sure to run **`manage.py migrate`**, optionally making migrations for them first with **`manage.py makemigrations`**.

Fields

The most important part of a model – and the only required part of a model – is the list of database fields it defines. Fields are specified by class attributes. Be careful not to choose field names that conflict with the models API like **`clean`**, **`save`**, or **`delete`**.

Example:

```
from django.db import models

class Musician(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    instrument = models.CharField(max_length=100)

class Album(models.Model):
    artist = models.ForeignKey(Musician, on_delete=models.CASCADE)
    name = models.CharField(max_length=100)
    release_date = models.DateField()
    num_stars = models.IntegerField()
```

Field types

Each field in your model should be an instance of the appropriate **`Field`** class. Django uses the field class types to determine a few things:

- The column type, which tells the database what kind of data to store (e.g. **`INTEGER`**, **`VARCHAR`**, **`TEXT`**).
- The default HTML widget to use when rendering a form field (e.g. **`<input type="text">`**, **`<select>`**).
- The minimal validation requirements, used in Django's admin and in automatically-generated forms.

Django ships with dozens of built-in field types; you can find the complete list in the model field reference. You can easily write your own fields if Django's built-in ones don't do the trick; see How to create custom model fields.

Field options

Each field takes a certain set of field-specific arguments (documented in the model field reference). For example, **`CharField`** (and its subclasses) require a **`max_length`** argument which specifies the size of the **`VARCHAR`** database field used to store the data.

There's also a set of common arguments available to all field types. All are optional. They're fully explained in the reference, but here's a quick summary of the most often-used ones:

`null`
If **`True`**, Django will store empty values as **`NULL`** in the database. Default is **`False`**.

`blank`

If **`True`**, the field is allowed to be blank. Default is **`False`**.

Note that this is different than **`null`**. **`null`** is purely database-related, whereas **`blank`** is validation-related. If a field has **`blank=True`**, form validation will allow entry of an empty value. If a field has **`blank=False`**, the field will be required.

`choices`

A sequence of 2-tuples to use as choices for this field. If this is given, the default form widget will be a select box instead of the standard text field and will limit choices to the choices given.

A choices list looks like this:

```
YEAR_IN_SCHOOL_CHOICES = [
    ('FR', 'Freshman'),
    ('SO', 'Sophomore'),
    ('JR', 'Junior'),
    ('SR', 'Senior'),
    ('GR', 'Graduate'),
]
```

Note

A new migration is created each time the order of **choices** changes.

The first element in each tuple is the value that will be stored in the database. The second element is displayed by the field's form widget.

Given a model instance, the display value for a field with **choices** can be accessed using the **get_FOO_display()** method. For example:

```
from django.db import models

class Person(models.Model):
    SHIRT_SIZES = (
        ('S', 'Small'),
        ('M', 'Medium'),
        ('L', 'Large'),
    )
    name = models.CharField(max_length=60)
    shirt_size = models.CharField(max_length=1, choices=SHIRT_SIZES)
```

```
>>> p = Person(name="Fred Flintstone", shirt_size="L")
>>> p.save()
>>> p.shirt_size
'L'
>>> p.get_shirt_size_display()
'Large'
```

You can also use enumeration classes to define **choices** in a concise way:

```
from django.db import models

class Runner(models.Model):
    MedalType = models.TextChoices('MedalType', 'GOLD SILVER BRONZE')
    name = models.CharField(max_length=60)
    medal = models.CharField(blank=True, choices=MedalType.choices, max_length=10)
```

Further examples are available in the model field reference.

default

The default value for the field. This can be a value or a callable object. If callable it will be called every time a new object is created.

help_text

Extra "help" text to be displayed with the form widget. It's useful for documentation even if your field isn't used on a form.

primary_key

If **True**, this field is the primary key for the model.

If you don't specify **primary_key=True** for any fields in your model, Django will automatically add an **IntegerField** to hold the primary key, so you don't need to set **primary_key=True** on any of your fields unless you want to override the default primary-key behavior. For more, see Automatic primary key fields.

The primary key field is read-only. If you change the value of the primary key on an existing object and then save it, a new object will be created alongside the old one. For example:

```
from django.db import models

class Fruit(models.Model):
    name = models.CharField(max_length=100, primary_key=True)
```

```
>>> fruit = Fruit.objects.create(name='Apple')
>>> fruit.name = 'Pear'
>>> fruit.save()
>>> Fruit.objects.values_list('name', flat=True)
<QuerySet ['Apple', 'Pear']>
```

unique

If **True**, this field must be unique throughout the table.

Again, these are just short descriptions of the most common field options. Full details can be found in the common model field option reference.

Automatic primary key fields

By default, Django gives each model an auto-incrementing primary key with the type specified per app in **AppConfig.default_auto_field** or globally in the **DEFAULT_AUTO_FIELD** setting. For example:

```
id = models.BigAutoField(primary_key=True)
```

If you'd like to specify a custom primary key, specify **primary_key=True** on one of your fields. If Django sees you've explicitly set **Field.primary_key**, it won't add the automatic **id** column.

Each model requires exactly one field to have **primary_key=True** (either explicitly declared or automatically added).

Changed in Django 3.2:

In older versions, auto-created primary key fields were always **AutoField**s.

Verbose field names

Each field type, except for **ForeignKey**, **ManyToManyField** and **OneToOneField**, takes an optional first positional argument – a verbose name. If the verbose name isn't given, Django will automatically create it using the field's attribute name, converting underscores to spaces.

In this example, the verbose name is **"person's first name"**:

```
first_name = models.CharField("person's first name", max_length=30)
```

In this example, the verbose name is **"first name"**:

```
first_name = models.CharField(max_length=30)
```

ForeignKey, **ManyToManyField** and **OneToOneField** require the first argument to be a model class, so use the **verbose_name** keyword argument:

```
poll = models.ForeignKey(
    Poll,
    on_delete=models.CASCADE,
    verbose_name="the related poll",
)
sites = models.ManyToManyField(Site, verbose_name="list of sites")
place = models.OneToOneField(
    Place,
    on_delete=models.CASCADE,
    verbose_name="related place",
)
```

The convention is not to capitalize the first letter of the **verbose_name**. Django will automatically capitalize the first letter where it needs to.

Relationships

Clearly, the power of relational databases lies in relating tables to each other. Django offers ways to define the three most common types of database relationships: many-to-one, many-to-many and one-to-one.

Many-to-one relationships

To define a many-to-one relationship, use **`django.db.models.ForeignKey`**. You use it just like any other **`Field`** type: by including it as a class attribute of your model.

`ForeignKey` requires a positional argument: the class to which the model is related.

For example, if a **`Car`** model has a **`Manufacturer`** – that is, a **`Manufacturer`** makes multiple cars but each **`Car`** only has one **`Manufacturer`** – use the following definitions:

```
from django.db import models

class Manufacturer(models.Model):
    # ...
    pass

class Car(models.Model):
    manufacturer = models.ForeignKey(Manufacturer, on_delete=models.CASCADE)
    # ...
```

You can also create recursive relationships (an object with a many-to-one relationship to itself) and relationships to models not yet defined; see the model field reference for details.

It's suggested, but not required, that the name of a **`ForeignKey`** field (**`manufacturer`** in the example above) be the name of the model, lowercase. You can call the field whatever you want. For example:

```
class Car(models.Model):
    company_that_makes_it = models.ForeignKey(
        Manufacturer,
        on_delete=models.CASCADE,
    )
    # ...
```

See also

`ForeignKey` fields accept a number of extra arguments which are explained in the model field reference. These options help define how the relationship should work; all are optional.

For details on accessing backwards-related objects, see the Following relationships backward example.

For sample code, see the Many-to-one relationship model example.

Many-to-many relationships

To define a many-to-many relationship, use **`ManyToManyField`**. You use it just like any other **`Field`** type: by including it as a class attribute of your model.

`ManyToManyField` requires a positional argument: the class to which the model is related.

For example, if a **`Pizza`** has multiple **`Topping`** objects – that is, a **`Topping`** can be on multiple pizzas and each **`Pizza`** has multiple toppings – here's how you'd represent that:

```
from django.db import models

class Topping(models.Model):
    # ...
    pass

class Pizza(models.Model):
    # ...
    toppings = models.ManyToManyField(Topping)
```

As with **`ForeignKey`**, you can also create recursive relationships (an object with a many-to-many relationship to itself) and relationships to models not yet defined.

It's suggested, but not required, that the name of a **`ManyToManyField`** (**`toppings`** in the example above) be a plural describing the set of related model objects.

It doesn't matter which model has the **ManyToManyField**, but you should only put it in one of the models – not both.

Generally, **ManyToManyField** instances should go in the object that's going to be edited on a form. In the above example, **toppings** is in **Pizza** (rather than **Topping** having a **pizzas** **ManyToManyField**) because it's more natural to think about a pizza having toppings than a topping being on multiple pizzas. The way it's set up above, the **Pizza** form would let users select the toppings.

See also

See the Many-to-many relationship model example for a full example.

ManyToManyField fields also accept a number of extra arguments which are explained in the model field reference. These options help define how the relationship should work; all are optional.

Extra fields on many-to-many relationships

When you're only dealing with many-to-many relationships such as mixing and matching pizzas and toppings, a standard **ManyToManyField** is all you need. However, sometimes you may need to associate data with the relationship between two models.

For example, consider the case of an application tracking the musical groups which musicians belong to. There is a many-to-many relationship between a person and the groups of which they are a member, so you could use a **ManyToManyField** to represent this relationship. However, there is a lot of detail about the membership that you might want to collect, such as the date at which the person joined the group.

For these situations, Django allows you to specify the model that will be used to govern the many-to-many relationship. You can then put extra fields on the intermediate model. The intermediate model is associated with the **ManyToManyField** using the **through** argument to point to the model that will act as an intermediary. For our musician example, the code would look something like this:

```
from django.db import models

class Person(models.Model):
    name = models.CharField(max_length=128)

    def __str__(self):
        return self.name

class Group(models.Model):
    name = models.CharField(max_length=128)
    members = models.ManyToManyField(Person, through='Membership')

    def __str__(self):
        return self.name

class Membership(models.Model):
    person = models.ForeignKey(Person, on_delete=models.CASCADE)
    group = models.ForeignKey(Group, on_delete=models.CASCADE)
    date_joined = models.DateField()
    invite_reason = models.CharField(max_length=64)
```

When you set up the intermediary model, you explicitly specify foreign keys to the models that are involved in the many-to-many relationship. This explicit declaration defines how the two models are related.

There are a few restrictions on the intermediate model:

- Your intermediate model must contain one - and *only* one - foreign key to the source model (this would be **Group** in our example), or you must explicitly specify the foreign keys Django should use for the relationship using **ManyToManyField.through_fields**. If you have more than one foreign key and **through_fields** is not specified, a validation error will be raised. A similar restriction applies to the foreign key to the target model (this would be **Person** in our example).
- For a model which has a many-to-many relationship to itself through an intermediary model, two foreign keys to the same model are permitted, but they will be treated as the two (different) sides of the many-to-many relationship. If there are *more* than two foreign keys though, you must also specify **through_fields** as above, or a validation error will be raised.

Now that you have set up your **ManyToManyField** to use your intermediary model (**Membership**, in this case), you're ready to start creating some many-to-many relationships. You do this by creating instances of the intermediate model:

```
>>> ringo = Person.objects.create(name="Ringo Starr")
>>> paul = Person.objects.create(name="Paul McCartney")
>>> beatles = Group.objects.create(name="The Beatles")
>>> m1 = Membership(person=ringo, group=beatles,
...     date_joined=date(1962, 8, 16),
...     invite_reason="Needed a new drummer.")
>>> m1.save()
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>]>
>>> ringo.group_set.all()
<QuerySet [<Group: The Beatles>]>
>>> m2 = Membership.objects.create(person=paul, group=beatles,
...     date_joined=date(1960, 8, 1),
...     invite_reason="Wanted to form a band.")
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>, <Person: Paul McCartney>]>
```

You can also use **`add()`**, **`create()`**, or **`set()`** to create relationships, as long as you specify **`through_defaults`** for any required fields:

```
>>> beatles.members.add(john, through_defaults={'date_joined': date(1960, 8, 1)})
>>> beatles.members.create(name="George Harrison", through_defaults={'date_joined': date(1960, 8, 1)})
>>> beatles.members.set([john, paul, ringo, george], through_defaults={'date_joined': date(1960, 8, 1)})
```

You may prefer to create instances of the intermediate model directly.

If the custom through table defined by the intermediate model does not enforce uniqueness on the **`(model1, model2)`** pair, allowing multiple values, the **`remove()`** call will remove all intermediate model instances:

```
>>> Membership.objects.create(person=ringo, group=beatles,
...     date_joined=date(1968, 9, 4),
...     invite_reason="You've been gone for a month and we miss you.")
>>> beatles.members.all()
<QuerySet [<Person: Ringo Starr>, <Person: Paul McCartney>, <Person: Ringo Starr>]>
>>> # This deletes both of the intermediate model instances for Ringo Starr
>>> beatles.members.remove(ringo)
>>> beatles.members.all()
<QuerySet [<Person: Paul McCartney>]>
```

The **`clear()`** method can be used to remove all many-to-many relationships for an instance:

```
>>> # Beatles have broken up
>>> beatles.members.clear()
>>> # Note that this deletes the intermediate model instances
>>> Membership.objects.all()
<QuerySet []>
```

Once you have established the many-to-many relationships, you can issue queries. Just as with normal many-to-many relationships, you can query using the attributes of the many-to-many-related model:

```
# Find all the groups with a member whose name starts with 'Paul'
>>> Group.objects.filter(members__name__startswith='Paul')
<QuerySet [<Group: The Beatles>]>
```

As you are using an intermediate model, you can also query on its attributes:

```
# Find all the members of the Beatles that joined after 1 Jan 1961
>>> Person.objects.filter(
...     group__name='The Beatles',
...     membership__date_joined__gt=date(1961,1,1))
<QuerySet [<Person: Ringo Starr]>
```

If you need to access a membership's information you may do so by directly querying the **`Membership`** model:

```
>>> ringos_membership = Membership.objects.get(group=beatles, person=ringo)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

Another way to access the same information is by querying the many-to-many reverse relationship from a **`Person`** object:

```
>>> ringos_membership = ringo.membership_set.get(group=beatles)
>>> ringos_membership.date_joined
datetime.date(1962, 8, 16)
>>> ringos_membership.invite_reason
'Needed a new drummer.'
```

One-to-one relationships

To define a one-to-one relationship, use **`OneToOneField`**. You use it just like any other **`Field`** type: by including it as a class attribute of your model.

This is most useful on the primary key of an object when that object "extends" another object in some way.

`OneToOneField` requires a positional argument: the class to which the model is related.

For example, if you were building a database of "places", you would build pretty standard stuff such as address, phone number, etc. in the database. Then, if you wanted to build a database of restaurants on top of the places, instead of repeating yourself and replicating those fields in the **`Restaurant`** model, you could make **`Restaurant`** have a **`OneToOneField`** to **`Place`** (because a restaurant "is a" place; in fact, to handle this you'd typically use inheritance, which involves an implicit one-to-one relation).

As with **`ForeignKey`**, a recursive relationship can be defined and references to as-yet undefined models can be made.

See also

See the One-to-one relationship model example for a full example.

`OneToOneField` fields also accept an optional **`parent_link`** argument.

`OneToOneField` classes used to automatically become the primary key on a model. This is no longer true (although you can manually pass in the **`primary_key`** argument if you like). Thus, it's now possible to have multiple fields of type **`OneToOneField`** on a single model.

Models across files

It's perfectly OK to relate a model to one from another app. To do this, import the related model at the top of the file where your model is defined. Then, refer to the other model class wherever needed. For example:

```
from django.db import models
from geography.models import ZipCode

class Restaurant(models.Model):
    # ...
    zip_code = models.ForeignKey(
        ZipCode,
        on_delete=models.SET_NULL,
        blank=True,
        null=True,
    )
```

Field name restrictions

Django places some restrictions on model field names:

1. A field name cannot be a Python reserved word, because that would result in a Python syntax error. For example:

```
class Example(models.Model):
    pass = models.IntegerField() # 'pass' is a reserved word!
```

2. A field name cannot contain more than one underscore in a row, due to the way Django's query lookup syntax works. For example:

```
class Example(models.Model):
    foo__bar = models.IntegerField() # 'foo__bar' has two underscores!
```

3. A field name cannot end with an underscore, for similar reasons.

These limitations can be worked around, though, because your field name doesn't necessarily have to match your database column name. See the **db_column** option.

SQL reserved words, such as **join**, **where** or **select**, *are* allowed as model field names, because Django escapes all database table names and column names in every underlying SQL query. It uses the quoting syntax of your particular database engine.

Custom field types

If one of the existing model fields cannot be used to fit your purposes, or if you wish to take advantage of some less common database column types, you can create your own field class. Full coverage of creating your own fields is provided in How to create custom model fields.

Meta options

Give your model metadata by using an inner **class Meta**, like so:

```
from django.db import models

class Ox(models.Model):
    horn_length = models.IntegerField()

    class Meta:
        ordering = ["horn_length"]
        verbose_name_plural = "oxen"
```

Model metadata is "anything that's not a field", such as ordering options (**ordering**), database table name (**db_table**), or human-readable singular and plural names (**verbose_name** and **verbose_name_plural**). None are required, and adding **class Meta** to a model is completely optional.

A complete list of all possible **Meta** options can be found in the model option reference.

Model attributes

`objects`

The most important attribute of a model is the **`Manager`**. It's the interface through which database query operations are provided to Django models and is used to retrieve the instances from the database. If no custom **`Manager`** is defined, the default name is **`objects`**. Managers are only accessible via model classes, not the model instances.

Model methods

Define custom methods on a model to add custom "row-level" functionality to your objects. Whereas **`Manager`** methods are intended to do "table-wide" things, model methods should act on a particular model instance.

This is a valuable technique for keeping business logic in one place – the model.

For example, this model has a few custom methods:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=50)
    last_name = models.CharField(max_length=50)
    birth_date = models.DateField()

    def baby_boomer_status(self):
        "Returns the person's baby-boomer status."
        import datetime
        if self.birth_date < datetime.date(1945, 8, 1):
            return "Pre-boomer"
        elif self.birth_date < datetime.date(1965, 1, 1):
            return "Baby boomer"
        else:
            return "Post-boomer"

    @property
    def full_name(self):
        "Returns the person's full name."
        return '%s %s' % (self.first_name, self.last_name)
```

The last method in this example is a property.

The model instance reference has a complete list of methods automatically given to each model. You can override most of these – see overriding predefined model methods, below – but there are a couple that you'll almost always want to define:

`__str__()`

A Python "magic method" that returns a string representation of any object. This is what Python and Django will use whenever a model instance needs to be coerced and displayed as a plain string. Most notably, this happens when you display an object in an interactive console or in the admin.

You'll always want to define this method; the default isn't very helpful at all.

`get_absolute_url()`

This tells Django how to calculate the URL for an object. Django uses this in its admin interface, and any time it needs to figure out a URL for an object.

Any object that has a URL that uniquely identifies it should define this method.

Overriding predefined model methods

There's another set of model methods that encapsulate a bunch of database behavior that you'll want to customize. In particular you'll often want to change the way **`save()`** and **`delete()`** work.

You're free to override these methods (and any other model method) to alter behavior.

A classic use-case for overriding the built-in methods is if you want something to happen whenever you save an object. For example (see **`save()`** for documentation of the parameters it accepts):

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        do_something()
        super().save(*args, **kwargs)  # Call the "real" save() method.
        do_something_else()
```

You can also prevent saving:

```
from django.db import models

class Blog(models.Model):
    name = models.CharField(max_length=100)
    tagline = models.TextField()

    def save(self, *args, **kwargs):
        if self.name == "Yoko Ono's blog":
            return # Yoko shall never have her own blog!
        else:
            super().save(*args, **kwargs)  # Call the "real" save() method.
```

It's important to remember to call the superclass method – that's that **`super().save(*args, **kwargs)`** business – to ensure that the object still gets saved into the database. If you forget to call the superclass method, the default behavior won't happen and the database won't get touched.

It's also important that you pass through the arguments that can be passed to the model method – that's what the **`*args, **kwargs`** bit does. Django will, from time to time, extend the capabilities of built-in model methods, adding new arguments. If you use **`*args, **kwargs`** in your method definitions, you are guaranteed that your code will automatically support those arguments when they are added.

Overridden model methods are not called on bulk operations

Note that the **`delete()`** method for an object is not necessarily called when deleting objects in bulk using a QuerySet or as a result of a **`cascading delete`**. To ensure customized delete logic gets executed, you can use **`pre_delete`** and/or **`post_delete`** signals.

Unfortunately, there isn't a workaround when **`creating`** or **`updating`** objects in bulk, since none of **`save()`**, **`pre_save`**, and **`post_save`** are called.

Executing custom SQL

Another common pattern is writing custom SQL statements in model methods and module-level methods. For more details on using raw SQL, see the documentation on using raw SQL.

Model inheritance

Model inheritance in Django works almost identically to the way normal class inheritance works in Python, but the basics at the beginning of the page should still be followed. That means the base class should subclass **`django.db.models.Model`**.

The only decision you have to make is whether you want the parent models to be models in their own right (with their own database tables), or if the parents are just holders of common information that will only be visible through the child models.

There are three styles of inheritance that are possible in Django.

1. Often, you will just want to use the parent class to hold information that you don't want to have to type out for each child model. This class isn't going to ever be used in isolation, so Abstract base classes are what you're after.
2. If you're subclassing an existing model (perhaps something from another application entirely) and want each model to have its own database table, Multi-table inheritance is the way to go.
3. Finally, if you only want to modify the Python-level behavior of a model, without changing the models fields in any way, you can use 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.

An example:

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True

class Student(CommonInfo):
    home_group = models.CharField(max_length=5)
```

The **`Student`** model will have three fields: **`name`**, **`age`** and **`home_group`**. The **`CommonInfo`** model cannot be used as a normal Django model, since it is an abstract base 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`**.

For many uses, this type of model inheritance will be exactly what you want. It provides a way to factor out common information at the Python level, while still only creating one database table per child model at the database level.

`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. For example:

```
from django.db import models

class CommonInfo(models.Model):
    # ...
    class Meta:
        abstract = True
        ordering = ['name']

class Student(CommonInfo):
    # ...
    class Meta(CommonInfo.Meta):
        db_table = 'student_info'
```

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. To make an abstract base class that inherits from another abstract base class, you need to explicitly set **`abstract=True`** on the child.

Some attributes won't make sense to include in the Meta class of an abstract base class. For example, including **`db_table`** would mean that all the child classes (the ones that don't specify their own Meta) would use the same database table, which is almost certainly not what you want.

Due to the way Python inheritance works, if a child class inherits from multiple abstract base classes, only the Meta options from the first listed class will be inherited by default. To inherit Meta options from multiple abstract base classes, you must explicitly declare the Meta inheritance. For example:

```
from django.db import models

class CommonInfo(models.Model):
    name = models.CharField(max_length=100)
    age = models.PositiveIntegerField()

    class Meta:
        abstract = True
        ordering = ['name']

class Unmanaged(models.Model):
    class Meta:
        abstract = True
        managed = False

class Student(CommonInfo, Unmanaged):
    home_group = models.CharField(max_length=5)

    class Meta(CommonInfo.Meta, Unmanaged.Meta):
        pass
```

Be careful with `related_name` and `related_query_name`

If you are using **`related_name`** or **`related_query_name`** on a **`ForeignKey`** or **`ManyToManyField`**, you must always specify a *unique* reverse name and query name for the field. This would normally cause a problem in abstract base classes, since the fields on this class are included into each of the child classes, with exactly the same values for the attributes (including **`related_name`** and **`related_query_name`**) each time.

To work around this problem, 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.

For example, given an app **`common/models.py`**:

```
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
```

Along with another app **`rare/models.py`**:

```
from common.models import Base

class ChildB(Base):
    pass
```

The reverse name of the **`common.ChildA.m2m`** field will be **`common_childa_related`** and the reverse query name will be **`common_childas`**. The reverse name of the **`common.ChildB.m2m`** field will be **`common_childb_related`** and the reverse query name will be **`common_childbs`**. Finally, the reverse name of the **`rare.ChildB.m2m`** field will be **`rare_childb_related`** and the reverse query name will be **`rare_childbs`**. It's up to you how you use the **`'%(class)s'`** and **`'%(app_label)s'`** portion to construct your related name or related query name but if you forget to use it, Django will raise errors when you perform system checks (or run **`migrate`**).

If you don't specify a **related_name** attribute for a field in an abstract base class, the default reverse name will be the name of the child class followed by **'_set'**, just as it normally would be if you'd declared the field directly on the child class. For example, in the above code, if the **related_name** attribute was omitted, the reverse name for the **m2m** field would be **childa_set** in the **ChildA** case and **childb_set** for the **ChildB** field.

Multi-table inheritance

The second type of model inheritance supported by Django is when each model in the hierarchy is a model all by itself. Each model corresponds to its own database table and can be queried and created individually. The inheritance relationship introduces links between the child model and each of its parents (via an automatically-created **OneToOneField**). For example:

```
from django.db import models

class Place(models.Model):
    name = models.CharField(max_length=50)
    address = models.CharField(max_length=80)

class Restaurant(Place):
    serves_hot_dogs = models.BooleanField(default=False)
    serves_pizza = models.BooleanField(default=False)
```

All of the fields of **Place** will also be available in **Restaurant**, although the data will reside in a different database table. So these are both possible:

```
>>> Place.objects.filter(name="Bob's Cafe")
>>> Restaurant.objects.filter(name="Bob's Cafe")
```

If you have a **Place** that is also a **Restaurant**, you can get from the **Place** object to the **Restaurant** object by using the lowercase version of the model name:

```
>>> p = Place.objects.get(id=12)
# If p is a Restaurant object, this will give the child class:
>>> p.restaurant
<Restaurant: ...>
```

However, if **p** in the above example was *not* a **Restaurant** (it had been created directly as a **Place** object or was the parent of some other class), referring to **p.restaurant** would raise a **Restaurant.DoesNotExist** exception.

The automatically-created **OneToOneField** on **Restaurant** that links it to **Place** looks like this:

```
place_ptr = models.OneToOneField(
    Place, on_delete=models.CASCADE,
    parent_link=True,
    primary_key=True,
)
```

You can override that field by declaring your own **OneToOneField** with **parent_link=True** on **Restaurant**.

Meta and multi-table inheritance

In the multi-table inheritance situation, it doesn't make sense for a child class to inherit from its parent's Meta class. All the Meta options have already been applied to the parent class and applying them again would normally only lead to contradictory behavior (this is in contrast with the abstract base class case, where the base class doesn't exist in its own right).

So a child model does not have access to its parent's Meta class. However, there are a few limited cases where the child inherits behavior from the parent: if the child does not specify an **ordering** attribute or a **get_latest_by** attribute, it will inherit these from its parent.

If the parent has an ordering and you don't want the child to have any natural ordering, you can explicitly disable it:

```
class ChildModel(ParentModel):
    # ...
    class Meta:
        # Remove parent's ordering effect
        ordering = []
```

Inheritance and reverse relations

Because multi-table inheritance uses an implicit **OneToOneField** to link the child and the parent, it's possible to move from the parent down to the child, as in the above example. However, this uses up the name that is the default **related_name** value for **ForeignKey** and **ManyToManyField** relations. If you are putting those types of relations on a subclass of the parent model, you **must** specify the **related_name** attribute on each such field. If you forget, Django will raise a validation error.

For example, using the above **Place** class again, let's create another subclass with a **ManyToManyField**:

```
class Supplier(Place):
    customers = models.ManyToManyField(Place)
```

This results in the error:

```
Reverse query name for 'Supplier.customers' clashes with reverse query
name for 'Supplier.place_ptr'.

HINT: Add or change a related_name argument to the definition for
'Supplier.customers' or 'Supplier.place_ptr'.
```

Adding **related_name** to the **customers** field as follows would resolve the error: **models.ManyToManyField(Place, related_name='provider')**.

Specifying the parent link field

As mentioned, Django will automatically create a **OneToOneField** linking your child class back to any non-abstract parent models. If you want to control the name of the attribute linking back to the parent, you can create your own **OneToOneField** and set **parent_link=True** to indicate that your field is the link back to the parent class.

Proxy models

When using multi-table inheritance, a new database table is created for each subclass of a model. This is usually the desired behavior, since the subclass needs a place to store any additional data fields that are not present on the base class. 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**.

For example, suppose you want to add a method to the **Person** model. You can do it like this:

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)

class MyPerson(Person):
    class Meta:
        proxy = True

    def do_something(self):
        # ...
        pass
```

The **MyPerson** class operates on the same database table as its parent **Person** class. In particular, any new instances of **Person** will also be accessible through **MyPerson**, and vice-versa:

```
>>> p = Person.objects.create(first_name="foobar")
>>> MyPerson.objects.get(first_name="foobar")
<MyPerson: foobar>
```

You could also use a proxy model to define a different default ordering on a model. You might not always want to order the **Person** model, but regularly order by the **last_name** attribute when you use the proxy:

```
class OrderedPerson(Person):
    class Meta:
        ordering = ["last_name"]
        proxy = True
```

Now normal **Person** queries will be unordered and **OrderedPerson** queries will be ordered by **last_name**.

Proxy models inherit **Meta** attributes in the same way as regular models.

QuerySets still return the model that was requested

There is no way to have Django return, say, a **MyPerson** object whenever you query for **Person** objects. A queryset for **Person** objects will return those types of objects. The whole point of proxy objects is that code relying on the original **Person** will use those and your own code can use the extensions you included (that no other code is relying on anyway). It is not a way to replace the **Person** (or any other) model everywhere with something of your own creation.

Base class restrictions

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.

Proxy model managers

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.

Continuing our example from above, you could change the default manager used when you query the **Person** model like this:

```
from django.db import models

class NewManager(models.Manager):
    # ...
    pass

class MyPerson(Person):
    objects = NewManager()

    class Meta:
        proxy = True
```

If you wanted to add a new manager to the Proxy, without replacing the existing default, you can use the techniques described in the custom manager documentation: create a base class containing the new managers and inherit that after the primary base class:

```
# Create an abstract class for the new manager.
class ExtraManagers(models.Model):
    secondary = NewManager()

    class Meta:
        abstract = True

class MyPerson(Person, ExtraManagers):
    class Meta:
        proxy = True
```

You probably won't need to do this very often, but, when you do, it's possible.

Differences between proxy inheritance and unmanaged models

Proxy model inheritance might look fairly similar to creating an unmanaged model, using the **managed** attribute on a model's **Meta** class.

With careful setting of **Meta.db_table** you could create an unmanaged model that shadows an existing model and adds Python methods to it. However, that would be very repetitive and fragile as you need to keep both copies synchronized if you make any changes.

On the other hand, proxy models are intended to behave exactly like the model they are proxying for. They are always in sync with the parent model since they directly inherit its fields and managers.

The general rules are:

1. If you are mirroring an existing model or database table and don't want all the original database table columns, use **Meta.managed=False**. That option is normally useful for modeling database views and tables not under the control of Django.
2. If you are wanting to change the Python-only behavior of a model, but keep all the same fields as in the original, use **Meta.proxy=True**. This sets things up so that the proxy model is an exact copy of the storage structure of the original model when data is saved.

Multiple inheritance

Just as with Python's subclassing, it's possible for a Django model to inherit from multiple parent models. Keep in mind that normal Python name resolution rules apply. The first base class that a particular name (e.g. Meta) appears in will be the one that is used; for example, this means that if multiple parents contain a Meta class, only the first one is going to be used, and all others will be ignored.

Generally, you won't need to inherit from multiple parents. The main use-case where this is useful is for "mix-in" classes: adding a particular extra field or method to every class that inherits the mix-in. Try to keep your inheritance hierarchies as simple and straightforward as possible so that you won't have to struggle to work out where a particular piece of information is coming from.

Note that inheriting from multiple models that have a common **id** primary key field will raise an error. To properly use multiple inheritance, you can use an explicit **AutoField** in the base models:

```
class Article(models.Model):
    article_id = models.AutoField(primary_key=True)
    ...

class Book(models.Model):
    book_id = models.AutoField(primary_key=True)
    ...

class BookReview(Book, Article):
    pass
```

Or use a common ancestor to hold the **AutoField**. This requires using an explicit **OneToOneField** from each parent model to the common ancestor to avoid a clash between the fields that are automatically generated and inherited by the child:

```
class Piece(models.Model):
    pass

class Article(Piece):
    article_piece = models.OneToOneField(Piece, on_delete=models.CASCADE, parent_link=True)
    ...

class Book(Piece):
    book_piece = models.OneToOneField(Piece, on_delete=models.CASCADE, parent_link=True)
    ...

class BookReview(Book, Article):
    pass
```

Field name “hiding” is not permitted

In normal Python class inheritance, it is permissible for a child class to override any attribute from the parent class. In Django, this isn’t usually permitted for model fields. If a non-abstract model base class has a field called **`author`**, you can’t create another model field or define an attribute called **`author`** in any class that inherits from that base class.

This restriction doesn’t apply to model fields inherited from an abstract model. Such fields may be overridden with another field or value, or be removed by setting **`field_name = None`**.

Warning

Model managers are inherited from abstract base classes. Overriding an inherited field which is referenced by an inherited **`Manager`** may cause subtle bugs. See custom managers and model inheritance.

Note

Some fields define extra attributes on the model, e.g. a **`ForeignKey`** defines an extra attribute with **`_id`** appended to the field name, as well as **`related_name`** and **`related_query_name`** on the foreign model.

These extra attributes cannot be overridden unless the field that defines it is changed or removed so that it no longer defines the extra attribute.

Overriding fields in a parent model leads to difficulties in areas such as initializing new instances (specifying which field is being initialized in **`Model.__init__`**) and serialization. These are features which normal Python class inheritance doesn’t have to deal with in quite the same way, so the difference between Django model inheritance and Python class inheritance isn’t arbitrary.

This restriction only applies to attributes which are **`Field`** instances. Normal Python attributes can be overridden if you wish. It also only applies to the name of the attribute as Python sees it: if you are manually specifying the database column name, you can have the same column name appearing in both a child and an ancestor model for multi-table inheritance (they are columns in two different database tables).

Django will raise a **`FieldError`** if you override any model field in any ancestor model.

Note that because of the way fields are resolved during class definition, model fields inherited from multiple abstract parent models are resolved in a strict depth-first order. This contrasts with standard Python MRO, which is resolved breadth-first in cases of diamond shaped inheritance. This difference only affects complex model hierarchies, which (as per the advice above) you should try to avoid.

Organizing models in a package

The **`manage.py startapp`** command creates an application structure that includes a **`models.py`** file. If you have many models, organizing them in separate files may be useful.

To do so, create a **`models`** package. Remove **`models.py`** and create a **`myapp/models/`** directory with an **`__init__.py`** file and the files to store your models. You must import the models in the **`__init__.py`** file.

For example, if you had **`organic.py`** and **`synthetic.py`** in the **`models`** directory:

`myapp/models/__init__.py`

```
from .organic import Person
from .synthetic import Robot
```

Explicitly importing each model rather than using **`from .models import *`** has the advantages of not cluttering the namespace, making code more readable, and keeping code analysis tools useful.

See also

The Models Reference
Covers all the model related APIs including model fields, related objects, and **`QuerySet`**.

Models and databases

Making queries

Sponsor Django

Official merchandise store

Amazon Smile

Benevity Workplace Giving Program