## **Django Models Best Practices**

Follow Django’s [defined conventions](https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/#model-style) for model code.

A common pattern in MVC-style programming is to build thick/fat models and thin controllers. For Django this translates to building models with lots of small methods attached to them and views which use those methods to keep their logic as minimal as possible. There are lots of benefits to this approach.

1. **DRY**: Rather than repeating the same logic in multiple views, it is defined once on the model.
2. **Testable**: Breaking up logic into small methods on the model makes your code easier to unit test.
3. **Readable**: By giving your methods friendly names, you can abstract ugly logic into something that is easily readable and understandable.

For a good example of a fat model in Django, look at [the definition of django.contrib.auth.models.User](https://github.com/django/django/blob/ff6ee5f06c2850f098863d4a747069e10727293e/django/contrib/auth/models.py#L225-404).

Following are a few suggested best practises:

## **1. Correct Model Naming**

It is generally recommended to use singular nouns for model naming, for example: User, Post, Article. That is, the last component of the name should be a noun, e.g.: Some New Shiny Item. It is correct to use singular numbers when one unit of a model does not contain information about several objects.

## **2. Relationship Field Naming**

For relationships such as ForeignKey, OneToOneKey, ManyToMany it is sometimes better to specify a name. Imagine there is a model called Article, - in which one of the relationships is ForeignKey for model User. If this field contains information about the author of the article, then author will be a more appropriate name than user.

## **3. Correct Related-Name**

It is reasonable to indicate a related-name in plural as related-name addressing returns queryset. Please, do set adequate related-names. In the majority of cases, the name of the model in plural will be just right. For example:

class Owner(models.Model):  
 pass  
class Item(models.Model):  
 owner = models.ForeignKey(Owner, related\_name='items')

## **4. Do not use ForeignKey with unique=True**

There is no point in using ForeignKey with unique=True as there exists OneToOneField for such cases.

## **5. Adding a Model via Migration**

If you need to add a model, then, having created a class of a model, execute serially manage.py commands makemigrations and migrate.

## **6. Denormalisations**

You should not allow thoughtless use of denormalization in relational databases. Always try to avoid it, except for the cases when you denormalise data consciously for whatever the reason may be (e.g. productivity). If at the stage of database designing you understand that you need to denormalise much of the data, a good option could be the use of NoSQL. However, if most of data does not require denormalisation, which cannot be avoided, think about a relational base with JsonField to store some data.

## **7. BooleanField**

Do not use null=True or blank=True for BooleanField. It should also be pointed out that it is better to specify default values for such fields. If you realise that the field can remain empty, you need ***NullBooleanField.***

## **8. Business Logic in Models**

The best place to allocate business logic for your project is in models, namely method models and model manager. It is possible that method models can only provoke some methods/functions. If it is inconvenient or impossible to allocate logic in models, you need to replace its forms or serializers in tasks.

## **9. Field Duplication in ModelForm**

Do not duplicate model fields in ModelForm or ModelSerializer without need. If you want to specify that the form uses all model fields, use MetaFields. If you need to redefine a widget for a field with nothing else to be changed in this field, make use of Meta widgets to indicate widgets.

## **10. Do not use ObjectDoesNotExist**

Using ModelName.DoesNotExist instead of ObjectDoesNotExist makes your exception intercepting more specialised, which is a positive practice.

## **11. Many flags in a model?**

If it is justified, replace several BooleanFields with one field, status-like. e.g.

class Article(models.Model):  
 is\_published = models.BooleanField(default=False)  
 is\_verified = models.BooleanField(default=False)  
 …

Assume the logic of our application presupposes that the article is not published and checked initially, then it is checked and marked is\_verified in True and then it is published. You can notice that article cannot be published without being checked. So there are 3 conditions in total, but with 2 boolean fields we do not have 4 possible variants, and you should make sure there are no articles with wrong boolean fields conditions combinations. That is why using one status field instead of two boolean fields is a better option:

class Article(models.Model):  
 STATUSES = Choices('new', 'verified', 'published')  
  
 status = models.IntegerField(choices=STATUSES, default=STATUSES.draft)  
 …

This example may not be very illustrative, but imagine that you have 3 or more such boolean fields in your model, and validation control for these field value combinations can be really tiresome.

## **12. Redundant model name in a field name**

Do not add model names to fields if there is no need to do so, e.g. if table User has a field user\_status - you should rename the field into status, as long as there are no other statuses in this model.

## **13. Dirty data should not be found in a base**

Always use PositiveIntegerField instead of IntegerField if it is not senseless, because “bad” data must not go to the base. For the same reason you should always use unique,unique\_together for logically unique data and never use required=False in every field.

## 

## **14. Getting the earliest/latest object**

You can use ModelName.objects.earliest('created'/'earliest') instead of order\_by('created')[0] and you can also put get\_latest\_by in Meta model. You should keep in mind that latest/earliest as well as get can cause an exception DoesNotExist. Therefore, order\_by('created').first() is the most useful variant.

## **15. Never make len(queryset)**

Do not use len to get queryset’s objects amount. The count method can be used for this purpose. Like this: len(ModelName.objects.all()), firstly the query for selecting all data from the table will be carried out, then this data will be transformed into a Python object, and the length of this object will be found with the help of len. It is highly recommended not to use this method as count will address to a corresponding SQL function COUNT(). With count, an easier query will be carried out in that database and fewer resources will be required for python code performance.

## **16. if queryset is a bad idea**

Do not use queryset as a boolean value: instead of if queryset: do something use if queryset.exists(): do something. Remember, that querysets are lazy, and if you use queryset as a boolean value, an inappropriate query to a database will be carried out.

## **17. Using help\_text as documentation**

Using model help\_text in fields as a part of documentation will definitely facilitate the understanding of the data structure by you, your colleagues, and admin users.

Ex: groups = models.ManyToManyField(Group, verbose\_name=\_('groups'), blank=True, help\_text=\_('The groups this user belongs to. A user will get all permissions granted to each of 'his/her group.'))

## **18. Money Information Storage**

Do not use FloatField to store information about the quantity of money. Instead, use DecimalField for this purpose. You can also keep this information in cents, units, etc.

## **19. Remove \_id**

Do not add \_id suffix to ForeignKeyField and OneToOneField.

## **20. Do not heap all files loaded by user in the same folder**

Sometimes even a separate folder for each FileField will not be enough if a large amount of downloaded files is expected. Storing many files in one folder means the file system will search for the needed file more slowly. To avoid such problems, you can do the following:

def get\_upload\_path(instance, filename):  
 return os.path.join('account/avatars/', now().date().strftime("%Y/%m/%d"), filename)  
  
class User(AbstractUser):  
 avatar = models.ImageField(blank=True, upload\_to=get\_upload\_path)

## **Python standards to follow while writing django models:**

* The Python files use 4 spaces for indentation and the HTML files use 2 spaces.
* Unless otherwise specified, follow [**PEP 8**](https://www.python.org/dev/peps/pep-0008).
* An exception to [**PEP 8**](https://www.python.org/dev/peps/pep-0008) is our rules on line lengths. Don’t limit lines of code to 79 characters if it means the code looks significantly uglier or is harder to read. We allow up to 119 characters as this is the width of GitHub code review; anything longer requires horizontal scrolling which makes review more difficult. This check is included when you run flake8. Documentation, comments, and docstrings should be wrapped at 79 characters, even though [**PEP 8**](https://www.python.org/dev/peps/pep-0008) suggests 72.
* Use underscores, not camelCase, for variable, function and method names (i.e. poll.get\_unique\_voters(), not poll.getUniqueVoters()).
* Use InitialCaps for class names (or for factory functions that return classes).
* In docstrings, follow [**PEP 257**](https://www.python.org/dev/peps/pep-0257). For example:
* def foo():  
   """  
   Calculate something and return the result.  
   """  
   ...
* In tests, use [assertRaisesMessage()](https://docs.djangoproject.com/en/dev/topics/testing/tools/#django.test.SimpleTestCase.assertRaisesMessage) instead of [assertRaises()](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaises) so you can check the exception message. Use [assertRaisesRegex()](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaisesRegex) only if you need regular expression matching.
* In test docstrings, state the expected behavior that each test demonstrates. Don’t include preambles such as “Tests that” or “Ensures that”.
* Reserve ticket references for obscure issues where the ticket has additional details that can’t be easily described in docstrings or comments. Include the ticket number at the end of a sentence like this:
* def test\_foo():  
   """  
   A test docstring looks like this (#123456).  
   """  
   ...
* Field names should be all lowercase, use underscores instead of camelCase.
* For readability, it is a good practice to separate the different properties of the model into **relations**, **attributes**, **manager** and **functions**. Especially if other programmers will have to deal with your code.
* Use a custom object **manager** if you want to define complex queries of that object. For example, you might want to find, several times in your code, the first 5 next tasks the user has to do. You can define this query in your custom manager and use it anywhere in your code more easily.

**In Django Validators can be useful for reusing validation logic between different types of fields:**

* [Validators](https://docs.djangoproject.com/en/1.10/ref/validators/)  reference link
* Built-in validators[¶](https://docs.djangoproject.com/en/1.10/ref/validators/#built-in-validators)

## **Oracle notes**[**¶**](https://docs.djangoproject.com/en/1.10/ref/databases/#oracle-notes)

Django supports Oracle Database Server

### **Naming issues[¶](https://docs.djangoproject.com/en/1.10/ref/databases/#naming-issues)**

### **NULL and empty strings[¶](https://docs.djangoproject.com/en/1.10/ref/databases/#null-and-empty-strings)**

### **TextField limitations**[**¶**](https://docs.djangoproject.com/en/1.10/ref/databases/#id14)

# 

# 

# 

# **Conventions to be followed for CR development:**

1. For every model, create the app using the below command and have static and template folders created for each app

$ python manage.py startapp module

Ex; For resources modules, create a resources app.

Refer [doc](https://docs.google.com/document/d/1-5V2dxd6IXHvcO4Giu_sNB76AYe0V69bEt7t2NHDoHw/edit#) for more information.

1. When we create a model ex: Users, The model 'User' will map to 'myapp\_user' in db. The name of the table myapp\_user, is automatically derived from some model metadata but can be overridden.

See the example below on how to define meta class for creating db tables in CPE\_table name.

**Example:**

class Rule:

base\_rule = models.ForeignKey(RuleHeader)

tech\_description = models.CharField(max\_length=64)

directive = models.CharField(max\_length=64)

.

.

Class Meta():

Db\_table = “CPE\_RULE”

**Output:**

In [1]: from rules.models import \*

In [2]: qu = Rule.objects.all()

In [3]: print qu.query # Restore the original 'query'.

SELECT **"CPE\_RULE**"."ID", "CPE\_RULE"."BASE\_RULE\_ID", "CPE\_RULE"."IS\_ACTIVE", "CPE\_RULE"."TECH\_DESCRIPTION", "CPE\_RULE"."DIRECTIVE", "CPE\_RULE"."SUCCESS\_CRITERIA", "CPE\_RULE"."FAILURE\_CRITERIA", "CPE\_RULE"."CREATED\_BY", "CPE\_RULE"."CREATE\_DATE", "CPE\_RULE"."UPDATED\_BY", "CPE\_RULE"."UPDATE\_DATE" FROM "CPE\_RULE"

**Without meta data:**

In [7]: qu.query

Out[7]: <django.db.models.sql.query.Query at 0x7f3b06e8ad50>

In [8]: print qu.query # Restore the original 'query'.

SELECT "**RULES\_RULE**"."ID", "RULES\_RULE"."BASE\_RULE\_ID", "RULES\_RULE"."IS\_ACTIVE", "RULES\_RULE"."TECH\_DESCRIPTION", "RULES\_RULE"."DIRECTIVE", "RULES\_RULE"."SUCCESS\_CRITERIA", "RULES\_RULE"."FAILURE\_CRITERIA", "RULES\_RULE"."CREATED\_BY", "RULES\_RULE"."CREATE\_DATE", "RULES\_RULE"."UPDATED\_BY", "RULES\_RULE"."UPDATE\_DATE" FROM "RULES\_RULE"

1. Create constants.py file in the app for defining all the constants.

4) Following are the choices to be used:

The first element in each tuple is the actual value to be set on the model, and the second

element is the human-readable name.

* For **is\_active** db field use Choice as below:

is\_active = models.CharField(choices = Is\_Active, default='N', max\_length=16)

**Is\_Active = (**

**('Y', 'Yes'),  
 ('N', 'No'),  
 ('I', 'InProgress'),  
)**

* For **criticality\_type** db field use Choice as below:

name = models.CharField(choices = Criticality\_Type, unique=True,

help\_text=’Criticality type’, max\_length=16)

**Criticality\_Type = (**

**('S','Severe'),**

**('M','Medium'),**

**('L','Low')**

**)**

5) Django default model fields are shown [here](https://docs.djangoproject.com/en/1.10/ref/models/fields/)’

Class based fields and its attributes options being used currently are shown below:

|  |  |
| --- | --- |
| Field Types | Options |
| **IntergerField** | max\_length=(int),  unique=True/False  blank=True/False,  help\_text=\_('Brief Description') |
| **CharField** | max\_length=(int),  unique=True/False  blank=True/False,  help\_text=\_('Brief Description') |
| **BooleanField** | default=False/True  help\_text=\_(''Brief Description') |
| **DateTimeField** | default=timezone.now |
| **PositiveIntegerField** | default=0 |
| [**GenericIPAddressField**](https://docs.djangoproject.com/en/1.10/ref/models/fields/#genericipaddressfield) | null=False,   protocol='both',   unpack\_ipv4=False,   help\_text=\_('Defines the ip address of eachresource'),  validators.ip\_address\_validators(protocol, unpack\_ipv4) |
| **DecimalField** | Max\_digits=6  decimal\_places=2  E.g. cost\_scan= DecimalField(max\_digits=6, decimal\_places=2) |

# References:

<http://steelkiwi.com/blog/best-practices-working-django-models-python/>

<https://docs.djangoproject.com/en/1.10/misc/design-philosophies/>

<https://docs.djangoproject.com/en/dev/internals/contributing/writing-code/coding-style/>

<http://www.marinamele.com/2014/03/django-best-practices-iv-models-forms.html>

<http://django-best-practices.readthedocs.io/en/latest/applications.html>