**SCRIPTING LANGUAGES**

**Django:-ASSIGNMENT**

**USN:-1MS14MCA50**

**NAME:-TAHER FAKHRUDDIN MAKADAM**

**1.** Explain Form Validation and Cleaning?

Sol.

Although forms are generally stateless, they do require some sort of trigger to perform

validation on the data they’ve been bound to, if they’ve been bound at all (validation and

cleaning do not apply to unbound forms).To cause a form to run its validation routines,

you can explicitly call its is\_valid Boolean method, or you can call one of its display

methods (see the following), all which implicitly perform validation as well.

Let’s rejigger our previous add\_relative form processing view, so it handles form

input as well as empty form display.This involves changing the logic around to be a bit

more flexible, with a common Django idiom that checks for the existence of the POST

dict and handles validation (or generates an empty form), and then “falls through” to

displaying the form.The form then displays either for non-POST requests or for POST

requests that failed to validate.

# View’s URL: /person/<id>/children/add/

def add\_relative(request, \*\*kwargs):

# Get the parent relative

# Validate if the form was POSTed

if request.POST:

form = PersonForm(request.POST)

if form.is\_valid:

new\_person = form.save()

return HttpResponseRedirect(new\_person.get\_absolute\_url())

# Otherwise, prep an empty form with the relative pre-filled

else:

relative = get\_object\_or\_404(Person, pk=kwargs[‘id’])

form = PersonForm(initial={‘last’: relative.last})

# Display the form for non POST requests or failed validations.

# Our template will display errors if they exist.

return render\_to\_response(‘person/form.html’, {‘form’: form})

Once validation has been triggered, the form object gains one of two new attributes:

errors, a dictionary of error messages, or cleaned\_data, a dictionary containing the

“clean” versions of the values originally bound to the form.You never find both at once,

as cleaned\_data is only generated when the form validates, and of course errors only

applies if the validation failed.

The format of the errors dictionary is simple; the keys are the field names, and the

values are lists of strings (each string being a message about why the form’s validation

failed). errors, naturally, only contain key/value pairs for fields with errors to display.

Later in the chapter, we explore some helper methods that Form objects provide for easy

display of these error messages.

Behind the concept of “clean” data is the need for input data to be **normalized**—

translated from one or more potential input formats into a consistent output format

appropriate for validation and database storage. For example, forms whose bound data

comes straight from a request.POST dictionary generally contains strings, and thus any

numeric fields’ cleansing process casts those strings to ints or longs, date-related fields

parse strings such as “2007-10-29” into datetime objects, and so forth.

Although normalization is required for the automatic validation and saving methods to

work correctly, it also means any Python code interacting with the form’s contents has

access to the correct data types. If you find the need to examine the original prenormalized

data, it is still available as the form’s data attribute.

**2.** Explain Model inheritance and Abstract Base Classes?

Sol.

A relatively new feature in Django’s ORM at the time of this writing is that of model

inheritance. In addition to foreign key and other relationships between otherwise distinct

model classes, it’s possible to define models which inherit from one another in the same

way that normal, non-ORM Python classes do. (Some examples of which can be found in

Chapter 1,“Practical Python for Django.”)

For example, the previous SmithBook class could be defined not as its own stand-alone

class that just happens to have the same two fields as the Book class, but as an explicit subclass

of Book.The benefits are hopefully obvious—the subclass can then add or override

only the fields that differentiate it from its parent, instead of replicating the entire definition

of the other class.

Our simplistic Book example doesn’t make this sound too exciting, but consider a more

realistic model with a dozen or more attributes and a handful of complex methods, and

suddenly inheritance becomes a compelling way to adhere to Don’t Repeat Yourself

(DRY). Do note, however, that composition—the use of ForeignKey or

OneToOneField—is still a viable alternative! Which technique you use is entirely up to

you and depends a lot on your planned model setup.

Django currently provides two different approaches to inheritance, each with its own

pluses and minuses: **abstract base classes** and **multi-table inheritance**.

**Abstract Base Classes**

The approach of using abstract base classes is, to put it simply,“Python-only” inheritance—

it enables you to refactor your Python model definitions such that common fields

and methods are inherited from base classes. However, at a database and query level, the

base classes don’t exist, and their fields are replicated in the database tables for the children.

This sounds like a violation of DRY, but is actually desirable in scenarios where you

don’t *want* an extra database table for the base class—such as when your underlying

database is legacy or otherwise being used by another application. It’s also just a neater

way to express refactoring of class definitions without implying an actual object hierarchy.

Let’s re-examine (and flesh out) the Book and SmithBook model hierarchy, using

abstract base classes.

class Author(models.Model):

name = models.CharField(max\_length=100)

class Book(models.Model):

title = models.CharField(max\_length=100)

genre = models.CharField(max\_length=100)

num\_pages = models.IntegerField()

authors = models.ManyToManyField(Author)

def \_\_unicode\_\_(self):

return self.title

class Meta:

abstract = True

class SmithBook(Book):

authors = models.ManyToManyField(Author, limit\_choices\_to={

'name\_\_endswith': 'Smith'

})

The key is the abstract = True setting in the Meta inner class of Book—it signifies

that Book is an abstract base class and only exists to provide its attributes to the actual

model classes which subclass it. Note SmithBook only redefines the authors field to provide

its limit\_choices\_to option—because it inherits from Book instead of the usual

models.Model, the resulting database layout has columns for title, genre, and

num\_pages, as well as a many-to-many lookup table for authors.The Python-level class

also has a \_\_unicode\_\_ method defined as returning the title field, just as Book does.

In other words, when created in the database, as well as when utilized for object creation,

ORM querying, and so forth, SmithBook behaves exactly as if it were the following

definition:

class SmithBook(models.Model):

title = models.CharField(max\_length=100)

genre = models.CharField(max\_length=100)

num\_pages = models.IntegerField()

authors = models.ManyToManyField(Author, limit\_choices\_to={

'name\_\_endswith': 'Smith'

})

def \_\_unicode\_\_(self):

return self.title

As mentioned, this behavior extends to the query mechanism as well as the attributes

of SmithBook instances, so the following query would be completely valid:

smith\_fiction\_books = SmithBook.objects.filter(genre=’Fiction’)

Our example isn’t fully suited to abstract base classes, however, you’d typically want to

create both normal Books as well as SmithBooks. Abstract base classes are, of course,

abstract—they cannot be created on their own, and as stated previously, are mostly useful

to provide DRY at the model definition level. Multi-table inheritance, outlined next, is a

better approach for our particular scenario.

Some final notes regarding abstract base classes:The inner Meta class on subclasses is

inherited from, or combined with, that of the parent class (with the natural exception of

the abstract option itself, which is reset to False, as well as some database-specific

options such as db\_name).

In addition, if a base class uses the related\_name argument to a relational field such as

ForeignKey, you need to use some string formatting, so subclasses don’t end up clashing.

Don’t use a normal string, such as "related\_employees", but one with %(class)s in it,

such as "related\_%(class)s" This way, the subclass name is substituted correctly,

and collisions are avoided.