**MAS SOBRE MODELOS Y FORMULARIOS**

**CAMPOS Y VALIDACIONES**

<https://docs.djangoproject.com/en/stable/ref/models/fields/> (para mayor informacion)

Leemos pagina 1 y 2 del Manual del alumno. Vemos los diferentes campos y las validaciones.

1.Partimos del siguiente código:



2. **models.py:** Argumento **Choices (tupla)**: No solo sirve para forms sino también para modelos. Agregamos el siguiente código al modelo **Curso.**

TURNOS = (

        (1, "Mañana" ),

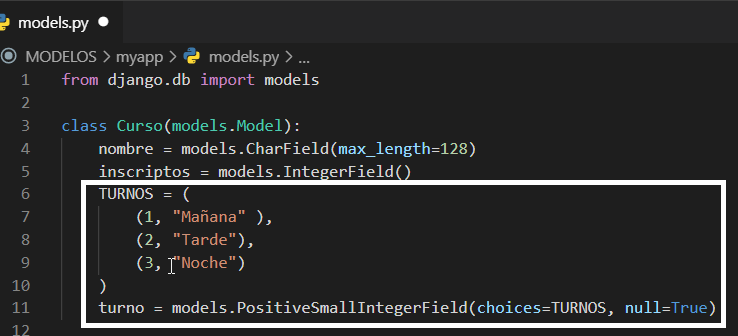
        (2, "Tarde"),

        (3, "Noche")

    )

    turno = models.PositiveSmallIntegerField(choices=TURNOS, null=True)

de forma tal que nos queda:



**null=True** permite que el campo este vacio.

El campo **turno** habíamos indicado que puede tener tres opciones: 1, 2, 3 (“mañana”, “tarde”, “noche”) que debería ver el usuario en lugar de nros. En este manejo la BBDD no interviene.

1. Creamos la migración y aplicamos:

python manage.py makemigrations myapp

python manage.py migrate myapp

1. Ejecutamos nuevamente la consola interactiva:

python manage.py shell

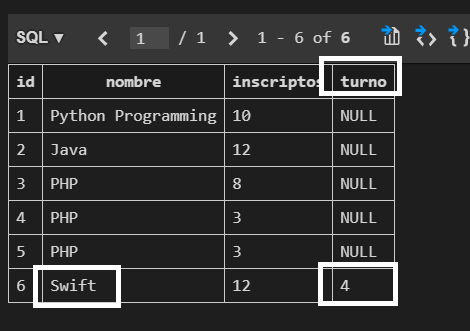
1. Escribimos:

In [1]: from myapp.models import Curso

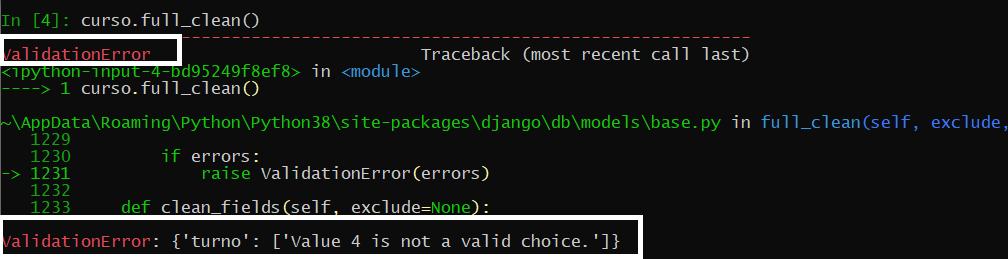
In [2]: curso = Curso(nombre="Swift", inscriptos=12, turno=4)

In [3]: curso.save()

**turno=4** no es una opción valida (ya que puede aceptar de 1 a 3). No obstante **save()** no arroja ningún error y lo inserta correctamente en la BBDD (ver myapp\_curso).



Quien valida correctamente ese valor es Django via **full\_clean()**.



**FILTROS**

1.**filter():** función de un modelo por la cual podemos hacer una consulta a la BBDD con determinadas condiciones.

a. Obtenemos todos los cursos con mas de 10 inscriptos:

(shell)

cursos= Curso.objects.filter(inscriptos\_\_gt=10)

Luego con un for obtenemos la informacion

In [13]: for curso in cursos:

...: print(curso.nombre)

...: print(curso.inscriptos)

Java

12

Swift

12

b. Cursos con < 10 inscriptos: cursos = Curso.objects.filter(inscriptos\_\_lt=10)

In [13]: for curso in cursos:

...: print(curso.nombre)

...: print(curso.inscriptos)

...:

PHP

8

PHP

3

PHP

3

c.Obtenemos los cursos que tienen 10 inscriptos.

cursos = Curso.objects.filter(inscriptos=10)

In [18]: for curso in cursos:

...: print(curso.nombre)

...: print(curso.inscriptos)

...:

Python Programming

10

2. filtros de Django para campos de text (ej ChalField)

contains/icontains

startwith/istartswith

endswith/iendswith

1. Retornar todos los cursos cuyo nombre contiene una “j”:

cursos = Curso.objects.filter(nombre\_\_contains="j")

In [22]: for curso in cursos:

...: print(curso.nombre)

...:

Java

1. Retornar aquellos cursos cuyos nombres empiezan con P:

In [24]: cursos = Curso.objects.filter(nombre\_\_startswith="P")

In [25]: for curso in cursos:

...: print(curso.nombre)

...:

Python Programming

PHP

PHP

PHP

1. Insensitive: Son los filtros que comienzan con mayuscula o minuscula. Ej **icontains, istartswith, iendswith**.

In [26]: **cursos = Curso.objects.filter(nombre\_\_istartswith="p")**

In [27]: for curso in cursos:

...: print(curso.nombre)

...:

Python Programming

PHP

PHP

PHP

In [28]: **cursos = Curso.objects.filter(nombre\_\_icontains="j")**

In [29]: for curso in cursos:

...: print(curso.nombre)

...:

Java

1. **Combinacion** de filtros:

cursos = Curso.objects.filter(nombre\_\_startswith="P", inscriptos\_\_gt=9)

In [33]: for curso in cursos:

...: print(curso.nombre)

...: print(curso.inscriptos)

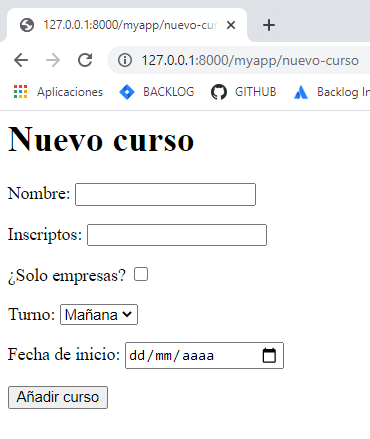
...:

Python Programming

10

**MODELOS y FORMULARIOS**

1.Partimos del siguiente formulario: http://127.0.0.1:8000/myapp/nuevo-curso



2. Django provee un mecanismo llamado **ModelForm** el cual permite generar **formularios** a partir de **modelos** con el objetivo de evitar la repetición de código: tanto en campos y sus tipos como sus validaciones.

Entonces reemplazamos el siguiente código de **forms.py**:

from django import forms

class FormularioCurso(forms.Form): #describe como un formulario funciona y aparece en la web

    nombre = forms.CharField(label="Nombre", max\_length=128)

    inscriptos = forms.IntegerField(label="Inscriptos")

    solo\_empresas = forms.BooleanField(label="¿Solo empresas?", required=False) #va False porque es una casilla de verificacion, no debe aparecer tildada

    TURNOS = (

        (1, "Mañana"),

        (2, "Tarde"),

        (3, "Noche")

    )

    turno = forms.ChoiceField(label="Turno", choices=TURNOS)

    fecha\_inicio = forms.DateField(

        label = "Fecha de inicio",

        widget=forms.DateInput(attrs={"type":"date"})

    )

Por este otro:

from django.forms import ModelForm

from .models import Curso

class FormularioCurso(ModelForm):

    class Meta:

        model = Curso

        fields = ("nombre", "inscriptos", "turno")

Es decir creamos una clase que hereda de *django.forms.ModelForm,* y en su interior otra clase llamada *Meta* donde indicamos dos atributos: el **modelo** a partir del cual queremos generar el formulario(*model*) y la **lista** de campos de ese modelo que queremos que aparezcan en el formulario (*fields*).

3. **views.py , nuevo-curso**: Esta función la usábamos para obtener los datos ingresados en *FormularioCurso* e insertarlos en el modelo *Curso*:

def nuevo\_curso(request): #obtiene los datos ingresados en FormularioCurso y los inserta en el modelo Curso

    if request.method == "POST":

        form = forms.FormularioCurso(request.POST) #crea una instancia de forms

        if form.is\_valid():  #validacion

            Curso.objects.create(

                nombre = form.cleaned\_data["nombre"],

                inscriptos = form.cleaned\_data["inscriptos"]

            )

            return HttpResponseRedirect(reverse("cursos"))

    else:

        form = forms.FormularioCurso()

    ctx = {"form": form}

    return render(request, "myapp/nuevo\_curso.html", ctx)

Puesto que ahora *FormularioCurso* es un **ModelForm,** en el cual indicamos que su modelo fuente es *Curso,* ya no es necesario que obtengamos los datos manualmente via *form.cleaned\_data* ni los insertemos via *Curso.objects.create(),* operaciones que ejecutara automáticamente la función **save()** del formulario. Entonces modificamos **nuevo\_curso** por el siguiente código:

def nuevo\_curso(request): #obtiene los datos ingresados en FormularioCurso y los inserta en el modelo Curso

    if request.method == "POST":

        form = forms.FormularioCurso(request.POST) #crea una instancia de forms

        if form.is\_valid():  #validacion

            form.save()

            return HttpResponseRedirect(reverse("cursos"))

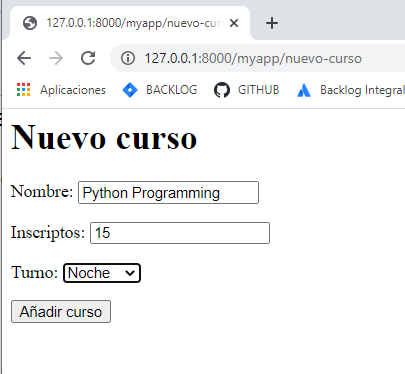
    else:

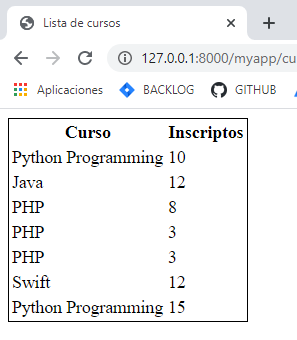
        form = forms.FormularioCurso()

    ctx = {"form": form}

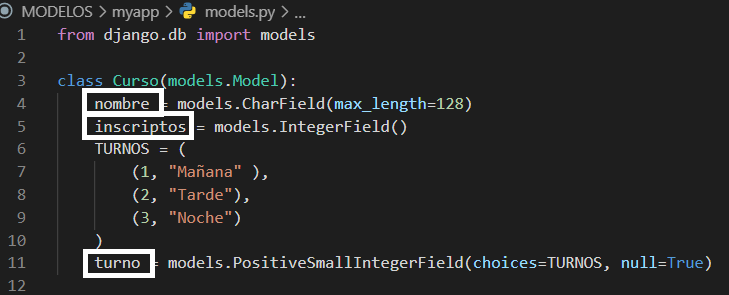
    return render(request, "myapp/nuevo\_curso.html", ctx)

4. <http://127.0.0.1:8000/myapp/nuevo-curso>





La etiqueta de cada uno de los campos del formulario (Nombre, Inscriptos y Turno) fue construida por Django a partir del nombre de los campos del **modelo** (*nombre, inscriptos, turno*).



Entonces si el formulario toma como base los campos del modelo, los debemos explicitar desde el **modelo**.

5. Checkpoint código:



**COMO LE PASAMOS SELECTORES al FRONT DESDE DJANGO usando MODELFORMS**

La idea es similar a cuando usábamos forms. En este caso planteamos una función de inicio la cual vaya planteando **widgets**. Y a través de esos widgets le pasamos selectores y hasta podemos pasar estilo desde django (aunque lo recomendable es pasar solamente selectores y el estilo darselo desde css).

Entonces hacemos el siguiente agregado en **forms.py**.

from django.forms.widgets import TextInput

from myapp.models import Curso

from django import forms

from django.forms.models import ModelForm

# describe como un formulario funciona y aparece en la web

class FormularioCurso(ModelForm):

    class Meta:

        model = Curso

        fields = ("nombre", "inscriptos")

    def \_\_init\_\_(self, \*args, \*\*kwargs):

        super(FormularioCurso, self).\_\_init\_\_(\*args, \*\*kwargs)

        self.fields['nombre'].widget.attrs\

            .update({

                'placeholder' : 'Nombre',

                'class' : 'claseNombre'

            })

        self.fields['inscriptos'].widget.attrs\

            .update({

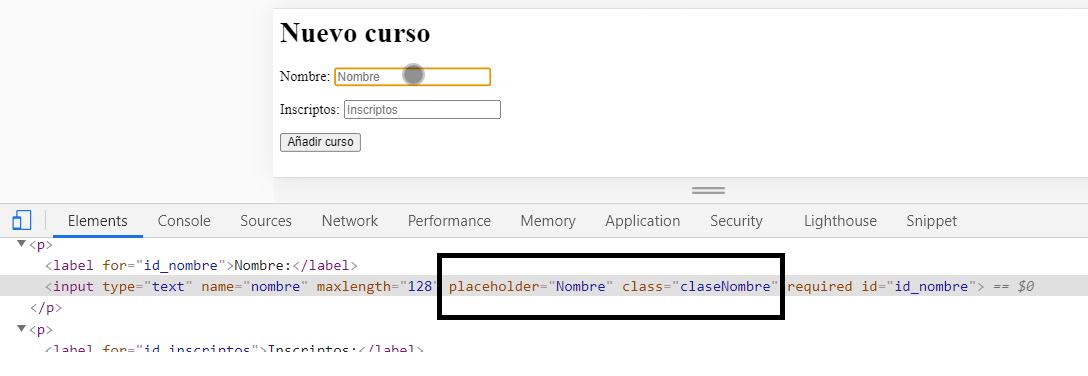
                'placeholder' : 'Inscriptos',

                'class' : 'claseInscriptos'

            })

En donde le ponemos un placeholder y un .class a ambos campos (nombre e inscriptos). Se puede ver desde el F12 como resulta:

<http://127.0.0.1:8000/myapp/nuevo-curso>



Otra forma mas sencilla de hacerlo es usar directamente widgets de esta manera:

class FormularioCurso(ModelForm):

    class Meta:

        model = Curso

        fields = ("nombre", "inscriptos")

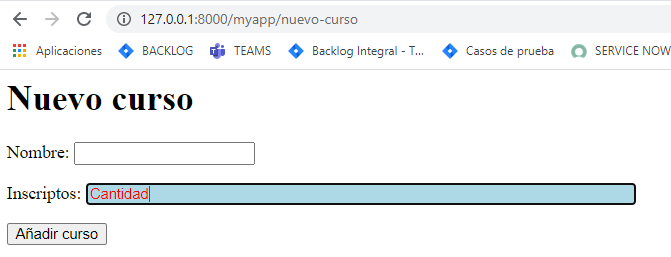
        widgets={

            "nombre" : forms.TextInput(attrs={'placeholder' : 'Ingrese Nombre Curso' , 'class' : 'nombreClase'}),

            "inscriptos" : forms.TextInput(attrs={'style': 'width:500px; color:red; background: lightblue;', 'placeholder': 'Inscriptos'})

        }

En este caso incluso le damos estilo desde django.

****

**AGREGANDO CAMPOS A MODELOS CON DATOS PREEXISTENTES**

1.Supongamos que tenemos en **models.py** el modelo **Curso** de la siguiente manera:

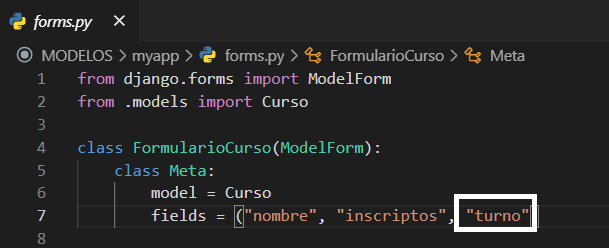
from django.db import models

class Curso(models.Model):

    nombre = models.CharField(max\_length=128)

    inscriptos = models.IntegerField()

debiendo tener que sacar el campo **turno** de **forms.py**:

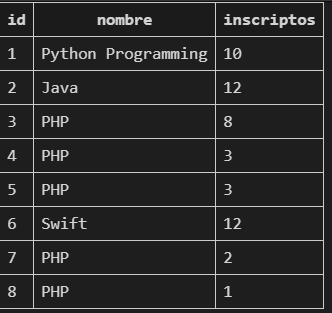


2. Migramos las modificaciones del modelo:

python manage.py makemigrations myapp

python manage.py migrate

3. Y la tabla **myapp\_curso** tiene los siguientes campos/datos:



4. Supongamos ahora que queremos **agregar** el campo **turno** al modelo (con la consecuente creacion en tabla):

TURNOS = {

        (1, "Mañana"),

        (2, "Tarde"),

        (3, "Noche")

    }

    turno = models.PositiveSmallIntegerField("Turno", choices=TURNOS)

de forma tal que el modelo **Curso** nos quede:

from django.db import models

class Curso(models.Model):

    nombre = models.CharField("Nombre", max\_length=128)

    inscriptos = models.IntegerField("Inscriptos")

    TURNOS = {

        (1, "Mañana"),

        (2, "Tarde"),

        (3, "Noche")

    }

    turno = models.PositiveSmallIntegerField("Turno", choices=TURNOS)

5. Hacemos las migraciones correspondientes por los cambios en el **modelo**:

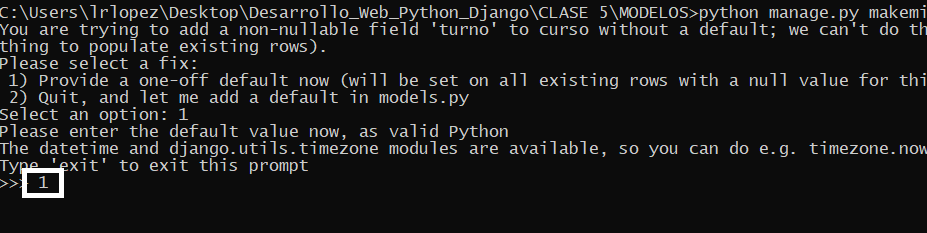
python manage.py makemigrations myapp

Al ejecutar la migración Python nos dice que estamos intentando ingresar el campo no nulo **turno** al modelo Curso, sin un valor por default para los registros previos. La BBDD necesita valores para popular esas rows existentes. Entonces tenemos **dos opciones**:

1. Proveer un valor particular para los campos preexistentes.
2. Salir y agregar manualmente un valor por default. O indicar que **turno** acepte valores nulos.

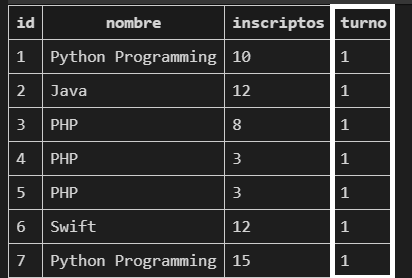
Hay que tener en cuenta que el **valor por defecto** en la definición de un campo para el argumento **null** es **False**. Por lo tanto, no admite nulos.

6. Seleccionamos la opcion **1**. Y luego tipeamos el turno que queremos que tenga: “1”.

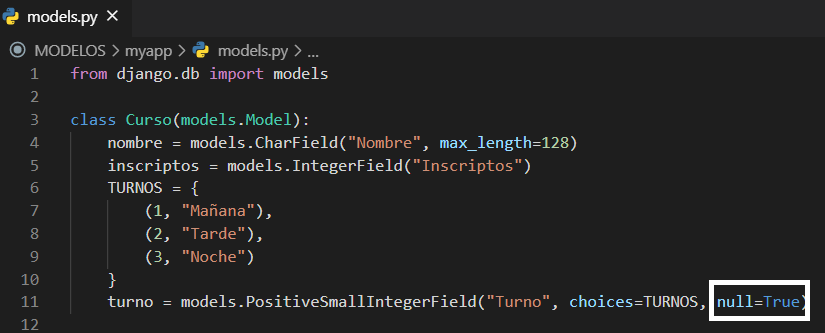


7. Proseguimos con la migración via **python manage.py migrate**.

En la BBDD vemos que a los cursos preexistentes se les agrego el turno 1 (“Mañana”).



8. La otra opción era salir y modificar la definición de **turno** para que acepte valores **nulos**:



1. Borramos el campo **turno** de models.py.

from django.db import models

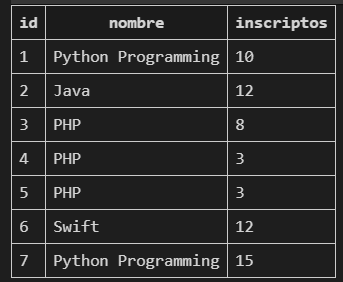
class Curso(models.Model):

    nombre = models.CharField("Nombre", max\_length=128)

    inscriptos = models.IntegerField("Inscriptos")

python manage.py makemigrations myapp

python manage.py migrate myapp



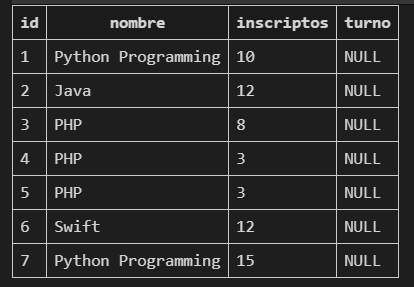
1. Optamos por la segunda opción, agregamos **null=True** a **Curso**.



python manage.py makemigrations myapp

python manage.py migrate myapp

1. Los cursos preexistentes se rellenan con **NULL**.



1. Esto implica que todos esos cursos, como **objectos de Python**, tendrán **None** como valor del campo **turno**:

Abrimos la consola de django=>

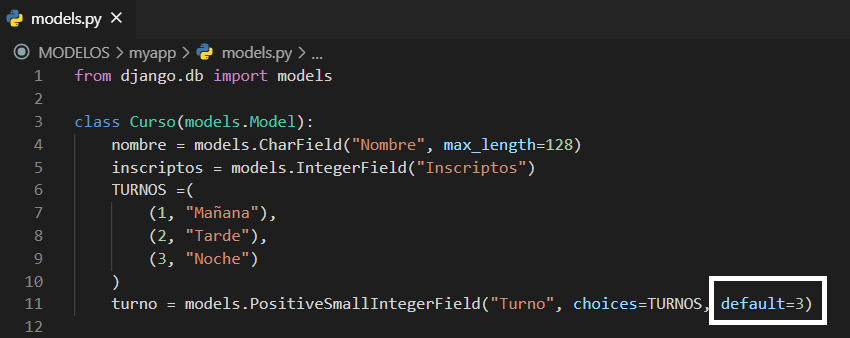
python manage.py shell

from myapp.models import Curso

java = Curso.objects.get(nombre="Java")

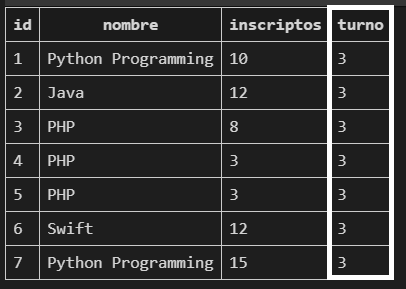
java.turno is None

9. La **3er opción** es también modificar el **models.py** via el argumento **default**, que indica el valor por defecto para un campo:



python manage.py makemigrations myapp

python manage.py migrate myapp



Solo que este mecanismo a diferencia del 1er caso, el valor del argumento **default** será asimismo utilizado en cualquier contexto (incluido los formularios). Por ejemplo cuando se cree un curso en donde no se especifique el campo **turno**.

python manage.py shell

In [1]: from myapp.models import Curso

In [2]: nuevo\_curso = Curso.objects.create(nombre="ANALISIS FUNCIONAL", inscriptos=14)

In [3]: nuevo\_curso.turno

Out[3]: 3

10. Checkpoint código:



**CAMPOS CON REFERENCIA A OTROS MODELOS**

1.Creamos un modelo para guardar información de los profesores. Y lograr que cada curso este asignado a un profesor.

2. Agregamos el modelo **Profesor** a **models.py**:

class Profesor(models.Model):

    nombre = models.CharField(max\_length=128)

    monotributista = models.BooleanField()

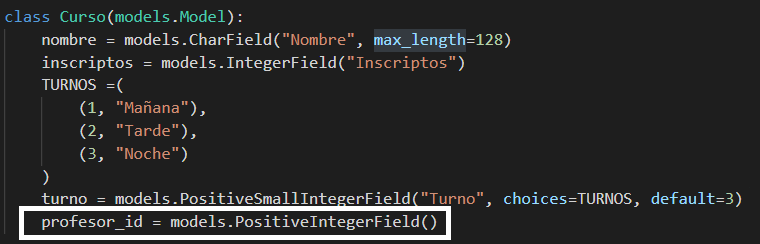
Django agrega automáticamente un campo **id**. Es el identificador único para cada instancia de un modelo.

3. Si cada curso debe tener asignado un profesor, podríamos crear un campo en **Curso** que contenga el ID del mismo:

Agregamos al modelo **Curso** el siguiente campo:

profesor\_id = models.PositiveIntegerField()

de forma tal que nos queda:



Entonces tenemos en el modelo **Curso** una referencia a un objeto de otro modelo **Profesor**.

4. django ofrece una **abstracción** sobre este concepto a través de un campo llamado **ForeignKey**:

En **Curso** modificamos:

profesor\_id = models.PositiveIntegerField()

por esto (le sacamos el \_id al nombre):

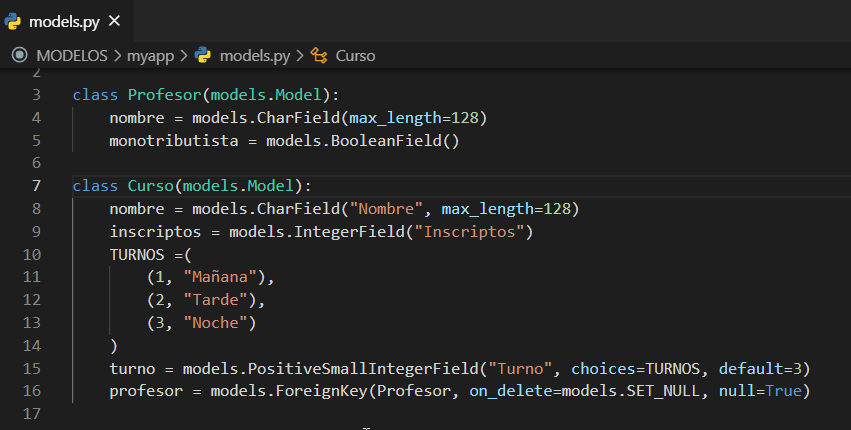
profesor = models.ForeignKey(Profesor, on\_delete=models.SET\_NULL, null=True)

5. **ForeignKey**  toma como argumento el otro modelo al que hace referencia.

**on\_delete** indica lo que debe ocurrir si un profesor es borrado de la base de datos mientras tiene asignado un curso. En ese caso debe quedar vacio (**set\_NULL=True**).

6. Migramos:

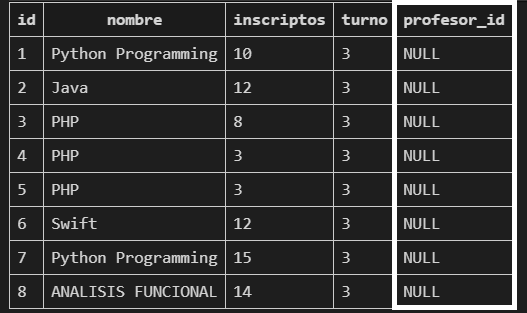
**Models.py** nos debe quedar de esta manera (Profesor 1ro y Curso 2do):



python manage.py makemigrations

python manage.py migrate

7. Resultado:



8. python manage.py shell

In [1]: from myapp.models import Curso, Profesor

In [2]: profesor\_juan = Profesor.objects.create(nombre="Juan", monotributista=True)

In [3]: curso\_testing\_soap = Curso.objects.create(nombre="Testing SOAP", inscriptos=7, profesor=profesor\_juan)

In [4]: curso\_go = Curso.objects.create(nombre="Go", inscriptos=15, profesor=profesor\_juan)

Creamos un profesor y lo asignamos a dos cursos que también creamos.

9. A partir de las instancias de los cursos, podemos acceder a los campos del profesor:

In [6]: curso\_testing\_soap.profesor.nombre

Out[6]: 'Juan'

In [7]: curso\_go.profesor.nombre

Out[7]: 'Juan'

In [8]: curso\_go.profesor.monotributista

Out[8]: True

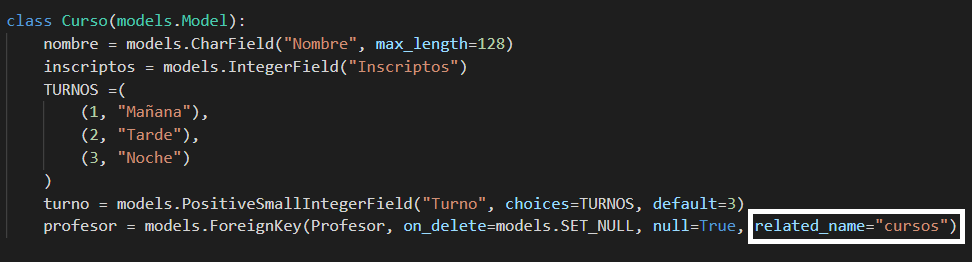
In [10]: curso\_testing\_soap.profesor == curso\_go.profesor

Out[10]: True

10. Tambien podemos obtener todos los cursos asignados a un profesor. Para ello, agregamos el atributo **related\_name** a la definición del campo **ForeignKey** en el modelo **Curso**:

related\_name="cursos")

de forma tal que nos queda:



**Efectuamos las migraciones correspondientes.**

Cerramos el shell, lo volvemos a abrir y ejecutamos:

In [1]: from myapp.models import Profesor

In [2]: profesor\_juan = Profesor.objects.get(nombre="Juan")

In [3]: for curso in profesor\_juan.cursos.all():

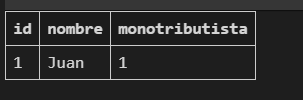
...: print(curso.nombre, curso.inscriptos)

...:

Testing SOAP 7

Go 15

11. Vemos la tabla **myapp\_profesor** creada:



Podemos hacer otro ejemplo cargando un nuevo profesor, un nuevo curso y haciendo las mismas consultas desde el Shell (Como tarea).

11. CheckPoint código:



**EJERCICIO\_1:**

1.Partimos del siguiente código:



2. Partimos del siguiente modelo **Curso**:

class Curso(models.Model):

    nombre = models.CharField("Nombre", max\_length=128)

    inscriptos = models.IntegerField("Inscriptos")

    TURNOS =(

        (1, "Mañana"),

        (2, "Tarde"),

        (3, "Noche")

    )

    turno = models.PositiveSmallIntegerField("Turno", choices=TURNOS)

3. Agregamos la siguiente línea a **urlpatterns** de **urls.py**:

path("cursos/turno-tarde", views.cursos\_turno\_tarde, name="cursos\_turno\_tarde")

4. Creamos **cursos\_turno\_tarde.html**:

<html>

  <h1>Turno tarde</h1>

  <table>

    <thead>

      <tr>

        <th>Nombre</th>

        <th>Inscriptos</th>

      </tr>

    </thead>

    {% for curso in cursos %}

      <tr>

        <td>{{ curso.nombre }}</td>

        <td>{{ curso.inscriptos }}</td>

      </tr>

    {% endfor %}

  </table>

</html>

5. Creamos la vista **cursos\_turno\_tarde** en **views.py**:

def cursos\_turno\_tarde(request):

    cursos = Curso.objects.filter(turno=3) #3 es turno tarde

    ctx = {"cursos": cursos}

    return render(request, "myapp/cursos\_turno\_tarde.html", ctx)

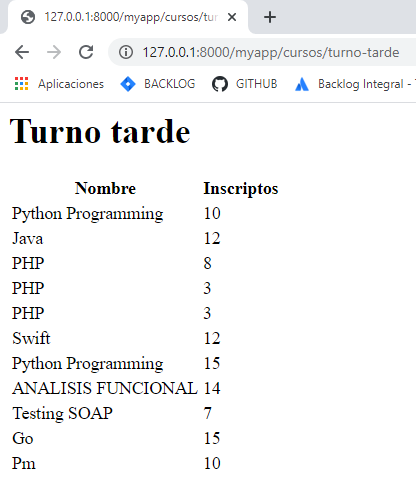
6. Migramos:

python manage.py makemigrations

python manage.py migrate

7. python manage.py runserver.

8. <http://127.0.0.1:8000/myapp/cursos/turno-tarde>



9. Codigo resultante:



**PROYECTO\_INTEGRADOR:**

1.Partimos del siguiente código:



2. Hasta ahora teníamos un modelo **Remera** y un formulario **FormularioRemera**. Según lo visto, podemos ahorrar mucho código si hacemos un **ModelForm** del formulario.

3. Vamos a **forms.py**:

Reemplazamos el formulario **FormularioRemera** por el siguiente código:

from django.forms import ModelForm

from .models import Remera

class FormularioRemera(ModelForm):

    class Meta:

        model = Remera

        fields = ("marca", "talle", "color", "lisa", "genero")

4. Vamos a **models.py** y reemplazamos el modelo **Remera** por el siguiente código:

from django.db import models

class Remera(models.Model):

    marca = models.CharField("Marca", max\_length=128)

    TALLES = (

        (1, "XS"), (2, "S"), (3, "M"), (4, "L"), (5, "XL"), (6, "XXL")

    )

    talle = models.PositiveSmallIntegerField("Talle", choices=TALLES)

    color = models.CharField("Color", max\_length=128)

    lisa = models.BooleanField("Lisa")

    GENEROS = (

        (1, "Hombre"),

        (2, "Mujer"),

        (3, "Unisex")

    )

    genero = models.PositiveSmallIntegerField("Genero", choices=GENEROS)

5. Actualizamos **views.py** la vista **nueva\_remera** que procesa los datos del formulario:

Reemplazamos el código por:

def nueva\_remera(request):

    if request.method == "POST":

        form = FormularioRemera(request.POST)

        if form.is\_valid():

            form.save()

            return HttpResponseRedirect(reverse("index"))

    else:

        form = FormularioRemera()

    ctx = {"form": form}

    return render(request, "shop/nueva\_remera.html", ctx)

6. **models.py.** El atributo **required=False** no será necesario en el modelo para el campo **lisa**.

7. Hacemos las migraciones correspondientes:

python manage.py makemigrations shop

python manage.py migrate shop

Starteamos el servidor

8. Probamos cargar una remera, vamos a <http://127.0.0.1:8000/shop/nueva-remera>:

Y obtenemos el resultado:



Observemos que aun tenemos el problema de que los valores de las columnas **talle** y **genero** se siguen mostrando como números.

Django puede mostrar el texto correspondiente a cada uno de los nros (según la tupla pasada al argumento **choices**) usando en la plantilla {{ fila.get\_columna\_display }}, donde **fila** es una instancia de un modelo y **columna** el nombre del campo definido en ese modelo.

9. Entonces en **index.html** modificamos esto:

…

 <td>{{ remera.talle  }}</td>

…

 <td>{{ remera.genero }}</td>

…

Por esto:

            <td>{{ remera.get\_talle\_display  }}</td>

…

    <td>{{ remera.get\_genero\_display }}</td>

10. <http://127.0.0.1:8000/shop/nueva-remera>:

Y vemos el resultado con el texto en lugar del nro para **Talle** y **Genero.**



11. CheckPoint código:

