Manual básico de Django

# Construir objetos

Como lenguaje orientado a objetos, en Python se pueden crear constructores de objetos. La sintaxis para crear una clase Persona es la siguiente:

class Persona(object):

def \_\_init\_\_(self, nombre, apellido):

self.nombre = nombre

self.apellido = apellido

El método **\_\_init\_\_** es el constructor que acepta dos argumentos: nombre y apellido. Para asignar estos dos argumentos, usamos el otro argumento, **self**, y el punto.

# Levantar el servidor

Nos vamos al directorio que contenga el fichero manage.py y ejecutamos la siguiente orden en el terminal de comandos.

python manage.py runserver

# Plantillas I

¿Qué es una plantilla?

En general es una cadena de texto con código HTML, aunque no siempre.

¿Para qué sirve?

Para separar la lógica del diseño web.

¿Cómo se utilizan?

La forma más habitual es guardar un texto en un documento externo y cargarlo desde la vista.

Uso de plantillas

Se empieza abriendo el documento externo con el método **open()**. Se crea un objeto de tipo **Template** y se le pasa por parámetro el documento abierto y llamando al método **read()**.

doc\_externo = open("C:/Dev/ProyectosDjango/Proyecto1/Proyecto1/plantillas/miplantilla.html")

plt = Template(doc\_externo.read())

A continuación, se crea un objeto de tipo **Context** y se llama al método **render()** del objeto **Template**.

doc\_externo.close()

ctx = Context()

documento = plt.render(ctx)

La vista completa quedaría así:

def saludo(request): #primera vista

doc\_externo = open("C:/Dev/ProyectosDjango/Proyecto1/Proyecto1/plantillas/miplantilla.html")

plt = Template(doc\_externo.read())

doc\_externo.close()

ctx = Context()

documento = plt.render(ctx)

return HttpResponse(documento)

# Plantillas II

Variables en las plantillas

Para esto, necesitamos un diccionario (como un mapa en Java).

nombre = "Juan"

apellido = "Díaz"

ahora = datetime.datetime.now()

diccionario = {"nombre\_persona": nombre, "apellido\_persona": apellido, "momento\_actual": ahora}

Este diccionario se lo pasaremos al contexto cuando lo construyamos para que las plantillas dispongan de estas variables.

ctx = Context(diccionario)

Luego, en la plantilla HTML, usaremos las claves para hacer referencia a lo que contengan. Si es una variable, se hará referencia al contenido de esa variable.

Para llamar a una clave en HTML, usaremos dobles llaves.

<p>El nombre del profesor es {{nombre\_persona}} y su apellido es {{apellido\_persona}}</p>

Si además la variable es un objeto, como en el caso de la variable **ahora**, podemos usar POO y el punto para llamar a métodos y miembros de ese objeto. Por ejemplo, si en vez de la fecha completa queremos el día, escribiríamos **ahora.day**. Esto puede hacerse directamente en la plantilla HTML.

<p>Estamos viendo el tema de las plantillas a d&iacute;a {{momento\_actual.day}} / {{momento\_actual.month}} / {{momento\_actual.year}}</p>

Nomenclatura del punto

Ya hemos visto que se puede usar un punto para acceder a un método o propiedad de un objeto, pero igualmente se puede usar para acceder a un elemento dentro de una lista.

temas\_del\_curso = ["Plantillas", "Modelos", "Formularios", "Vistas", "Despliegue"]

Si tenemos esta lista, podemos acceder al elemento 1 en una plantilla usando un punto de la siguiente forma:

{{temas\_del\_curso.1}}

Esta sintaxis no es válida en django ni en Python.

Orden

Cuando django se encuentra con un punto en una plantilla, internamente busca un elemento que pueda dar significado a ese punto. El orden es el siguiente:

1. Diccionario
2. Atributo
3. Método
4. Índice de lista

Bucle for en plantillas

Si tengo una lista llamada **temas** y quiero recorrerla desde la plantilla, se puede hacer usando un bucle for. Supongamos que queremos imprimir en pantalla la lista desordenada.

<ul>

{% for t in temas %}

<li>{{t}}</li>

{% endfor %}

</ul>

Este tipo de bucle es un **foreach** y en cada iteración la variable se ha llamado **t**.

Estructuras de control en plantillas

Igual que los bucles **for**, se pueden usar estructuras de control en las plantillas. Además, de forma similar a JS, el **if** no solo comprobará que la variable existe, sino también que contenga información.

{% if temas %}

<ul>

{% for t in temas %}

<li>{{t}}</li>

{% endfor %}

</ul>

{% else %}

<p>No hay elementos que mostrar</p>

{% endif %}

Además, a la hora de usar los **if**, se pueden usar operadores de comparación de forma normal como en cualquier otro lenguaje.

{% if nombre\_persona == 'Paco' %}

<p>El curso lo imparte el profesor titular.</p>

{% endif %}

Comentarios

En las plantillas se pueden insertar comentarios de una línea o de varias.

{# Esto es un comentario de una línea #}

{% comment %}

Comentario de varias líneas

{% endcomment %}

Filtros

Un filtro sirve para aplicar alguna propiedad a una variable. Se coloca un *piper* | (en inglés, tubería) y a continuación la propiedad que se quiere aplicar. Por ejemplo, si queremos que una palabra se pase a mayúsculas o minúsculas, tenemos los filtros **upper** y **lower**, respectivamente.

<p>El nombre del profesor es {{nombre\_persona|upper}} y su apellido es {{apellido\_persona|lower}}</p>

Un filtro se puede encadenar a continuación de otro para que sus efectos se apliquen. Por ejemplo, el filtro **first** toma la primera letra de una palabra, y si luego aplicamos el filtro **lower**, pasará esa letra a minúscula.

{% for t in temas %}

<li>{{t|first|lower}}</li>

{% endfor %}

Para más información acerca de filtros, mirar la página <https://docs.djangoproject.com/en/2.2/ref/templates/builtins/>

# Plantillas 3: Cargadores de plantillas

Hay que comenzar importando el **loader** en el fichero con la vista que vamos a usar:

from django.template import loader

Luego, en el fichero **settings.py**, vamos a añadir el directorio donde almacenaremos las plantillas.

TEMPLATES = [

{

'BACKEND': 'django.template.backends.django.DjangoTemplates',

'DIRS': [**'C:/Dev/ProyectosDjango/Proyecto1/Proyecto1/plantillas**'],

'APP\_DIRS': True,

'OPTIONS': {

'context\_processors': [

'django.template.context\_processors.debug',

'django.template.context\_processors.request',

'django.contrib.auth.context\_processors.auth',

'django.contrib.messages.context\_processors.messages',

],

},

},

]

De vuelta en la vista donde queremos cargar la plantilla, usaremos el método **get\_templates** de la clase **loader** para cargar la plantilla que queramos.

doc\_externo = loader.get\_template('miplantilla.html')

En este caso, tenemos un objeto **doc\_externo** de un tipo **Template** distinto del que teníamos antes. Es por esto que el método **render** no admitirá un contexto igual que hemos hecho anteriormente. Lo bueno es que admite el diccionario directamente, por lo que nos ahorramos crear ese código.

class Persona(object):

def \_\_init\_\_(self, nombre, apellido):

self.nombre = nombre

self.apellido = apellido

def saludo(request):

p1 = Persona('Juan', 'Díaz')

temas = ["Estadística univariante", "Estadística bivariante", "Cálculo de probabilidades"]

diccionario = {

'nombre\_persona': p1.nombre,

'apellido\_persona': p1.apellido,

'temas': temas

}

ahora = datetime.datetime.now()

doc\_externo = get\_template('plantilla1.html')

documento = doc\_externo.render(diccionario)

return HttpResponse(documento)

Se puede condensar aún más el código si importamos el **get\_template** en vez del **loader**.

from django.template.loader import get\_template

doc\_externo = get\_template('miplantilla.html')

Esto es más recomendable cuando vayamos a trabajar con muchas plantillas.

# Módulo shortcuts

Del módulo shortcuts nos va a interesar la función render, que nos permite simplificar aún más el código.

A la hora de devolver algo en una vista, hemos estado devolviendo un HttpResponse con un cierto documento. Esto ha implicado crear variables de un cierto tipo, contextos, etc. que consumen memoria y visualmente está más recargado.

En el módulo **shortcuts** contamos con la función **render** (documentación [aquí](https://docs.djangoproject.com/en/2.2/topics/http/shortcuts/)). El primer parámetro es una **request**, que es obligatoria. El segundo parámetro, que también es obligatorio, es el nombre de la plantilla que vamos a cargar. Hay que recordar haber modificado el fichero settings.py para que django sepa dónde buscar las plantillas. El tercer parámetro es el diccionario que usa esa plantilla. De modo que la línea de código en cuestión quedaría así:

return render(request, "miplantilla.html", diccionario)

# Incluir un HTML dentro de otro

Supongamos que queremos meter una botonera fija dentro de varias plantillas HTML. Para no tener que hacerlo “a pelo” en cada plantilla, django nos permite usar la etiqueta **include**. Supongamos que tenemos la botonera guardada en una plantilla **barra.html** que tiene este aspecto:

<html>

<style>

#barra {

margin: 0;

padding: 0;

list-style: none;

text-transform: uppercase;

}

#barra li {

display: inline;

margin: 0 20px;

}

</style>

<body>

<ul id="barra">

<li>Home</li>

<li>Servicios </li>

<li>Quienes somos</li>

<li>Contactos</li>

<li>Acerca de</li>

</ul>

</body>

</html>

Nota: los estilos son escuetos, pero sobre todo nos interesa el **display: inline** para que los botones aparezcan en horizontal y no en vertical.

Ahora veamos cómo se introduciría esta botonera en el fichero miplantilla.html:

<html>

<body>

{% include "barra.html" %}

<h1 style="color: brown">

Hola alumnos, estamos viendo nuestra primera plantilla

</h1>

(más código HTML)

De este modo, django insertará el código del fichero barra.html en el fichero **miplantilla.html** justo al principio del **body**.

# Herencia de plantillas

De forma similar a la POO, en django existe el concepto de herencia de plantillas. La idea es que podemos tener una plantilla padre con una cabecera y un pie de página y después un bloque central que diremos que es cambiante. De este modo, todos los hijos compartirán la cabecera y el pie de página, pero cada uno tendrá su bloque central.

La etiqueta clave aquí es la etiqueta **extends**:

{% extends “padre.html” %}

El aspecto de la plantilla padre sería el siguiente:

<html>

<head>

<title>

{% block title %} {% endblock %}

</title>

<style>

#titulo {

background-color: darkgreen;

color: white;

text-align: center;

}

#pie {

background-color: darkgreen;

color: white;

text-align: right;

}

</style>

</head>

<body>

<h1 id="titulo">Píldoras informáticas</h1>

{% include "barra.html" %}

<h3>Cursos de programación, ofimática y diseño web</h3>

{% block content %} {% endblock %}

<p id="pie">Cursos gratuitos, gracias por su visita.</p>

</body>

</html>

Queremos que el título y el contenido sean cambiantes cuando las hijas hereden de esta plantilla. Para eso utilizamos las etiquetas **{% block *title* %} {% endblock %}** para el título y **{% block *content* %} {% endblock %}** para el contenido. Las palabras en cursiva son escogidas por el usuario, pero es importante que se preserven luego en las plantillas hijas.

Las hijas deben comenzar por la etiqueta **extends** para heredar el código de la plantilla padre. Después, usaremos de nuevo **{% block *title* %} {% endblock %}** para escribir el título y **{% block *content* %} {% endblock %}** para el código del contenido. En el caso de algo que está en el **body**, puede ser tanto texto plano como más contenido html.

{% extends "base.html" %}

{% block title %}

Curso C#

{% endblock %}

{% block content %}

<p>Estamos a día: {{dameFecha}}</p>

<iframe width="560" height="315" src="https://www.youtube.com/embed/d1-UX6aR\_TM" frameborder="0"

allow="accelerometer; encrypted-media; gyroscope; picture-in-picture" allowfullscreen></iframe>

{% endblock %}

Aquí se ha llamado a la página Curso C# y en el contenido se ha introducido la fecha y un cierto vídeo. Ahora toca ir al fichero **views.py** y crear una nueva vista. Usaremos el método **render** visto anteriormente para simplificar el código al máximo:

def curso\_c(request):

fecha\_actual = datetime.datetime.now()

return render(request, "cursoc.html", {"dameFecha": fecha\_actual})

Recordar que **render** necesitaba 3 parámetros: la request, la plantilla que queríamos mostrar y de forma opcional, un diccionario con los términos que aparecen en nuestra plantilla. Como la plantilla que estamos usando lleva el término **dameFecha**, necesitamos introducir un diccionario si no queremos problemas luego (en este caso, no aparecería la fecha).

Por último, nos vamos al fichero **urls.py** para indicar a django qué URL va a ser la que nos lleve a esta página. Comenzaremos importando la vista mediante el import correspondiente y luego, como siempre, indicamos la URL y la vista a la que conduce:

from Proyecto1.views import curso\_c

urlpatterns = [

…

path('curso\_c/', curso\_c),

…

]

Si quisiéramos hacer ahora otra plantilla hija, el proceso sería similar. Comenzamos heredando de **base.html**, modificando el título y el contenido.

{% extends "base.html" %}

{% block title %}

Curso C#

{% endblock %}

{% block content %}

<p>Estamos a día: {{dameFecha}}</p>

<iframe width="560" height="315" src="https://www.youtube.com/embed/HmPpMreucyc" frameborder="0"

allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"

allowfullscreen></iframe>{% endblock %}

Luego crearíamos la vista:

def curso\_css(request):

fecha\_actual = datetime.datetime.now()

return render(request, "cursocss.html", {"dameFecha": fecha\_actual})

Y terminamos añadiendo la URL al fichero **urls.py**:

from Proyecto1.views import curso\_css

urlpatterns = [

…

path('curso\_css/', curso\_css),

…

]

# Bases de datos

Primero hay que recalcar que en django tenemos proyectos donde cada proyecto consta de una o varias aplicaciones. Estas aplicaciones son las que acceden a bases de datos. Para el ejemplo, crearemos un proyecto **TiendaOnline** que será una página web, y dentro de este proyecto, una aplicación **gestionPedidos** que será la que acceda a la base de datos.

Comenzamos creando el proyecto con django. En la terminal, tecleamos la orden

django-admin startproject TiendaOnline

Con eso se crea la carpeta del proyecto con la subcarpeta del mismo nombre. Vamos dentro de la primera carpeta y tecleamos la siguiente orden:

Cd TiendaOnline

python manage.py startapp gestionPedidos

Esto creará la carpeta **gestionPedidos** con un par de subcarpetas y varios ficheros Python dentro. En el que nos centraremos ahora será **models.py**, que es donde especificaremos qué tablas queremos crear.

Vamos al modelo y creamos una clase por cada tabla. Estas clases tendrán de parámetro **models.Model**. Cada campo será de la clase **models.\***, donde **\*** será **CharField** para texto, **IntegerField** para número, **EmailField**, **DateField** o **BooleanField**. Algunos, como CharField, admiten además parámetros.

class Clientes(models.Model):

nombre = models.CharField(max\_length=30)

direccion = models.CharField(max\_length=50)

email = models.EmailField()

telefono = models.CharField(max\_length=9)

class Articulos(models.Model):

nombre = models.CharField(max\_length=30)

seccion = models.CharField(max\_length=20)

precio = models.IntegerField()

class Pedidos(models.Model):

numero = models.IntegerField()

fecha = models.DateField()

entregado = models.BooleanField()

Ahora debemos decirle a django que esta aplicación forma parte del proyecto **TiendaOnline**. Para ello, nos vamos al fichero **settings.py** y en la lista **INSTALLED\_APPS** añadimos al final el nombre de nuestra nueva aplicación.

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

**'gestionPedidos',**

]

Para ver que todo está bien, podemos escribir en la terminal

python manage.py check gestionPedidos

Para crear el texto que crearía las tablas, tenemos que usar los siguientes dos comandos:

python manage.py makemigrations

Este primer comando nos da un número identificador que tenemos que usar a continuación:

python manage.py sqlmigrate gestionPedidos 0001

Y de aquí sale el código SQL para crear las tablas:

BEGIN;

--

-- Create model Articulos

--

CREATE TABLE "gestionPedidos\_articulos" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "nombre" varchar(30) NOT NULL, "seccion" varchar(20) NOT NULL, "precio" integer NOT NULL);

--

-- Create model Clientes

--

CREATE TABLE "gestionPedidos\_clientes" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "nombre" varchar(30) NOT NULL, "direccion" varchar(50) NOT NULL, "email" varchar(254) NOT NULL, "telefono" varchar(9) NOT NULL);

--

-- Create model Pedidos

--

CREATE TABLE "gestionPedidos\_pedidos" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "numero" integer NOT NULL, "fecha" date

NOT NULL, "entregado" bool NOT NULL);

COMMIT;

Para no tener que aplicar nosotros estas queries, usamos la siguiente orden y ya django se encarga de hacerlo él solo:

python manage.py migrate

## Insertar, actualizar, borrar

Comenzaremos viendo cómo se ejecutan estas instrucciones en la consola. Abrimos el Shell e importamos el modelo que vamos a usar, que en este caso es Artículos.

python manage.py Shell

from gestionPedidos.models import Articulos

Para insertar, tenemos dos opciones: crear una variable con el modelo y usar el método **save()**, o bien usar el método **Articulos.objects.create()** y pasarle los valores que queremos insertar:

>>> art = Articulos(nombre='mesa', seccion='decoracion', precio=90)

>>> art.save()

>>> art2 = Articulos(nombre='camisa', seccion='confeccion', precio=75)

>>> art2.save()

>>> art3 = Articulos.objects.create(nombre='taladro', seccion='ferreteria', precio=65)

Para modificar un registro, modificamos la variable donde está contenido y de nuevo usamos el método **save()**

>>> art.precio = 95

>>> art.save()

Para eliminar un registro, lo recuperamos mediante el método **Articulos.objects.get()** e introducimos el criterio para recuperar, en nuestro caso, el **id**.

>>> art5 = Articulos.objects.get(id=2)

>>> art5.delete()

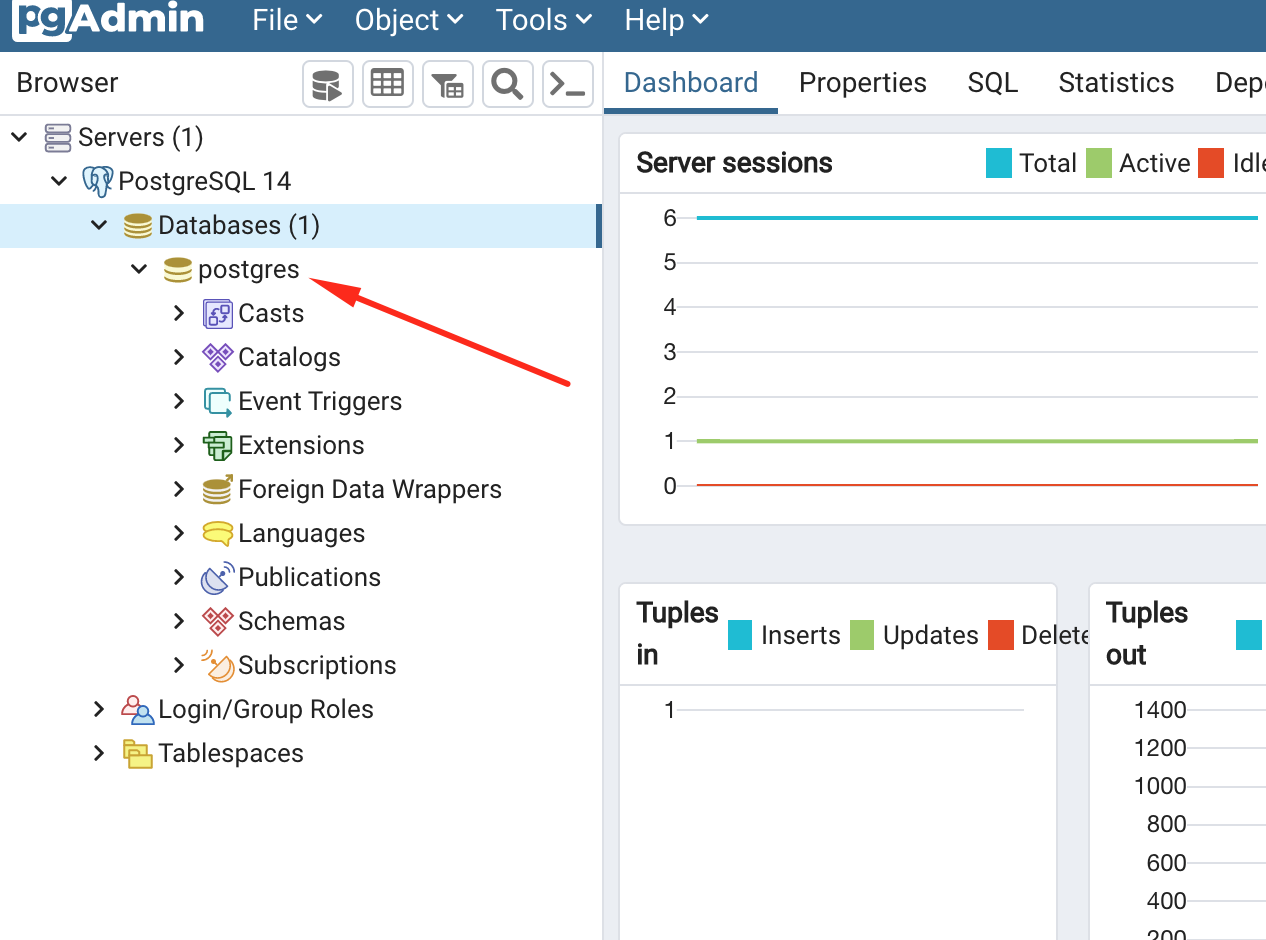
Si queremos hacer un **select**, tenemos que usar el método **Articulos.objects.all()**, que nos devuelve una lista. Para ver la **query** como tal, usamos el método **query** y, a continuación, el método **\_\_str\_\_()**.

>>> lista = Articulos.objects.all()

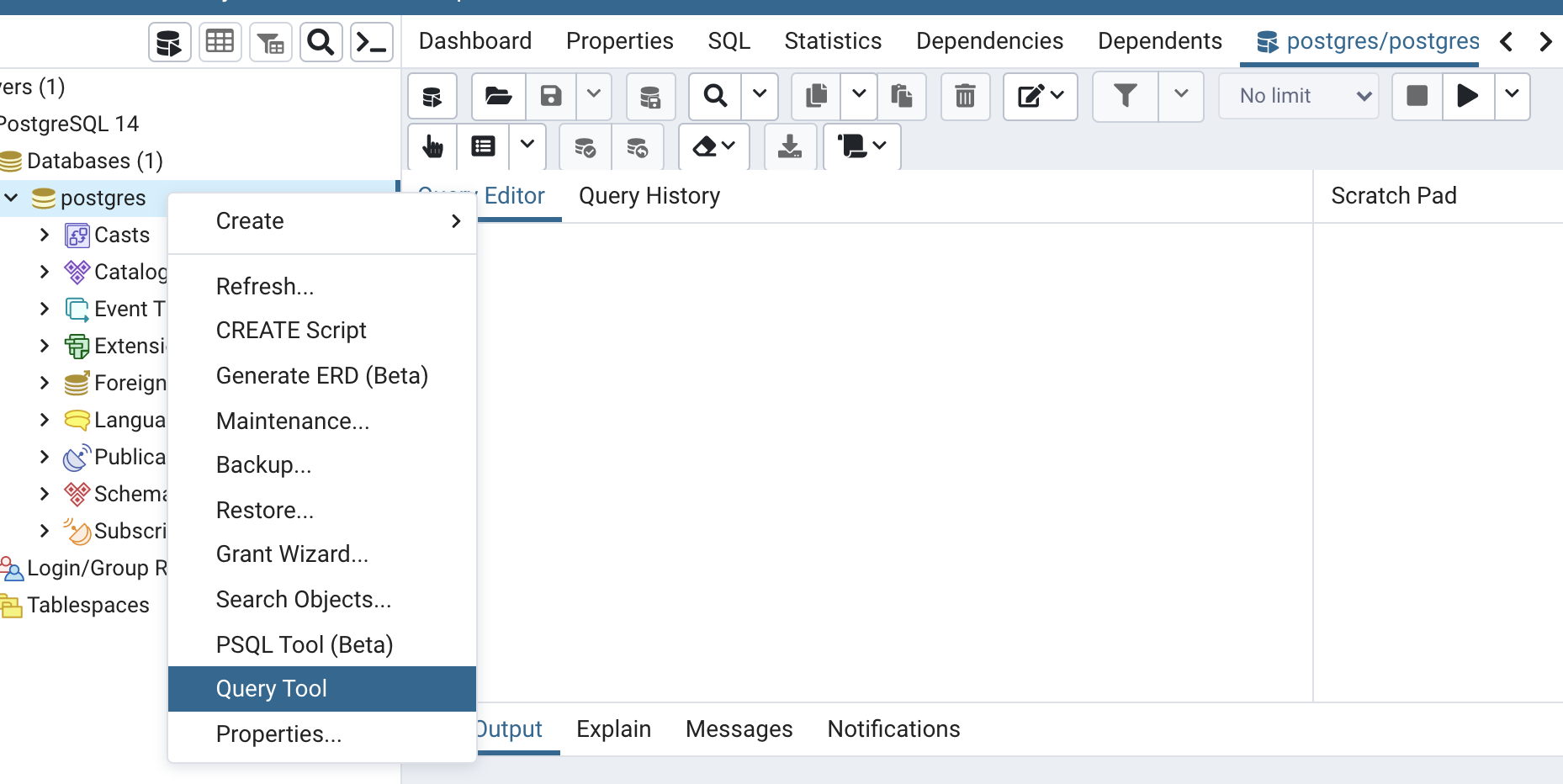
>>> lista.query.\_\_str\_\_()

# PostgreSQL

Nos vamos a la página de PostgreSQL y descargamos el instalador. Nos pedirá una contraseña para acceder a la base de datos y un puerto desde el que desplegarla. Una vez instalado, buscamos la aplicación pgAdmin4. Ya hay una base datos por defecto.



Click derecho sobre ella y seleccionamos ‘Query tool’ para introducir nuestras queries.



Introducimos la siguiente query para crear una base de datos y pulsamos el botón .

create database EnciclopediaEstadistica

**IMPORTANTE**: la base de datos se crea en minúsculas. Esto será importante a continuación, en la configuración del acceso a la base de datos. Si quisiéramos usar mayúsculas y minúsculas, habría que escribir entre comillas: “EnciclopediaEstadistica”.

Para ver la nueva base de datos, actualizamos el panel de la izquierda con el botón derecho sobre “Databases”. Las tablas estarán dentro de schemas.

Conectamos el proyecto de Django con la base de datos. Primero, debemos instalar una librería llamada psycopg2:

pip install psycopg2

Cambiamos la configuración en settings.py. La que viene por defecto es para sqlite3, ponemos la de postgresql:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql\_psycopg2',

'NAME': 'EnciclopediaEstadistica',

'USER': 'postgres',

'PASSWORD': '\*\*\*\*\*\*\*\*',

'HOST': '127.0.01',

'DATABASE\_PORT': '5432',

}

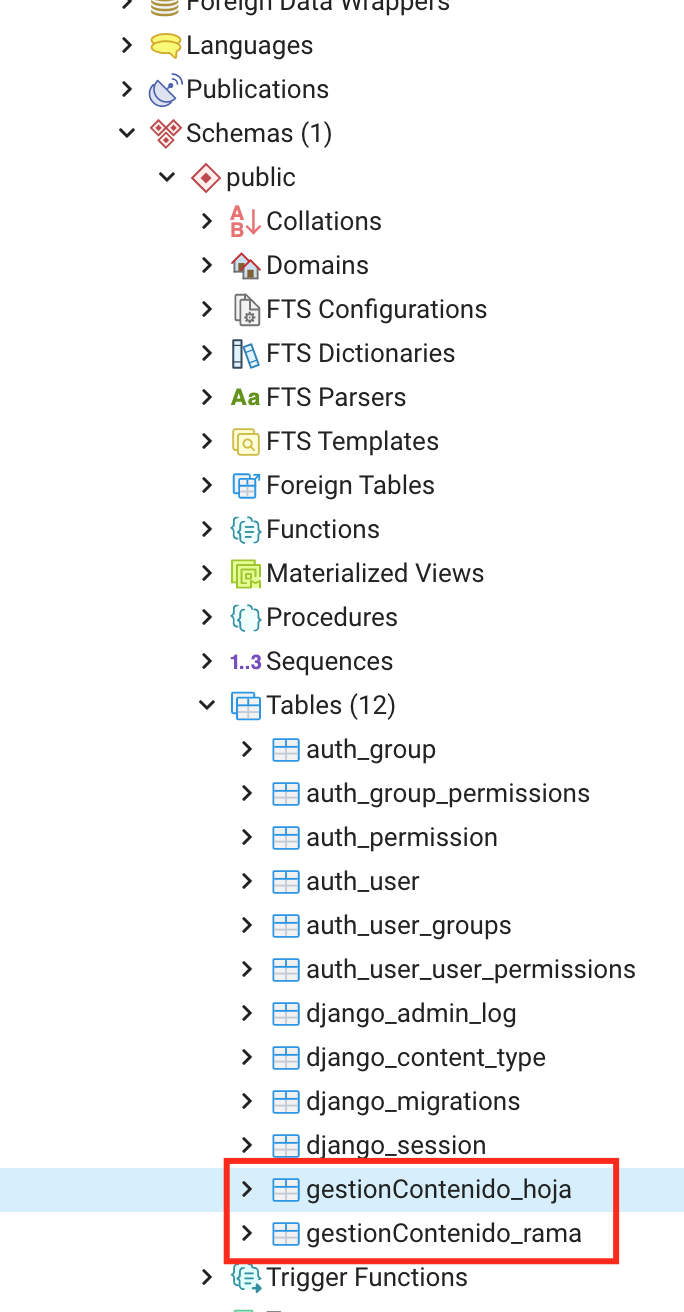
}

Comenzamos haciendo las migraciones. En el directorio, hacemos las migraciones en la terminal. Esto es necesario hacerlo cada vez que se cambie el modelo. Para hacer efectivas las migraciones, ejecutamos esto:

python3 manage.py makemigrations

python3 manage.py migrate

Ya estarían nuestras tablas creadas.



Para crear el primer registro, entramos de nuevo en el Shell:

python manage.py shell

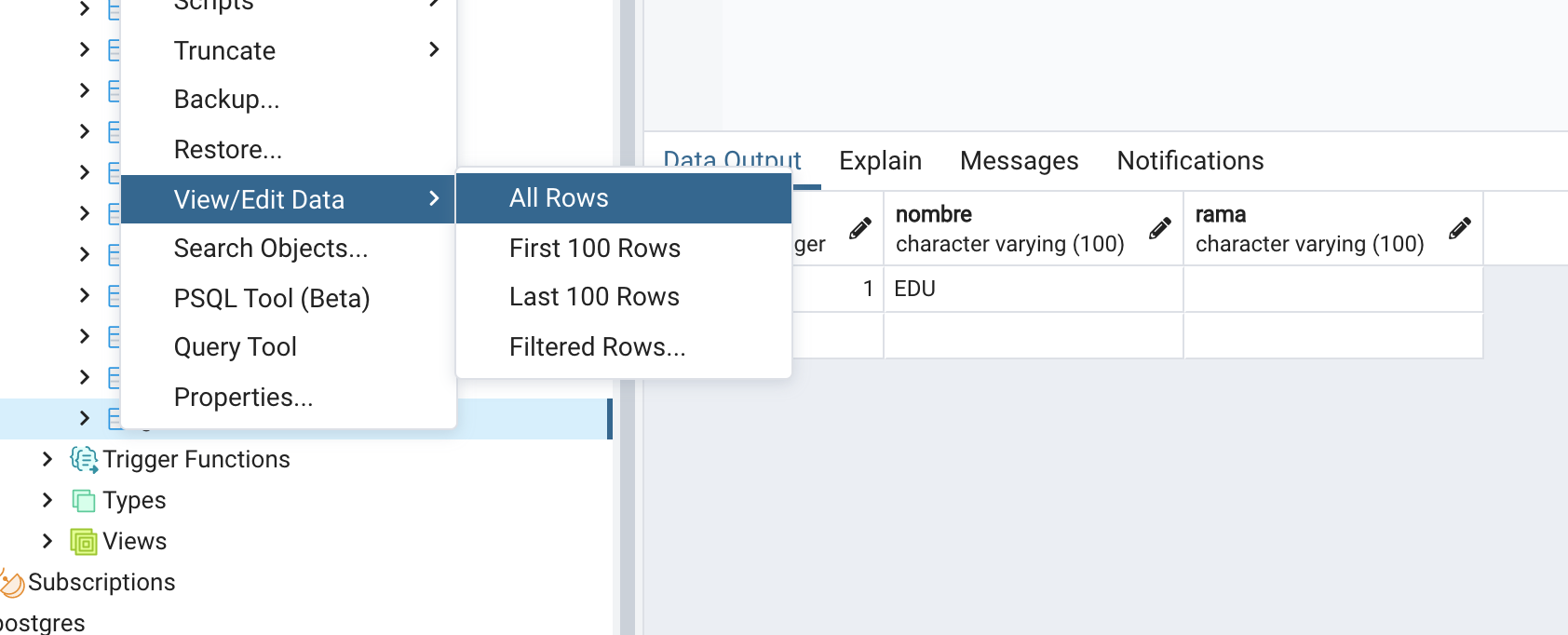
Una vez dentro, importamos el modelo Rama, creamos una instancia y la guardamos:

from gestionContenido.models import Rama

ram = Rama(nombre='EDU', rama='')

ram.save()

Y ya la Podemos ver en el pg4:



Para realizar una consulta con un WHERE, lo haríamos dentro del shell, importando primero el modelo (la rama, en este caso) y usando el método filter. Si queremos añadir más criterios, usaremos una coma para separarlos.

from gestionContenido.models import Rama

Rama.objects.filter(rama='Estadística descriptiva univariante')

Rama.objects.filter(rama='Estadística descriptiva univariante', nombre = 'Medidas de posición')

Nota: los símbolos < y > están restringidos. En vez de escribir precio > 100, escribiríamos precio\_\_gte 100 y en vez de precio < 100 escribiríamos precio\_\_lte 100. Para una horquilla de valores, podemos usar los símbolos gte y lte o bien precio.\_\_range(minimo, máximo).

Si no implementamos el método \_\_str\_\_ en la clase de Python, la respuesta será ininteligible. Por tanto, la clase debería tener el método implementado de la siguiente manera:

class Rama(models.Model):

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

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

def \_\_str\_\_(self):

return 'El nombre es <<%s>>, la rama es <<%s>>' % (self.nombre, self.rama)

Nótese el operador % para incluir los parámetros dentro de la cadena y el uso de la palabra self para hacer referencia al valor del objeto que llama.

Nota: para que estos cambios se hagan efectivos, hay que realizar de nuevo las migraciones.

Para ordenar, es decir, usar un ORDER BY, tendríamos que concatenar a la orden filter o all la función order\_by(‘campo’). Si delante del campo colocamos un signo -, se ordena al revés.

Rama.objects.filter(rama='Estadística descriptiva univariante').order\_by('nombre')

Rama.objects.filter(rama='Estadística descriptiva univariante').order\_by('-nombre')

# Panel de administración

Django crea por defecto un panel de administración al que se accede por <http://127.0.0.1:8000/admin/login>

Primero tendremos que crear un súperusuario con nombre y contraseña. En la terminal, nos vamos al directorio y escribimos

python3 manage.py createsuperuser

Nos preguntará nombre, email y contraseña. Si esta es muy débil, nos lo indicará, pero aún así, nos dejará crearla:

Username (leave blank to use 'enriquenaranjoguerra'): enriqueuca

Email address: enrique.naranjo@uca.es

Password: 123456

Password (again): 123456

This password is too short. It must contain at least 8 characters.

This password is too common.

This password is entirely numeric.

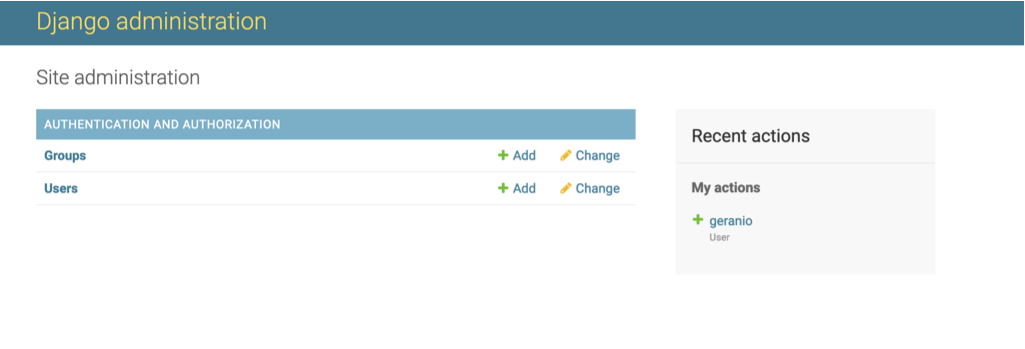
Bypass password validation and create user anyway? [y/N]: y

Superuser created successfully.

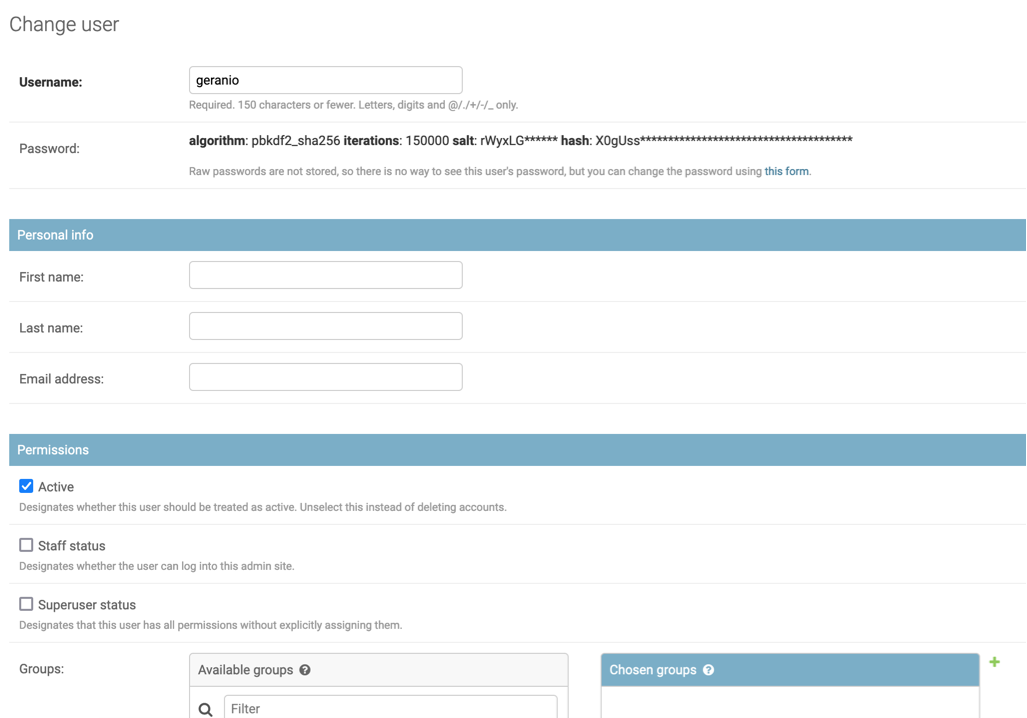
Nota: en el momento de escribir esto, tuve que downgradear la versión de psycopg2-binary usando esta orden en la terminal:

pip install psycopg2-binary==2.8.6

De primeras, el panel estará más vacío que una biblioteca en carnavales:



Podremos agregar usuarios y organizarlos en grupos.



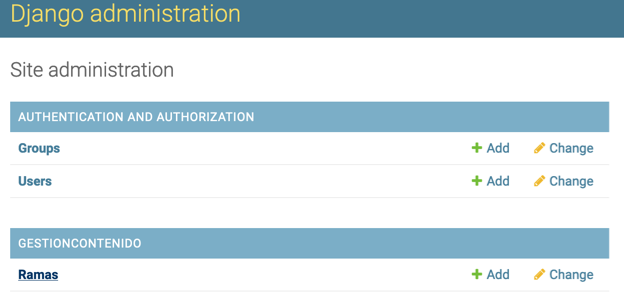
Para incluir las ramas en el panel de administración, nos vamos al fichero admin.py y escribimos lo siguiente:

from django.contrib import admin

from gestionContenido.models import Rama

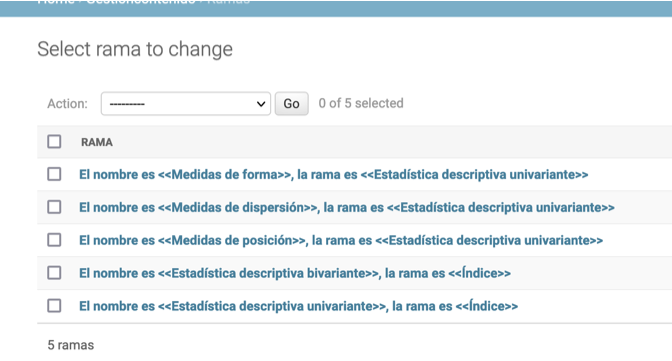
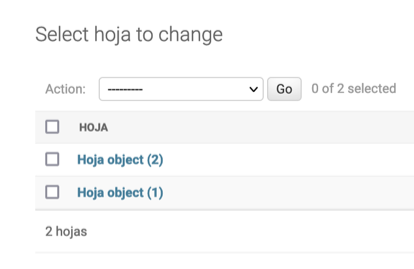
admin.site.register(Rama)

Nota: en el IDE PyCharm CE me daba error because yes. Ni caso, funciona perfectamente:



El uso del panel de administración es completamente intuitivo: crear, cambiar, eliminar, etc.

El panel de administración usa el método \_\_str\_\_ para mostrar la información del registro. Si ese método no está implementado, será ilegible, como pasa con Hoja:

Si queremos que alguno de los campos sea opcional, debemos irnos al modelo en Python y, dentro del campo en cuestión, añadir blank=True, null=True.

class Rama(models.Model):

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

rama = models.CharField(max\_length=100, blank=True, null=True)

def \_\_str\_\_(self):

return 'El nombre es <<%s>>, la rama es <<%s>>' % (self.nombre, self.rama)

Migramos y ya no nos pedirá la rama como campo requerido (se verá sin negrita).

