


Tutorial

django *girls*

Tabla de contenido

1. [Introducción](#)
2. [¿Cómo funciona Internet?](#)
3. [Introducción a la línea de comandos](#)
4. [Instalación de Python](#)
5. [Editor de código](#)
6. [Introducción a Python](#)
7. [¿Qué es Django?](#)
8. [Instalación de Django](#)
9. [Comenzar un proyecto en Django](#)
10. [Modelos en Django](#)
11. [Administrador de Django](#)
12. [¡Desplegar!](#)
13. [Django urls](#)
14. [Vistas de Django - ¡Es hora de crear!](#)
15. [Introducción a HTML](#)
16. [ORM de Django \(Querysets\)](#)
17. [Datos dinámicos en plantillas](#)
18. [Plantillas de Django](#)
19. [CSS - Hazlo bonito](#)
20. [Extender plantillas](#)
21. [Amplía tu aplicación](#)
22. [Formularios en Django](#)
23. [Dominio](#)
24. [¿Qué sigue?](#)

Tutorial de Django Girls

 GITTER 

Este trabajo está bajo la licencia internacional Creative Commons Attribution-ShareAlike 4.0. Para ver una copia de esta licencia, visita el siguiente enlace <http://creativecommons.org/licenses/by-sa/4.0/>

Translation

This tutorial has been translated from English into Spanish by a wonderful group of volunteers. Special thanks goes to Victoria Martinez de la Cruz, Kevin Morales, Joshua Aranda, Silvia Frias, Leticia, Andrea Gonzalez, Adrian Manjarres, Rodrigo Caicedo, Maria Chavez, Marcelo Nicolas Manso, Rosa Durante, Moises, Israel Martinez Vargas, JuanCarlos_, N0890Dy, Ivan Yivoff, Khaterine Castellano, Erick Navarro, cyncyncyn, ZeroSoul13, Erick Aguayo, Ernesto Rico-Schmidt, Miguel Lozano, osueboy, dynarro and Geraldina Garcia Alvarez.

Introducción

¿Alguna vez has sentido que el mundo está cada vez más cercano a la tecnología y de cierto modo te has quedado atrás? ¿Alguna vez te has preguntado cómo crear un sitio web pero nunca has tenido la suficiente motivación para empezar? ¿Has pensado alguna vez que el mundo del software es demasiado complicado para ti como para intentar hacer algo por tu cuenta?

Bueno, ¡tenemos buenas noticias para ti! Programar no es tan difícil como aparenta y queremos mostrarte cuán divertido puede llegar a ser.

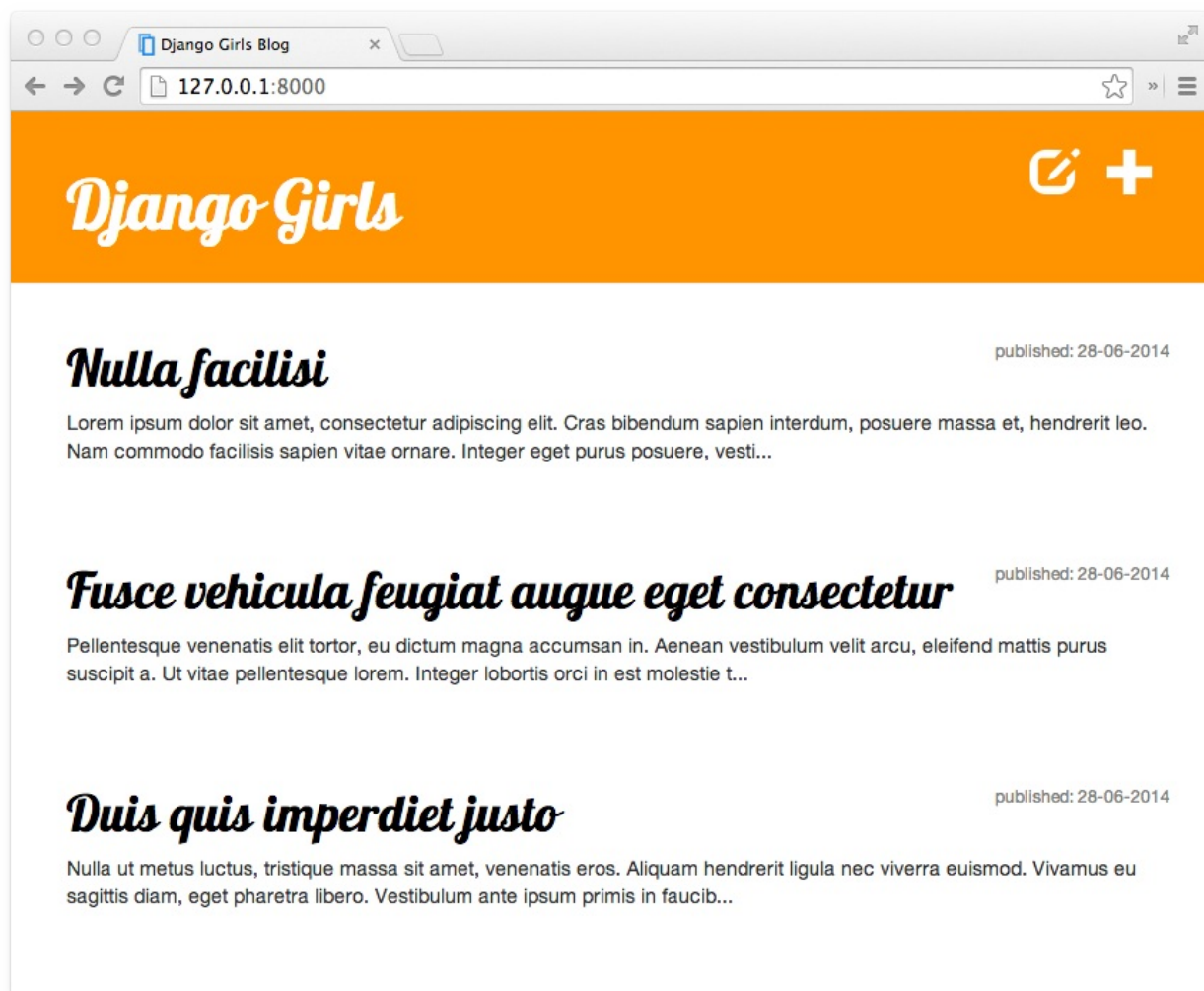
Este tutorial no te convertirá en programador mágicamente. Si quieres ser buena en esto, necesitarás meses o incluso años de aprendizaje y práctica. Pero queremos mostrarte que programar o crear sitios web no es tan complicado como parece. Intentaremos explicar pequeñas partes lo mejor que podamos, de forma que no te sientas intimidada por la tecnología.

¡Esperamos poder hacerte amar la tecnología tanto como nosotras lo hacemos!

¿Qué aprenderás con este tutorial?

Cuando termines el tutorial, tendrás una aplicación web simple y funcional: tu propio blog. Te mostraremos como publicarla online, ¡así otros podrán ver tu trabajo!

Tendrá (más o menos) ésta apariencia:



Si estás siguiendo este tutorial por tu cuenta y no tienes a nadie que te ayude en caso de surgir algún problema, tenemos un chat para ti: [GITTER](#) [JOIN CHAT →](#). ¡Hemos pedido a nuestros tutores y participantes anteriores que estén ahí de vez en cuando para ayudar a otros con el tutorial! ¡No temas dejar tus preguntas allí!

Bien, [empecemos por el principio...](#)

Sobre nosotros y cómo contribuir

Este tutorial lo mantiene [DjangoGirls](#). Si encuentras algún error o quieres actualizar el tutorial, por favor [sigue la guía de cómo contribuir](#).

¿Te gustaría ayudarnos a traducir el tutorial a

otros idiomas?

Actualmente, las traducciones se llevan a cabo sobre la plataforma crowdin.com en:

<https://crowdin.com/project/django-girls-tutorial>

Si tu idioma no esta listado en crowdin, por favor [abre un nuevo problema](#) informando el idioma así podemos agregarlo.

¿Cómo funciona Internet?

Este capítulo está inspirado por la charla "How the Internet works" de Jessica McKellar (<http://web.mit.edu/jesstess/www/>).

Apostamos que utilizas Internet todos los días. Pero, ¿sabes lo que pasa cuando escribes una dirección como <http://djangogirls.org> en tu navegador y presionas 'Enter'?

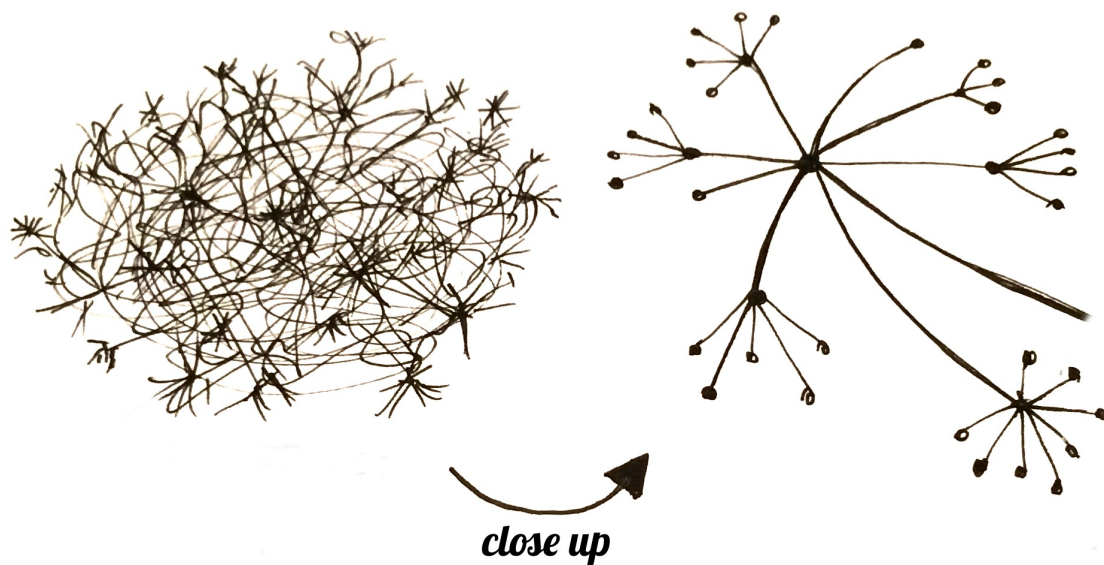
Lo primero que tienes que entender es que un sitio web es sólo un montón de archivos guardados en un disco duro. Al igual que tus películas, música o fotos. Sin embargo, los sitios web poseen una peculiaridad: ellos incluyen un código de computadoras llamado HTML.

Si no estás familiarizada con la programación, puede ser difícil de captar HTML al principio, pero tus navegadores web (como Chrome, Safari, Firefox, etc.) lo aman. Los navegadores web están diseñados para entender este código, seguir sus instrucciones y mostrar todos esos archivos de los cuales tu sitio web está hecho de la manera exacta como tu quieres que se muestren.

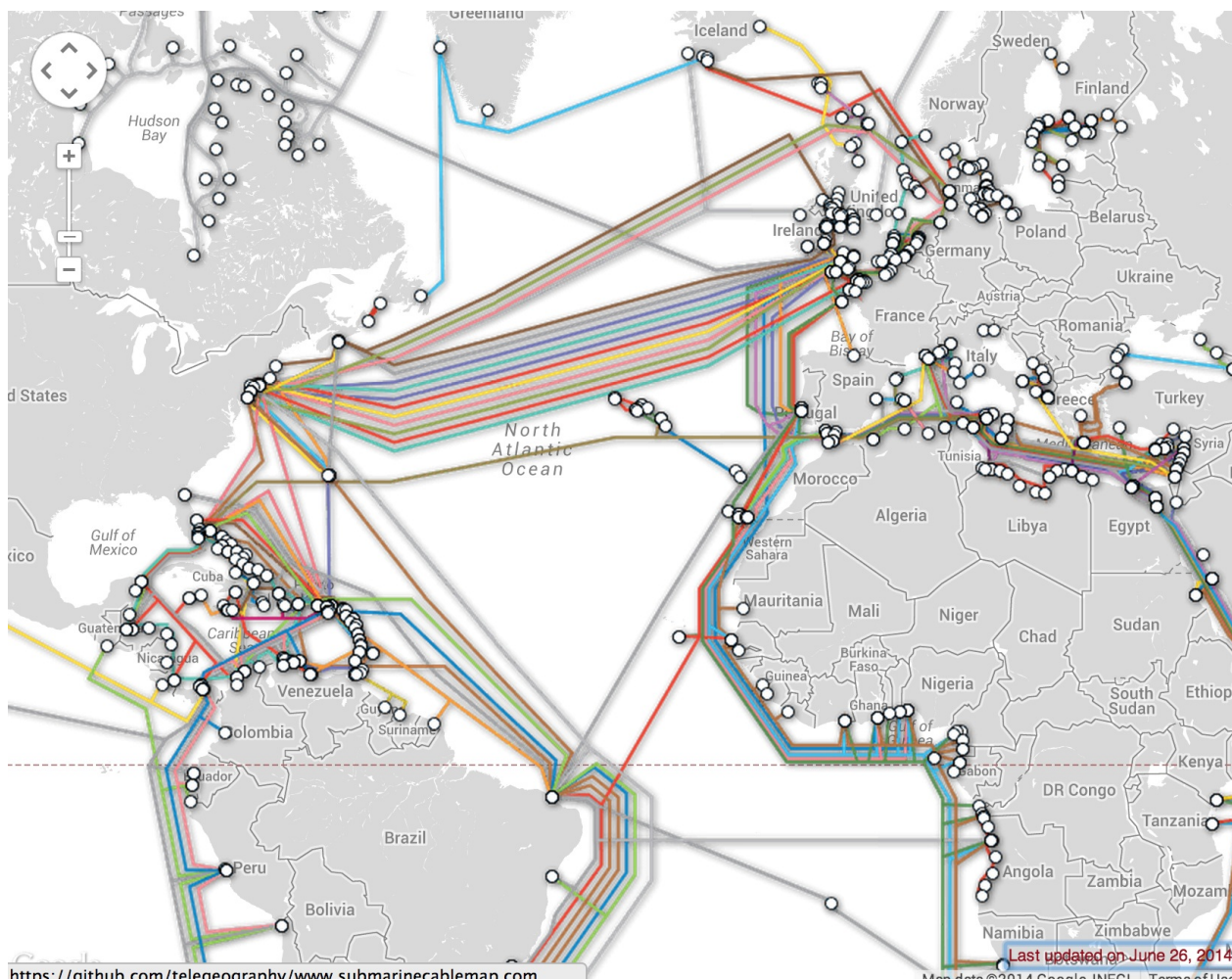
Como cualquier otro archivo, tenemos que guardar los archivos HTML en algún lugar de un disco duro. Para Internet, usamos unas computadoras especiales y poderosas llamadas *servidores*. Ellas no tienen una pantalla, mouse o teclado, debido a que su propósito es almacenar datos y servirlos. Por esa razón son llamados *servidores* -- porque ellos *sirven* los datos.

Ok, quizás te preguntes cómo luce Internet, ¿cierto?

¡Te hemos hecho una imagen! Luce algo así:

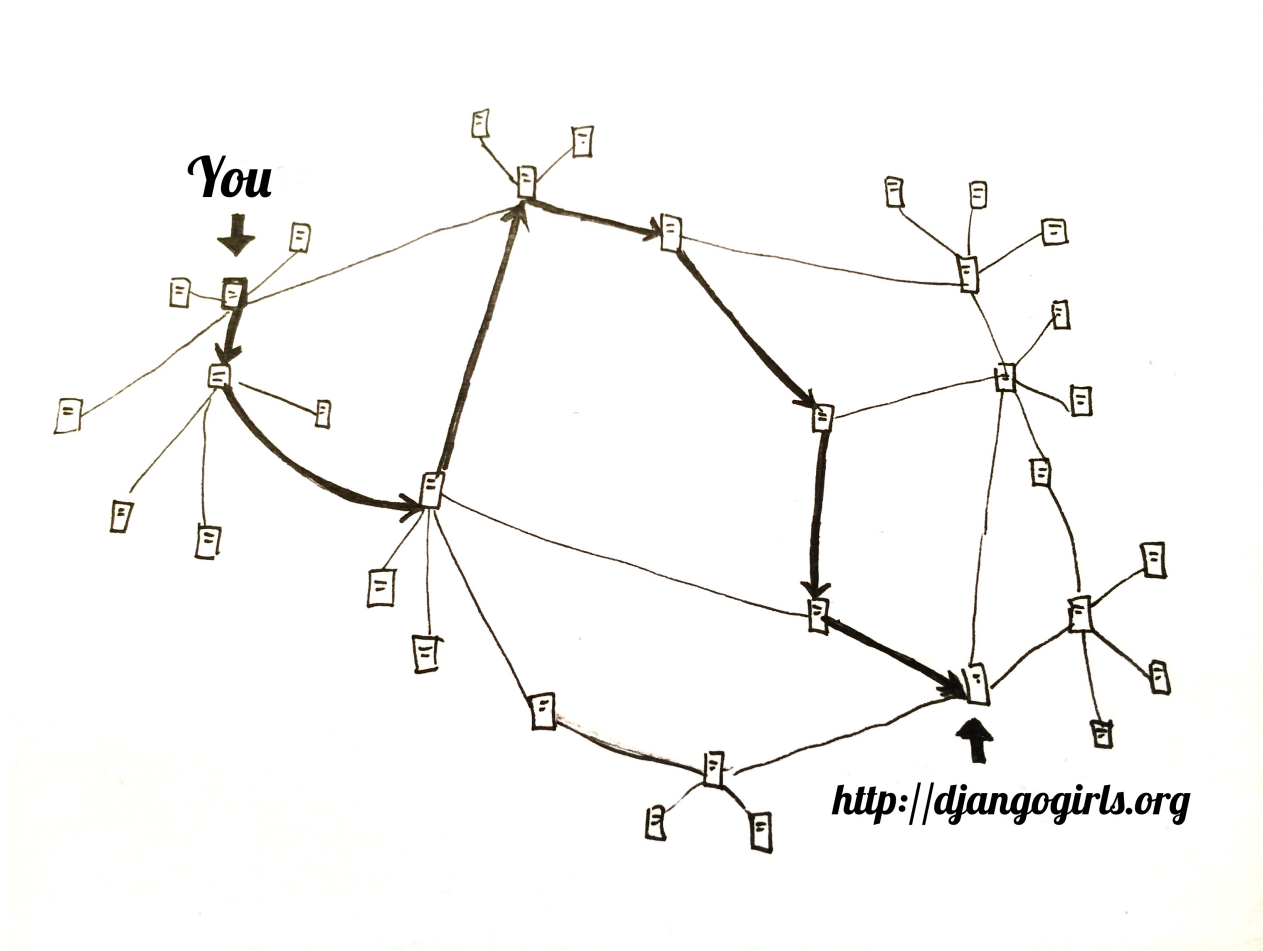


Parece un lío, ¿no? En realidad es una red de máquinas conectadas (los mencionados *servidores*). ¡Cientos de miles de máquinas! ¡Muchos, muchos kilómetros de cables alrededor del mundo! Puedes visitar el sitio web [Submarine Cable Map](http://submarinecablemap.com/) (<http://submarinecablemap.com/>) donde se muestran las conexiones de cables submarinos alrededor del mundo y ver lo complicada que es la red. Aquí hay una captura de pantalla de la página web:



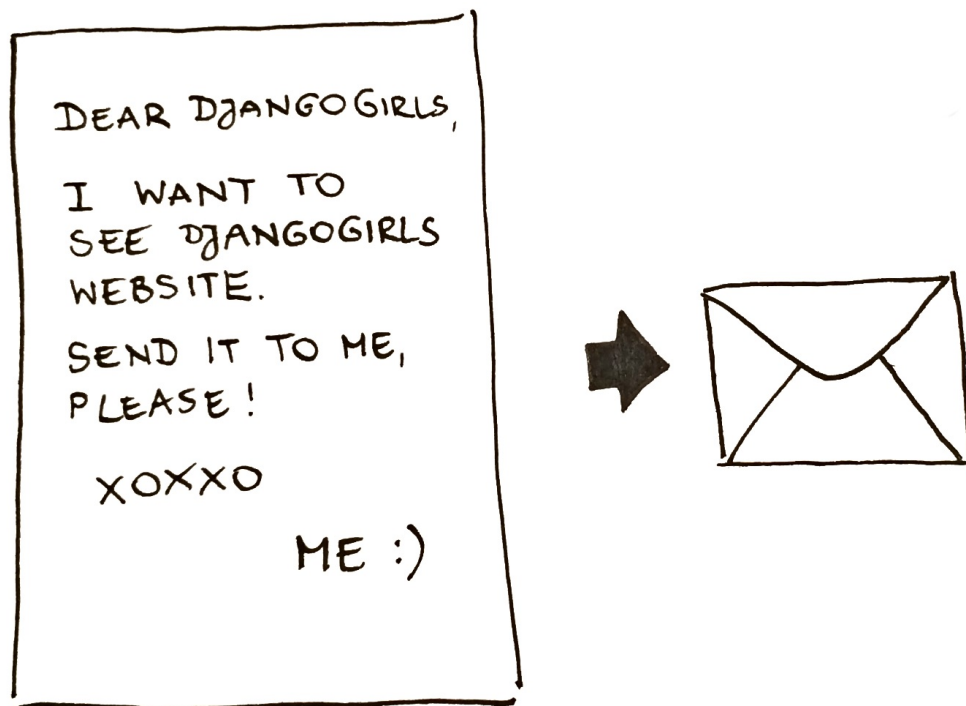
Es fascinante, ¿no? Pero, obviamente, no es posible tener un cable entre cada máquina conectada a Internet. Así que, para llegar a una máquina (por ejemplo la que aloja a <http://djangogirls.org>) tenemos que pasar una solicitud a través de muchas máquinas diferentes.

Se parece a esto:



Imagina que cuando escribes <http://djangogirls.org>, estas enviando una carta que dice: "Queridos Django Girls, me gustaría ver su sitio web djangogrils.org. Por favor, envíenmelo!"

Tu carta va hacia la oficina de correo más cercana. Luego va a otra un poco más cercana de su destinatario, luego a otra y a otra hasta que es entregada en su destino. Lo única cosa diferente es que si tu envías cartas (*paquetes de datos*) con frecuencia al mismo lugar, cada carta puede pasar por oficinas de correos (*routers*) totalmente diferentes, dependiendo de cómo se distribuyen en cada oficina.



Sí, es tan simple como eso. Enviar mensajes y esperar alguna respuesta. Por supuesto, en vez de papel y lapicera usas bytes de datos, ¡pero la idea es la misma!

En lugar de direcciones con el nombre de la calle, ciudad, código postal y nombre del país, utilizamos direcciones IP. Tu computadora pide primero el DNS (Domain Name System - en español Sistema de Nombres de Dominio) para traducir `django girls.org` a una dirección IP. Funciona como los viejos directorios telefónicos donde puedes buscar el nombre de la persona que se deseas contactar y este nos muestra su número de teléfono y dirección.

Cuando envías una carta, ésta necesita tener ciertas características para ser entregada correctamente: una dirección, sello, etc. También utilizas un lenguaje que el receptor pueda entender, ¿cierto? Lo mismo sucede con los *paquetes de datos* que envías para ver un sitio web: utilizas un protocolo llamado HTTP (Hypertext Transfer Protocol - en español Protocolo de Transferencia de Hipertexto).

Así que, básicamente, cuando tienes un sitio web necesitas tener un *servidor* (la máquina) donde vive. El *servidor* está esperando cualquier *solicitud* entrante (cartas que piden al servidor que envíe tu sitio web) y éste responde enviando tu sitio web (en otra carta).

Puesto que este es un tutorial de Django, seguro te preguntarás qué es lo que hace Django. Bueno, cuando envías una respuesta, no siempre quieres enviar lo mismo a todo el mundo. Es mucho mejor si tus cartas son personalizadas, especialmente para la persona que acaba de escribir, ¿cierto? Django nos ayuda con la creación de estas cartas personalizadas :).

Basta de charlas, ¡pongamos manos a la obra!

Introducción a la interfaz de línea de comandos

Es emocionante, ¿verdad? Vas a escribir tu primera línea de código en pocos minutos :)

Permítenos presentarte a tu primer nuevo amigo: ¡la línea de comandos!

Los siguientes pasos te mostrarán cómo usar aquella ventana negra que todos los hackers usan. Puede parecer un poco aterrador al principio pero es solo un mensaje en pantalla que espera a que le des órdenes.

¿Qué es la línea de comandos?

La ventana, que generalmente es llamada **línea de comandos** o **interfaz de línea de comandos**, es una aplicación basada en texto para la ver, manejar y manipular archivos en tu computadora (como por ejemplo el Explorador de Windows o Finder en Mac, pero sin la interfaz gráfica). Otros nombres para la línea de comandos son: *cmd*, *CLI*, *símbolo del sistema*, *consola* o *terminal*.

Abrir la interfaz de línea de comandos

Lo primero que debemos hacer para empezar a experimentar con nuestra interfaz de línea de comandos es abrirla.

Windows

Ir al menú Inicio → Todos los programas → Accesorios → Command Prompt

Mac OS X

Aplicaciones → Servicios → Terminal

Linux

Está probablemente en Aplicaciones → Accesorios → Terminal, pero eso depende de tu distribución. Si no lo encuentras, Googlealo :)

Prompt

Ahora deberías ver una ventana blanca o negra que está esperando tus órdenes.

Si estás en Mac o Linux, probablemente verás `$`, así:

```
$
```

En Windows, es un signo así `>`, como este:

```
>
```

Cada comando será precedido por este signo y un espacio, pero no tienes que escribirlo. Tu computadora lo hará por ti :)

Sólo una pequeña nota: en tu caso, tal vez hay algo como `C:\Users\ola>` o `olas-MacBook-Air:~ ola$` antes del prompt y eso es 100% correcto. En este tutorial lo simplificaremos lo más posible.

Tu primer comando (¡YAY!)

Vamos a empezar con algo simple. Escribe este comando:

```
$ whoami
```

O

```
> whoami
```

Y luego oprime la tecla Enter. Este es el resultado:

```
$ whoami olasitarska
```

Como puedes ver, la computadora sólo te presentó tu nombre de usuario. Bien, ¿eh? :)

Trata de escribir cada comando, no copies y pegues. ¡Te acordarás más de esta manera!

Básicos

Cada sistema operativo tiene un conjunto diferente de comandos para la línea de comandos, así que asegúrate de seguir las instrucciones para tu sistema operativo. Vamos a intentarlo, ¿de acuerdo?

Directorio actual

Sería bueno saber dónde estamos ahora, ¿cierto? Vamos a ver. Escribe este comando y oprime Enter:

```
$ pwd
/Users/olasitarska
```

Si estás en Windows:

```
> cd
C:\Users\olasitarska
```

Probablemente verás algo similar en tu máquina. Una vez que abres la línea de comandos generalmente empiezas en el directorio home de tu usuario.

Nota: 'pwd' significa 'print working directory' - en español, 'mostrar directorio de trabajo'.

Lista de archivos y directorios

¿Qué hay aquí? Sería bueno saber. Veamos:

```
$ ls
Applications
Desktop
Downloads
Music
...
```

Windows:

```
> dir
Directory of C:\Users\olasitarska
05/08/2014 07:28 PM <DIR> Applications
05/08/2014 07:28 PM <DIR> Desktop
05/08/2014 07:28 PM <DIR> Downloads
05/08/2014 07:28 PM <DIR> Music
```

```
...
```

Cambia el directorio actual

¿Quizás podemos ir a nuestro escritorio?

```
$ cd Desktop
```

Windows:

```
> cd Desktop
```

Comprueba si realmente ha cambiado:

```
$ pwd
/Users/olasitarska/Desktop
```

Windows:

```
> cd
C:\Users\olasitarska\Desktop
```

¡Aquí está!

Pro tip: si escribes `cd D` y luego oprimas `tab` en el teclado, la línea de comandos automáticamente completará el resto del nombre para que puedas navegar más rápido. Si hay más de una carpeta que empiece con "D", presiona el botón `tab` dos veces para obtener una lista de opciones.

Crear directorio

¿Qué tal si creamos un directorio de Django Girls en tu escritorio? Puedes hacerlo de esta manera:

```
$ mkdir.djangogirls
```

Windows:

```
> mkdir.djangogirls
```

Este pequeño comando creará una carpeta con el nombre `djangogirls` en tu escritorio. ¡Puedes comprobar si está allí buscando en tu escritorio o ejecutando el comando `ls/dir` ! Inténtalo :)

Pro tip: Si no quieres escribir una y otra vez los mismos comandos, prueba oprimiendo la `flecha arriba` y `flecha abajo` de tu teclado para ver recientes comandos utilizados.

¡Ejercicios!

Un pequeño reto para ti: en el directorio recién creado `djangogirls` crea un directorio llamado `test` . Utiliza los comandos `cd` y `mkdir` .

Solución:

```
$ cd.djangogirls
$ mkdir test
$ ls
```

Windows:

```
> cd.djangogirls
> mkdir test
> dir
08/05/2014 19:28 < DIR > test
```

¡Felicitaciones! :)

Limpiar

No queremos dejar un desorden, así que vamos a eliminar todo lo que hicimos hasta este momento.

En primer lugar, tenemos que volver al escritorio:

```
$ cd ..
```

Windows:


```
> cd ..
```

`cd ..` cambiará el directorio actual al directorio padre (que significa el directorio que contiene el directorio actual).

Revisa dónde estás:

```
$ pwd
/Users/olasitarska/Desktop
```

Windows:

```
> cd
C:\Users\olasitarska\Desktop
```

Ahora es hora de eliminar el directorio `djangogirls`.

Atención: Eliminar archivos utilizando `del`, `rmdir` o `rm` hace que no puedan recuperarse, lo que significa que los *archivos borrados desaparecerán para siempre*. Debes ser muy cuidadosa con este comando.

```
$ rm -r djangogirls
```

Windows:

```
> rmdir/s djangogirls
djangogirls, ¿Estás seguro <Y/N>? Y
```

Hecho! Asegurémonos que en verdad fueron borrados, vamos a ver:

```
$ ls
```

Windows:

```
> dir
```

Salida

¡Esto es todo por ahora! Ahora puedes cerrar la línea de comandos sin problemas. Vamos a hacerlo al estilo hacker, ¿bien? :)

```
$ exit
```

Windows:

```
> exit
```

Genial, ¿no? :)

Índice

Aquí hay una lista de algunos comandos útiles:

Comando (Windows)	Comando (Mac OS / Linux)	Descripción	Ejemplo
exit	exit	Cierra la ventana	exit
cd	cd	Cambia el directorio	cd test
dir	ls	Lista directorios/archivos	dir
copy	cp	Copia de archivos	copy c:\test\test.txt c:\windows\test.txt
move	mv	Mueve archivos	move c:\test\test.txt c:\windows\test.txt
mkdir	mkdir	Crea un nuevo directorio	mkdir testdirectory
del	rm	Elimina archivos/directorios	del c:\test\test.txt

Estos son solo algunos de los comandos que puedes ejecutar en la línea de comandos. No vas a usar nada más que esos por ahora.

Si tienes curiosidad, ss64.com contiene una referencia completa de comandos para todos los sistemas operativos.

¿Lista?

¡Vamos a sumergirnos en Python!

Vamos a empezar con Python

¡Por fin estamos aquí!

Pero primero, déjenos decirte qué es Python. Python es un lenguaje de programación muy popular que puede utilizarse para la creación de sitios web, juegos, software académico, gráficos y mucho, mucho más.

Python se originó en la década de 1980 y su objetivo principal es ser legible por los seres humanos (¡no sólo para las máquinas!), por eso parece mucho más simple que otros lenguajes de programación. Esto hace que sea más fácil de aprender, pero no te preocupes, ¡Python es también muy poderoso!

Instalación de Python

Este subcapítulo se basa en un tutorial de Geek Girls Carrots (<http://django.carrots.pl/>)

Django está escrito en Python. Necesitamos Python para cualquier cosa en Django. ¡Vamos a empezar con la instalación! Queremos que instales Python 3.4, así que si tienes alguna versión anterior, deberás actualizarla.

Windows

Puedes descargar Python para Windows desde el sitio web <https://www.python.org/downloads/release/python-343/>. Después de descargar el archivo ***.msi**, debes ejecutarlo (haz doble click en el archivo) y sigue las instrucciones. Es importante recordar la ruta (el directorio) donde se ha instalado Python. ¡Será necesario más adelante!

Algo para tener en cuenta: en la segunda pantalla del asistente de instalación, llamada "Customize", asegúrate de ir hacia abajo y elegir la opción "Add python.exe to the Path", como en



Linux

Es muy posible que ya tengas Python instalado. Para verificar que ya lo tienes instalado (y qué versión es), abre una consola y tipea el siguiente comando:

```
$ python3 --version
Python 3.4.2
```

Si no tienes Python instalado o si quieres una versión diferente, puedes instalarlo como sigue:

Ubuntu

Tipea este comando en tu consola:

```
sudo apt-get install python3.4
```

Fedora

Usa este comando en tu consola:

```
sudo yum install python3.4
```

OS X

Debes ir al sitio web <https://www.python.org/downloads/release/python-342/> y descargar el instalador de Python:

- descarga el archivo *DMG Mac OS X 64-bit/32-bit installer*,
- haz doble click para abrirlo,
- doble click en *Python.mpkg* para ejecutar al instalador.

Verifica que la instalación fue exitosa abriendo la *Terminal* y ejecutando el comando

`python3 :`

```
$ python3 --version
Python 3.4.2
```

Si tienes alguna duda o si algo salió mal y no sabes cómo resolverlo - ¡pide ayuda a tu tutor! Algunas veces las cosas no salen tan fácilmente y es mejor pedir ayuda a alguien con más experiencia.

Editor de código

Estás a punto de escribir tu primera línea de código, así que ¡es hora de descargar un editor de código!

Hay muchos editores diferentes, cuál usar depende mucho de la preferencia personal. La mayoría de programadores de Python usan IDEs (Entornos de Desarrollo Integrados) complejos pero muy poderosos, como PyCharm. Sin embargo, como principiante, eso es probablemente menos conveniente; nuestras recomendaciones son igual de poderosas pero mucho mas simples.

Nuestras sugerencias están listadas abajo, pero siéntete libre de preguntarle a tu tutor cuáles son sus preferencias - así será más fácil obtener su ayuda.

Gedit

Gedit es un editor de código abierto, gratis, disponible para todos los sistemas operativos.

[Descárgalo aquí](#)

Sublime Text 2

Sublime Text es un editor muy popular con un periodo de prueba gratis. Es fácil de instalar y está disponible para todos los sistemas operativos.

[Descárgalo aquí](#)

Atom

Atom es un editor de código muy nuevo creado por [GitHub](#). Es gratis, de código abierto, fácil de instalar y fácil de usar. Está disponible para Windows, OSX y Linux.

[Descárgalo aquí](#)

¿Por qué estamos instalando un editor de código?

Puedes estar preguntándote por qué estamos instalando un editor especial, en lugar de usar un editor convencional como Word o Notepad.

En primer lugar, el código tiene que ser **texto plano** y el problema de las aplicaciones como Word o Textedit es que en realidad no producen texto plano. Lo que generan es texto enriquecido (con tipografías y formato), usando formatos propios como rtf.

La segunda razón es que los editores de código son herramientas especializadas y, como tales, tienen características muy útiles, como resaltar la sintáxis del código con diferentes colores de acuerdo a su significado o cerrar comillas por ti automáticamente.

Veremos todo esto en acción más adelante. En breve empezarás a pensar en tu fiel editor de código como una de tus herramientas favoritas :)

Introducción a Python

Parte de este capítulo se basa en tutoriales por Geek Girls Carrots (<http://django.carrots.pl/>).

¡Vamos a escribir algo de código!

Python prompt

Para empezar a jugar con Python, tenemos que abrir una *línea de comandos* en nuestra computadora. Ya sabes cómo hacerlo, lo aprendiste en el capítulo de [Introducción a la línea de comandos](#).

Una vez que estés listo, sigue las siguientes instrucciones.

Queremos abrir una consola de Python, así que escribe `python3` y pulsa Enter.

```
$ python3
Python 3.4.2 (...)
Type "copyright", "credits" or "license" for more information.
>>>
```

Tu primer comando en Python!

Después de ejecutar el comando de Python, el cursor cambia a `>>>`. Para nosotros esto significa que por ahora sólo podemos utilizar comandos en el lenguaje Python. No tienes que escribir el `>>>` - Python lo hará por ti.

Si deseas salir de la consola de Python en cualquier momento, simplemente escribe `exit()` o usa el atajo `ctrl + z` para Windows y `ctrl + d` para Mac/Linux. Luego no verás más `>>>`.

Pero ahora no queremos salir de la consola de Python. Queremos aprender más sobre ella. Vamos a empezar con algo muy simple. Por ejemplo, trata de escribir algo de matemáticas, como `2 + 3` y pulsa Enter.

```
>>> 2 + 3
5
```


¡Bien! ¿Ves como salió la respuesta? ¡Python sabe matemáticas! Podrías intentar otros comandos como: - `4 * 5` - `5 - 1` - `40 / 2`

Diviértete con esto por un momento y luego vuelve aquí :).

Como puedes ver, Python es una gran calculadora. Si te estás preguntando qué más puede hacer...

Strings

¿Y tu nombre? Escribe tu nombre de pila en frases como ésta:

```
>>> "Ola"  
'Ola'
```

¡Has creado tu primer string! Es una secuencia de caracteres que puede ser procesada por una computadora. El string (o en español, cadena) debe comenzar y terminar con el mismo carácter. Esto puede ser comillas simples (`' '`) o dobles (`" "`) - ellas le dicen a Python que lo que esta dentro es una cadena.

Las cadenas pueden ser concatenadas. Prueba esto:

```
>>> "Hola " + "Ola"  
'Hola Ola'
```

También puedes multiplicar las cadenas con un número:

```
>>> "Ola" * 3  
'OlaOlaOla'
```

Si necesitas poner un apóstrofe dentro de tu cadena, tienes dos maneras de hacerlo.

Usando comillas dobles:

```
>>> "Runnin' down the hill"  
"Runnin' down the hill"
```

o escapando el apóstrofe con una barra invertida (```):

```
>>> 'Runnin\' down the hill'  
"Runnin' down the hill"
```

Bien, ¿eh? Para ver tu nombre en letras mayúsculas, simplemente escribe:

```
>>> "ola".upper()  
'OLA'
```

¡Usaste la **función** `upper` en tu cadena! Una función (como `upper()`) es un conjunto de instrucciones que Python tiene que realizar sobre un objeto determinado (`"ola"`) una vez que se llama.

Si quisieras saber el número de letras que contiene tu nombre, también existe una función para esto.

```
>>> len("ola")  
3
```

Te preguntarás por qué a veces se llama a las funciones con un `.` al final de una cadena (como `"ola".upper()`) y a veces se llama a una función y colocas la cadena entre paréntesis. Bueno, en algunos casos las funciones pertenecen a objetos, como `upper()`, que sólo puede ser utilizado sobre cadenas (`upper()` es una función de los objetos `string`). En este caso, llamamos **método** a esta función. Otra veces, las funciones no pertenecen a ningún objeto específico y pueden ser usados en diferentes objetos, como `len()`. Esta es la razón de por qué estamos pasando `"ola"` como un parámetro a la función `len`.

Resumen

Ok, suficiente sobre las cadenas. Hasta ahora has aprendido sobre:

- **la terminal** - teclear comandos (código) dentro de la terminal de Python resulta en respuestas de Python
- **números y strings** - en Python los números son usados para matemáticas y strings para objetos de texto
- **operadores** - como `+` y `*`, combina valores para producir uno nuevo
- **funciones** - como `upper()` y `len()`, realizan opciones sobre los objetos.

Estos son los conocimientos básicos que puedes aprender de cualquier lenguaje de programación. ¿Lista para algo un poco más difícil? ¡Apostamos que lo estás!

Errores

Intentemos con algo nuevo. ¿Podríamos obtener la longitud de un número de la misma manera que obtuvimos la longitud de nuestro nombre? Teclea `len(304023)` y presiona

Enter:

```
>>> len(304023)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
```

¡Obtuvimos nuestro primer error! Dice que los objetos de tipo "int" (números enteros) no tienen ninguna longitud. ¿Qué podemos hacer ahora? Quizás podemos escribir el número como un string. Los strings tienen longitud, ¿cierto?

```
>>> len(str(304023))
6
```

¡Funcionó! Utilizamos la función `str` dentro de la función `len`. `str()` convierte todo a strings.

- La función `str` convierte cosas en **strings**
- La función `int` convierte cosas en **integers**

Importante: podemos convertir números en texto, pero no podemos necesariamente convertir texto en números - ¿qué sería `int('hello')` ?

Variables

Un concepto importante en programación son las variables. Una variable no es más que un nombre para alguna cosa para que puedas usarla más tarde. Los programadores usan estas variables para almacenar datos, hacer su código más legible y así no tener que seguir recordando que hace cada cosa.

Supongamos que queremos crear una nueva variable llamada `name` :

```
>>> name = "Ola"
```

¿Ves? ¡Es fácil! Es simplemente: `name` equivale a `Ola`.

Como te has dado cuenta, el programa no regresa algo como lo hacía antes. Entonces, ¿Cómo sabemos que la variable existe realmente? Simplemente introduce `name` y pulsa Enter:

```
>>> name
'Ola'
```

¡Súper! Tu primer variable :). Siempre podrás cambiar a lo que se refiere:

```
>>> name = "Sonja"
>>> name
'Sonja'
```

Puedes usarla dentro de funciones también:

```
>>> len(name)
5
```

Increíble, ¿verdad? Por supuesto, las variables pueden ser cualquier cosa, ¡también números! Prueba esto:

```
>>> a = 4
>>> b = 6
>>> a * b
24
```

Pero ¿qué pasa si usamos el nombre equivocado? ¿Puedes adivinar qué pasaría? ¡Vamos a probar!

```
>>> city = "Tokyo"
>>> ctiy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ctiy' is not defined
```

¡Un error! Como puedes ver, Python tiene diferentes tipos de errores y este se llama **NameError**. Python te dará este error si intentas utilizar una variable que no ha sido definida aún. Si más adelante te encuentras con este error, verifica tu código para ver si no has escrito mal una variable.

Juega con esto por un rato y mira que puedes hacer!

La función print

Intenta esto:

```
>>> name = 'Maria'
>>> name
'Maria'
```

```
>>> print(name)
Maria
```

Cuando sólo escribes `name`, el intérprete de Python responde con la *representación* del string de la variable 'name', que son las letras M-a-r-i-a, rodeadas de comillas simples ". Cuando dices `print(name)`, Python va a "imprimir" el contenido de la variable a la pantalla, sin las comillas, que es mejor.

Como veremos después, `print()` también es útil cuando queremos imprimir cosas desde adentro de las funciones, o bien cuando queremos imprimir cosas en múltiples líneas.

Listas

Además de string e integers, Python tiene toda clase de diferentes tipos de objetos. Ahora vamos a introducir uno llamado **list**. Las listas son exactamente lo que piensas que son: son objetos que son listas de otros objetos :)

Anímate y crea una lista:

```
>>> []
[]
```

Sí, esta lista está vacía. No es muy útil, ¿verdad? Vamos a crear una lista de números de lotería. No queremos repetir todo el tiempo, así que los pondremos en una variable también:

```
>>> lottery = [3, 42, 12, 19, 30, 59]
```

Muy bien, ¡tenemos una lista! ¿Qué podemos hacer con ella? Vamos a ver cuántos números de lotería hay en la lista. ¿Tienes alguna idea de qué función deberías usar para eso? ¡Ya sabes esto!

```
>>> len(lottery)
6
```

¡Sí! `len()` puede darte el número de objetos en una lista. Útil, ¿verdad? Tal vez la ordenemos ahora:

```
>>> lottery.sort()
```

Esto no devuelve nada, sólo cambió el orden en que los números aparecen en la lista.

Vamos a imprimir la lista otra vez y ver que pasó:

```
>>> print(lottery)
[3, 12, 19, 30, 42, 59]
```

Como puedes ver, los números en tu lista ahora están ordenados de menor a mayor.
¡Felicidades!

¿Te gustaría invertir ese orden? ¡Vamos a hacerlo!

```
>>> lottery.reverse()
>>> print(lottery)
[59, 42, 30, 19, 12, 3]
```

Fácil, ¿no? Si quieres añadir algo a tu lista, puedes hacerlo escribiendo este comando:

```
>>> lottery.append(199)
>>> print(lottery)
[59, 42, 30, 19, 12, 3, 199]
```

Si deseas mostrar sólo el primer número, puedes hacerlo mediante el uso de **indexes** (en español, índices). Un índice es el número que te dice dónde en una lista aparece un ítem. La computadora inicia la cuenta en 0, así que el primer objeto en tu lista está en el índice 0, el siguiente es 1, y así sucesivamente. Intenta esto:

```
>>> print(lottery[0])
59
>>> print(lottery[1])
42
```

Como puedes ver, puedes acceder a diferentes objetos en tu lista utilizando el nombre de la lista y el índice del objeto dentro de corchetes.

Para diversión adicional, prueba algunos otros índices: 6, 7, 1000, -1, -6 ó -1000. A ver si se pueden predecir el resultado antes de intentar el comando. ¿Tienen sentido los resultados?

Puedes encontrar una lista de todos los métodos disponibles para listas en este capítulo de la documentación de Python: <https://docs.python.org/3/tutorial/datastructures.html>

Diccionarios

Un diccionario es similar a una lista, pero accedes a valores usando una clave en vez de un

índice. Una clave puede ser cualquier cadena o número. La sintaxis para definir un diccionario vacío es:

```
>>> {}  
{}
```

Esto demuestra que acabas de crear un diccionario vacío. ¡Hurra!

Ahora, trata escribiendo el siguiente comando (intenta reemplazando con propia información):

```
>>> participant = {'name': 'Ola', 'country': 'Poland', 'favorite_numbers': [7, 42, 92]}
```

Con este comando, acabas de crear una variable `participant` con tres pares clave-valor:

- La clave `name` apunta al valor `'ola'` (un objeto `string`),
- `country` apunta a `'Poland'` (otro `string`),
- y `favorite_numbers` apunta a `[7, 42, 92]` (una `list` con tres números en ella).

Puedes verificar el contenido de claves individuales con esta sintaxis:

```
>>> print(participant['name'])  
ola
```

Lo ves, es similar a una lista. Pero no necesitas recordar el índice - sólo el nombre.

¿Qué pasa si le pedimos a Python el valor de una clave que no existe? ¿Puedes adivinar? ¡Pruébalo y verás!

```
>>> participant['age']  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'age'
```

¡Mira, otro error! Este es un **KeyError**. Python te ayuda y te dice que la llave `'age'` no existe en este diccionario.

¿Cuándo utilizar un diccionario o una lista? Bueno, eso es un buen punto para reflexionar. Sólo ten una solución en mente antes de mirar la respuesta en la siguiente línea.

- ¿Sólo necesitas una secuencia ordenada de elementos? Usa una lista.
- ¿Necesitas asociar valores con claves, así puedes buscarlos eficientemente (usando

las claves) más adelante? Utiliza un diccionario.

Los diccionarios, como las listas, son *mutables*, lo que significa que pueden ser cambiados después de ser creados. Puedes agregar nuevos pares clave/valor en el diccionario después de que ha sido creado, por ejemplo:

```
>>> participant['favorite_language'] = 'Python'
```

Como en las listas, el método `len()` en los diccionarios, devuelve el número de pares clave-valor en el diccionario. Adelante, escribe el comando:

```
>>> len(participant)
4
```

Espero tenga sentido hasta ahora. :) ¿Lista para más diversión con los diccionarios? Salta a la siguiente línea para algunas cosas sorprendentes.

Puedes utilizar el comando `del` para borrar un elemento en el diccionario. Por ejemplo, si deseas eliminar la entrada correspondiente a la clave `'favorite_numbers'`, sólo tienes que escribir el siguiente comando:

```
>>> del participant['favorite_numbers']
>>> participant
{'country': 'Poland', 'favorite_language': 'Python', 'name': 'Ola'}
```

Como puedes ver en la salida, el par de clave-valor correspondiente a la clave `'favorite_numbers'` ha sido eliminado.

Además de esto, también puedes cambiar un valor asociado a una clave ya creada en el diccionario. Teclea:

```
>>> participant['country'] = 'Germany'
>>> participant
{'country': 'Germany', 'favorite_language': 'Python', 'name': 'Ola'}
```

Como puedes ver, el valor de la clave `'country'` ha sido modificado de `'Poland'` a `'Germany'`. :) ¿Emocionante? ¡Hurra! Has aprendido otra cosa asombrosa.

Resumen

¡Genial! Sabes mucho sobre programación ahora. En esta última parte aprendiste sobre:

- **errors** - ahora sabes cómo leer y entender los errores que aparecen si Python no entiende un comando que le has dado
- **variables** - nombres para los objetos que te permiten codificar más fácilmente y hacer el código más legible
- **lists** - listas de objetos almacenados en un orden determinado
- **dictionaries** - objetos almacenados como pares clave-valor

¿Emocionada por la siguiente parte? :)

Compara cosas

Una gran parte de la programación incluye comparar cosas. ¿Qué es lo más fácil para comparar? Números, por supuesto. Vamos a ver cómo funciona:

```
>>> 5 > 2
True
>>> 3 < 1
False
>>> 5 > 2 * 2
True
>>> 1 == 1
True
>>> 5 != 2
True
```

Le dimos a Python algunos números para comparar. Como puedes ver, Python no sólo puede comparar números, sino que también puede comparar resultados de método. Bien, ¿eh?

¿Te preguntas por qué pusimos dos signos igual `==` al lado del otro para comparar si los números son iguales? Utilizamos un solo `=` para asignar valores a las variables. Siempre, **siempre** es necesario poner dos `==`. Si deseas comprobar que las cosas son iguales entre sí. También podemos afirmar que las cosas no son iguales a otras. Para eso, utilizamos el símbolo `!=`, como mostramos en el ejemplo anterior.

Da dos tareas más a Python:

```
>>> 6 >= 12 / 2
True
>>> 3 <= 2
False
```

`>` y `<` son fáciles, pero ¿qué es significa `>=` y `<=`? Se leen así:

- `x > y` significa: x es mayor que y
- `x < y` significa: x es menor que y
- `x <= y` significa: x es menor o igual que y
- `x >= y` significa: x es mayor o igual que y

¡Genial! ¿Quieres hacer uno mas? Intenta esto:

```
>>> 6 > 2 y 2 < 3
True
>>> 3 > 2 y 2 < 1
False
>>> 3 > 2 ó 2 < 1
True
```

Puedes darle a Python todos los números para comparar que quieras, y siempre te dará una respuesta. Muy inteligente, ¿verdad?

- **and** - si utilizas el operador `and`, ambas comparaciones deben ser True para que el resultado de todo el comando sea True
- **or** - si utilizas el operador `or`, sólo una de las comparaciones tiene que ser True para que el resultado de todo el comando sea True

¿Has oído la expresión "comparar manzanas con naranjas"? Vamos a probar el equivalente en Python:

```
>>> 1 > 'django'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: int() > str()
```

Aquí verás que al igual que en la expresión, Python no es capaz de comparar un número (`int`) y un string (`str`). En cambio, muestra un **TypeError** y nos dice que los dos tipos no se pueden comparados.

Boolean

Por cierto, acabas de aprender acerca de un nuevo tipo de objeto en Python. Se llama un **Boolean** -- y es probablemente el tipo más simple que existe.

Hay sólo dos objetos Boolean: - True - False

Pero para que Python entienda esto, es necesario que siempre lo escribas como True (primera letra mayúscula, con el resto de la letras minúsculas). **true**, **TRUE**, **tTRUE** no

funcionarán -- sólo True es correcto. (Lo mismo aplica a False también, por supuesto.)

Los valores booleanos pueden ser variables, también. Ve el siguiente ejemplo:

```
>>> a = True
>>> a
True
```

También puedes hacerlo de esta manera:

```
>>> a = 2 > 5
>>> a
False
```

Practica y diviértete con los booleanos ejecutando los siguientes comandos:

- `True and True`
- `False and True`
- `True or 1 == 1`
- `1 != 2`

¡Felicidades! Los booleanos son una de las funciones más geniales en programación y acabas de aprender cómo usarlos.

¡Guárdalo!

Hasta ahora hemos estado escribiendo nuestro código Python en el intérprete, lo cual nos limita a una línea de código a la vez. Normalmente los programas son guardados en archivos y son ejecutados por el **intérprete** o **compilador** de nuestro lenguaje de programación. Hasta ahora, hemos estado corriendo nuestros programas de a una línea por vez en el **intérprete** de Python. Necesitaremos más de una línea de código para las siguientes tareas, entonces necesitaremos hacer rápidamente lo que sigue:

- Salir del intérprete de Python
- Abrir el editor de texto de nuestra elección
- Guardar algo de código en un nuevo archivo de Python
- ¡Ejecutarlo!

Para salir del intérprete de Python que hemos estado usando, simplemente escribe la función `exit()`:

```
>>> exit()
```

```
$
```

Esto te llevará de vuelta a la línea de comandos.

Anteriormente, elegimos un editor de código en la sección de [Editor de código](#). Tendremos que abrir el editor ahora y escribir algo de código en un archivo nuevo:

```
print('Hello, Django girls!')
```

Nota Deberías notar una de las cosas más geniales de los editores de código: ¡los colores! En la consola de Python, todo era del mismo color, pero ahora puedes ver que la función `print` es de un color diferente del string que está adentro de ella. Eso se denomina "resaltado de sintaxis", y es una gran ayuda cuando estás programando. Presta atención a los colores, y obtendrás una pista cuando te olvides de cerrar un string o cometes un error al escribir una palabra clave (como el `def` en una función, que veremos abajo). Esta es una de las razones por las cuales usar un editor de código :)

Obviamente, ahora eres una desarrolladora Python muy experimentada, así que siéntete libre de escribir algo del código que has aprendido hoy.

Ahora tenemos que guardar el archivo y asignarle un nombre descriptivo. Vamos a llamar al archivo **python_intro.py** y guardarlo en tu escritorio. Podemos nombrar el archivo de cualquier manera que queramos, lo importante aquí es asegurarse que el archivo finalice con **.py**, esto le indica a nuestra computadora que este es un **archivo ejecutable de Python** y que Python puede correrlo.

Con el archivo guardado, ¡es hora de ejecutarlo! Utilizando las habilidades que has aprendido en la sección de línea de comandos, utiliza la terminal para **cambiar los directorios** e ir al escritorio.

En una Mac, el comando se verá algo como esto:

```
cd /Users/<your_name>/Desktop
```

En Linux, va a ser así (la palabra "Desktop" puede estar traducida a tu idioma):

```
cd /home/<your_name>/Desktop
```

Y en Windows, será así:

```
cd C:\Users\<your_name>\Desktop
```

Si te quedas atascada, sólo pide ayuda.

y luego usa Python para ejecutar el código en el archivo como sigue:

```
$ python3 python_intro.py
Hello, Django girls!
```

¡Muy bien! Ejecutaste tu primer programa de Python desde un archivo. ¿No se siente increíble?

Ahora puedes moverte a una herramienta esencial en la programación:

If...elif...else

Un montón de cosas en el código sólo son ejecutadas cuando se cumplen las condiciones dadas. Por eso Python tiene algo llamado **sentencias if**.

Reemplaza el código en tu archivo **python_intro.py** por esto:

```
if 3 > 2:
```

Si lo guardáramos y lo ejecutáramos, veríamos un error como este:

```
$ python3 python_intro.py
File "python_intro.py", line 2
    ^
SyntaxError: unexpected EOF while parsing
```

Python espera que le demos más instrucciones que se supone serán ejecutadas si la condición `3 > 2` resulta ser verdadera (o `True` en este caso). Intentemos hacer que Python imprima "It works!". Cambia tu código en el archivo **python_intro.py** para que se vea como esto:

```
if 3 > 2:
    print('It works!')
```

¿Observas cómo hemos indentado la siguiente línea de código con 4 espacios? Tenemos que hacer esto para que Python sepa qué código ejecutar si la comparación resulta verdadera. Puedes poner un espacio, pero casi todos los programadores Python hacen 4

espacios para hacer que el código sea más legible. Un solo tab también contará como 4 espacios.

Guárdalo y ejecútalo de nuevo:

```
$ python3 python_intro.py
It works!
```

¿Qué pasa si la condición no es verdadera?

En ejemplos anteriores, el código fue ejecutado sólo cuando las condiciones eran ciertas. Pero Python también tiene declaraciones `elif` y `else`:

```
if 5 > 2:
    print('5 is indeed greater than 2')
else:
    print('5 is not greater than 2')
```

Al ejecutar esto se imprimirá:

```
$ python3 python_intro.py
5 is indeed greater than 2
```

Si 2 fuera un número mayor que 5, entonces el segundo comando sería ejecutado. Fácil, ¿verdad? Vamos a ver cómo funciona `elif`:

```
name = 'Sonja'
if name == 'Ola':
    print('Hey Ola!')
elif name == 'Sonja':
    print('Hey Sonja!')
else:
    print('Hey anonymous!')
```

y al ejecutarlo:

```
$ python3 python_intro.py
Hey Sonja!
```

¿Ves lo que pasó ahí?

Resumen

En los últimos tres ejercicios aprendiste acerca de:

- **Comparar cosas** - en Python puedes comparar cosas haciendo uso de `>`, `>=`, `==`, `<=`, `<` y de los operadores `and` y `or`
- **Boolean** - un tipo de objeto que sólo puede tener uno de dos valores: `True` o `False`
- **Guardar archivos** - cómo almacenar código en archivos así puedes ejecutar programas más grandes
- **if... elif... else** - sentencias que te permiten ejecutar código sólo cuando se cumplen ciertas condiciones

¡Es hora de leer la última parte de este capítulo!

¡Tus propias funciones!

¿Recuerdas las funciones como `len()` que puedes ejecutar en Python? Bien, buenas noticias, ¡ahora aprenderás cómo escribir tus propias funciones!

Una función es una secuencia de instrucciones que Python debe ejecutar. Cada función en Python comienza con la palabra clave `def`, se le asigna un nombre y puede tener algunos parámetros. Vamos a empezar con algo fácil. Reemplaza el código en **python_intro.py** con lo siguiente:

```
def hi():  
    print('Hi there!')  
    print('How are you?')  
  
hi()
```

Bien, ¡nuestra primera función está lista!

Te preguntarás por qué hemos escrito el nombre de la función en la parte inferior del archivo. Esto es porque Python lee el archivo y lo ejecuta desde arriba hacia abajo. Así que para poder utilizar nuestra función, tenemos que reescribir su nombre en la parte inferior.

Ejecutemos esto y veamos qué sucede:

```
$ python3 python_intro.py  
Hi there!  
How are you?
```

¡Eso fue fácil! Vamos a construir nuestra primera función con parámetros. Utilizaremos el ejemplo anterior - una función que dice 'Hi' a la persona que ejecuta el programa - con un nombre:

```
def hi(name):
```

Como puedes ver, ahora dimos a nuestra función un parámetro que llamamos `name` :

```
def hi(name):  
    if name == 'Ola':  
        print('Hi Ola!')  
    elif name == 'Sonja':  
        print('Hi Sonja!')  
    else:  
        print('Hi anonymous!')  
  
hi()
```

Como puedes notar, tuvimos que poner dos indentaciones antes de la función `print` porque `if` necesita saber lo que debería ocurrir cuando se cumple la condición. Vamos a ver cómo funciona:

```
$ python3 python_intro.py  
Traceback (most recent call last):  
File "python_intro.py", line 10, in <module>  
    hi()  
TypeError: hi() missing 1 required positional argument: 'name'
```

Oops, un error. Por suerte, Python nos da un mensaje de error bastante útil. Nos dice que la función `hi()` (la que definimos) tiene un argumento requerido (llamado `name`) y que se nos olvidó pasarlo al llamar a la función. Vamos a arreglarlo en la parte inferior del archivo:

```
hi("Ola")
```

y lo ejecutamos otra vez:

```
$ python3 python_intro.py  
Hi Ola!
```

¿Y si cambiamos el nombre?

```
hi("Sonja")
```

y lo corremos:

```
$ python3 python_intro.py  
Hi Sonja!
```


Ahora, ¿qué crees que pasará si escribes otro nombre allí? (No Ola o Sonja). Pruébalo y verás si tienes razón. Esto debería imprimir:

```
Hi anonymous!
```

Esto es increíble, ¿verdad? De esta forma no tienes que repetir todo cada vez que deseas cambiar el nombre de la persona a la que la función debería saludar. Y eso es exactamente por qué necesitamos funciones - ¡para no repetir tu código!

Vamos a hacer algo más inteligente - hay más de dos nombres, y escribir una condición para cada uno sería difícil, ¿no?

```
def hi(name):  
    print('Hi ' + name + '!')  
  
hi("Rachel")
```

Ahora vamos a llamar al código:

```
$ python3 python_intro.py  
Hi Rachel!
```

¡Felicidades! Acabas de aprender cómo escribir funciones :)

Bucles

Esta ya es la última parte. ¿Eso fue rápido, verdad? :)

Como hemos mencionado, los programadores son perezosos, no les gusta repetir cosas. La programación intenta automatizar las cosas, así que no queremos saludar a cada persona por su nombre manualmente, ¿verdad? Es ahí donde los bucles se vuelven muy útiles.

¿Todavía recuerdas las listas? Hagamos una lista de las chicas:

```
girls = ['Rachel', 'Monica', 'Phoebe', 'Ola', 'You']
```

Queremos saludar a todas ellas por su nombre. Tenemos la función `hi` que hace eso, así que vamos a usarla en un bucle:

```
for name in girls:
```

La sentencia `for` se comporta de manera similar a la sentencia `if`, el código que sigue continuación debe estar indentado usando cuatro espacios.

Aquí está el código completo que estará en el archivo:

```
def hi(name):  
    print('Hi ' + name + '!')  
  
girls = ['Rachel', 'Monica', 'Phoebe', 'Ola', 'You']  
for name in girls:  
    hi(name)  
    print('Next girl')
```

y cuando lo ejecutamos:

```
$ python3 python_intro.py  
Hi Rachel!  
Next girl  
Hi Monica!  
Next girl  
Hi Phoebe!  
Next girl  
Hi Ola!  
Next girl  
Hi You!  
Next girl
```

Como puedes ver, todo lo que pones con una indentación dentro de una sentencia `for` será repetido para cada elemento de la lista `girls`.

También puedes usar el `for` en números usando la función `range`:

```
for i in range(1, 6):  
    print(i)
```

Lo que imprimirá:

```
1  
2  
3  
4  
5
```

`range` es una función que crea una lista de números en serie (estos números son proporcionados por ti como parámetros).

Ten en cuenta que el segundo de estos dos números no será incluido en la lista que retornará Python (es decir, `range(1, 6)` cuenta desde 1 a 5, pero no incluye el número 6).

Resumen

Eso es todo. ¡Eres genial! Esto no fue tan fácil realmente, así que deberías sentirte orgullosa de ti misma. ¡Estamos muy orgullosos de que hayas llegado hasta aquí!

Tal vez quieras hacer algo distinto por un momento - estirarte, caminar un poco, descansar tus ojos - antes de pasar al siguiente capítulo. :)



¿Qué es Django?

Django (*gdh/'dʒæŋɡou/jang-goh*) es un framework para aplicaciones web gratuito y open source, escrito en Python. Es un WEB framework - un conjunto de componentes que te ayudan a desarrollar sitios web más fácil y rápidamente.

Verás, cuando estás construyendo un sitio web, frecuentemente necesitas un conjunto de componentes similares: una manera de manejar la autenticación de usuarios (registrarse, iniciar sesión, cerrar sesión), un panel de administración para tu sitio web, formularios, una forma de subir archivos, etc.

Por suerte para ti, hace tiempo varias personas notaron que los desarrolladores web enfrentan problemas similares cuando construyen un sitio nuevo, por eso juntaron cabezas y crearon frameworks (Django es uno de ellos) que te ofrecen componentes listos para usarse.

Los frameworks existen para ahorrarte tener que reinventar la rueda y ayudarte a aliviar la carga cuando construyes un sitio.

¿Por qué necesitas un framework?

Para entender para que es Django, necesitamos mirar mas de cerca a los servidores. Lo primero es que el servidor necesita saber que quieres que te sirva una página web.

Imagina un buzón (puerto) el cual es monitoreado por cartas entrantes (peticiones). Esto es realizado por un servidor web. El servidor web lee la carta, y envía una respuesta con una página web. Pero cuando quieres enviar algo, tienes que tener algún contenido. Y Django es algo que te ayuda a crear el contenido.

¿Qué sucede cuando alguien solicita una página web de tu servidor?

Cuando llega una petición a un servidor web es pasado a Django que intenta averiguar lo que realmente es solicitado. Toma primero una dirección de página web y trata de averiguar qué hacer. Esta parte es realizada por **urlresolver** de Django (tenga en cuenta que la dirección de un sitio web es llamada URL - Uniform Resource Locator - así que el nombre *urlresolver* tiene sentido). Este no es muy inteligente - toma una lista de patrones y trata de igualar la URL. Django comprueba los patrones de arriba hacia abajo y si algo se coincide

entonces Django le pasa la solicitud a la función asociada (que se llama *vista*).

Imagina a un cartero llevando una carta. Ella está caminando por la calle y comprueba cada número de casa con el que está en la carta. Si coincide, ella deja la carta allí. Así es como funciona el `urlresolver`!

En la función de *vista* se hacen todas las cosas interesantes: podemos mirar a una base de datos para buscar alguna información. ¿Tal vez el usuario pidió cambiar algo en los datos? Como una carta diciendo "Por favor cambia la descripción de mi trabajo." La *vista* puede comprobar si tienes permitido hacer eso, entonces actualizar la descripción del trabajo para usted y devolverle un mensaje: "¡hecho!". Entonces la *vista* genera una respuesta y Django puede enviarla al navegador del usuario.

Por supuesto, la descripción anterior se simplifica un poco, pero no necesitas saber todas las cosas técnicas aun. Tener una idea general es suficiente.

Así que en lugar de meternos demasiado en los detalles, simplemente comenzaremos creando algo con Django y aprenderemos todas las piezas importantes en el camino!

Instalación de Django

Parte de este capítulo esta basado en los tutoriales de Geek Girls Carrots (<http://django.carrots.pl/>).

Parte de este capítulo se basa en el [django-marcador tutorial](#) bajo licencia de Creative Commons Attribution-ShareAlike 4.0 internacional. El tutorial de django-marcador tiene derechos de autor de Markus Zapke-Gündemann et al.

Entorno virtual

Antes de instalar Django, instalaremos una herramienta extremadamente útil que ayudará a mantener tu entorno de desarrollo ordenado en su computadora. Es posible omitir este paso, pero es muy recomendable no hacerlo - ¡comenzar con la mejor configuración posible ayudara a evitar muchos problemas en el futuro!

Así que, vamos a crear un **entorno virtual** (también llamado un *virtualenv*). Aislará la configuración Python/Django en base de cada proyecto, lo que significa que cualquier cambio que realice en un sitio web no afectará a otros que también estés desarrollando. Genial, ¿no?

Todo lo que necesitas hacer es encontrar un directorio en el que desees crear el `virtualenv` ; tu directorio home, por ejemplo. En Windows puede verse como `C:\Users\Name` (donde `nombre` es el nombre de tu usuario).

Para este tutorial usaremos un nuevo directorio `django girls` en tu directorio home:

```
mkdir.djangogirls
cd.djangogirls
```

Haremos un `virtualenv` llamado `myvenv` . El comando general estará en el formato:

```
python3 -m venv myvenv
```

Windows

Para crear un nuevo `virtualenv` , debes abrir la consola (te lo indicamos unos cuantos capítulos antes, ¿recuerdas?) y ejecuta `C:\Python34\python -m venv myvenv` . Se verá así:

```
C:\Users\Name\djangoirls> C:\Python34\python -m venv myvenv
```

en donde `C:\Python34\python` es el directorio en el que instalaste Python previamente y `myvenv` es el nombre de tu `virtualenv`. Puedes utilizar cualquier otro nombre, pero asegúrate de usar minúsculas y no dejar espacios, acentos o caracteres especiales. También es una buena idea mantener el nombre corto. ¡Vas a referirte a él mucho!

Linux y OS X

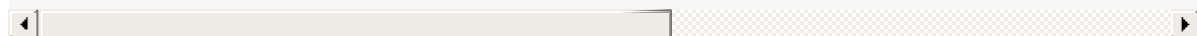
Crear un `virtualenv` en Linux y OS X es tan simple como ejecutar `python3 -m venv myvenv`. Se verá así:

```
~/djangoirls$ python3 -m venv myvenv
```

`myvenv` es el nombre de tu `virtualenv`. Puedes usar cualquier otro nombre, pero mantén el uso de minúsculas y no incluyas espacios. También es una buena idea mantener el nombre corto. ¡Vas a referirte a él mucho!

Nota: Actualmente, iniciar el entorno virtual en Ubuntu 14.04 de esta manera produce el siguiente error:

```
Error: Command '['/home/eddie/Slask/tmp/venv/bin/python3', '-Im', 'ensurepip', '--up
```



Para evitar esto, utiliza directamente el comando `virtualenv`.

```
~/djangoirls$ sudo apt-get install python-virtualenv
~/djangoirls$ virtualenv --python=python3.4 myvenv
```

Trabajar con virtualenv

Este comando anterior creará un directorio llamado `myvenv` (o cualquier nombre que hayas escogido) que contiene nuestro entorno virtual (básicamente un montón de archivos y carpetas). Todo lo que queremos hacer ahora es iniciarlo ejecutando:

```
C:\Users\Name\djangoirls> myvenv\Scripts\activate
```

en Windows, o:

```
~/djangogirls$ source myvenv/bin/activate
```

en OS X y Linux.

¡Recuerda reemplazar `myvenv` con tu nombre de `virtualenv` que hayas elegido!

Nota: a veces la `fuentes` podría no estar disponible. En esos casos trata de hacerlo esto:

```
~/djangogirls$ . myvenv/bin/activate
```

Sabrás que tienes `virtualenv` iniciado cuando veas que parece este mensaje en la consola:

```
(myvenv) C:\Users\Name\djangogirls>
```

O:

```
(myvenv) ~/djangogirls$
```

¡Nota que el prefijo `(myvenv)` aparece!

Cuando trabajes en un entorno virtual, `python` automáticamente se referirá a la versión correcta, de modo que puedes utilizar `python` en vez de `python3`.

Tenemos todas las dependencias importantes en su lugar. ¡Finalmente podemos instalar Django!

Instalar Django

Ahora que tienes tu `virtualenv` iniciado, puedes instalar Django usando `pip`. En la consola, ejecuta `pip install django==1.8` (fíjate que utilizamos un doble signo igual: `==`).

```
(myvenv) ~$ pip install django==1.8
Downloading/unpacking django==1.8
Installing collected packages: django
Successfully installed django
Cleaning up...
```

En Windows

Si obtienes un error al ejecutar pip en Windows comprueba si la ruta de tu proyecto contiene espacios, acentos o caracteres especiales (por ejemplo, `C:\Users\User Name\djangogirls`). Si lo tiene, por favor considera moverla a otro lugar sin espacios, acentos o caracteres especiales (sugerencia: `C:\djangogirls`). Después de hacerlo ejecuta nuevamente el comando anterior.

en Linux

Si obtienes un error al correr pip en Ubuntu 12.04 ejecuta `python -m pip install- U - force-resintall pip` para arreglar la instalación de pip en el virtualenv.

Eso es todo! Ahora estás listo (por fin) para crear una aplicación Django!

¡Tu primer proyecto en Django!

Parte de este capítulo está basado en los tutoriales de Geek Girls Carrots (<http://django.carrots.pl/>).

Parte de este capítulo se basa en el [django-marcador tutorial](#) bajo licencia de Creative Commons Attribution-ShareAlike 4.0 internacional. El tutorial de django-marcador tiene derechos de autor de Markus Zapke-Gündemann et al.

Vamos a crear un simple blog!

El primer paso para crearlo es para iniciar un nuevo proyecto en Django. Básicamente, esto significa que podrás correr algunos scripts proporcionados por Django que crearán el esqueleto de un proyecto para nosotros: un montón de directorios y archivos que vamos a utilizar más adelante.

Los nombres de algunos archivos y directorios son muy importantes para Django. No deberías renombrar los archivos que estamos a punto de crear. Moverlos a un lugar diferente tampoco es una buena idea. Django tiene que mantener una cierta estructura para ser capaz de encontrar cosas importantes.

Recuerda correr todo en el virtualenv. Si no ves un prefijo `(myvenv)` en tu consola necesitas activar tu virtualenv. Explicamos cómo hacer eso en el capítulo de **Instalación de Django** en la sección **Trabajando con virtualenv**. Puedes hacerlo escribiendo el siguiente comando: `myvenv\Scripts\activate` en Windows o `myvenv/bin/activate` en Mac OS / Linux.

Nota Controla dos veces que incluiste el punto (`.`) al final del comando, es importante porque le dice al script que instale Django en el directorio actual.

En la consola debes ejecutar (recuerda no escribir `(myvenv) ~/djangogirls$`, ¿ok?):

```
(myvenv) ~/djangogirls$ django-admin startproject mysite .
```

En Windows:

```
(myvenv) C:\Users\Name\djangogirls> django-admin.py startproject mysite .
```

`django-admin.py` es un script que creará los archivos y directorios para ti. Ahora deberías tener una estructura de directorios parecida a esto:

```

djangogirls
├── manage.py
└── mysite
    ├── settings.py
    ├── urls.py
    ├── wsgi.py
    └── __init__.py

```

`manage.py` es un script que ayuda con la administración del sitio. Con ello podremos iniciar un servidor web en nuestro ordenador sin necesidad de instalar nada más, entre otras cosas.

El archivo `settings.py` contiene la configuración de tu sitio web.

¿Recuerdas cuando hablamos de un cartero que debía comprobar donde entregar una carta? El archivo `urls.py` contiene una lista de los patrones utilizados por `urlresolver`.

Ignoremos los otros archivos por ahora - no los cambiaremos. ¡Lo único que debes recordar es no borrarlos por accidente!

Cambiando la configuración

Vamos a hacer algunos cambios en `mysite/settings.py`. Abre el archivo usando el editor de código que has instalado anteriormente.

Sería bueno tener el horario correcto en nuestro sitio web. Ve a la [lista de husos horarios de Wikipedia](#) y copia tu zona horaria (TZ). (por ejemplo, `Europe/Berlin`)

En `settings.py`, encuentra la línea que contiene `TIME_ZONE` y modifícala para elegir tu propia zona horaria:

```
TIME_ZONE = 'Europe/Berlin'
```

Modificando "Europe/Berlin" como corresponda

También necesitaremos agregar una ruta para los archivos estáticos (aprenderemos todo sobre los archivos estáticos y CSS más tarde en este tutorial). Ve hacia abajo hasta el *final* del archivo, y justo por debajo de la entrada `STATIC_URL`, agrega una nueva llamada

```
STATIC_ROOT :
```

```

STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')

```

Configurar una base de datos

Hay una gran variedad de opciones de bases de datos para almacenar los datos de tu sitio. Utilizaremos el que viene por defecto, `sqlite3`.

Esto ya está configurado en esta parte de tu archivo `mysite/settings.py`:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}
```

Para crear una base de datos para nuestro blog, ejecutemos lo siguiente en la consola:

`python manage.py migrate` (necesitamos estar en el directorio de `djangogirls` que contiene el archivo `manage.py`). Si eso va bien, deberías ver algo así:

```
(myvenv) ~/djangogirls$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying sessions.0001_initial... OK
```

¡Y listo! ¡Es hora de iniciar el servidor web y ver si nuestro sitio web está funcionando!

Debes estar en el directorio que contiene el archivo `manage.py` (en la carpeta `djangogirls`). En la consola, podemos iniciar el servidor web ejecutando `python manage.py runserver`:

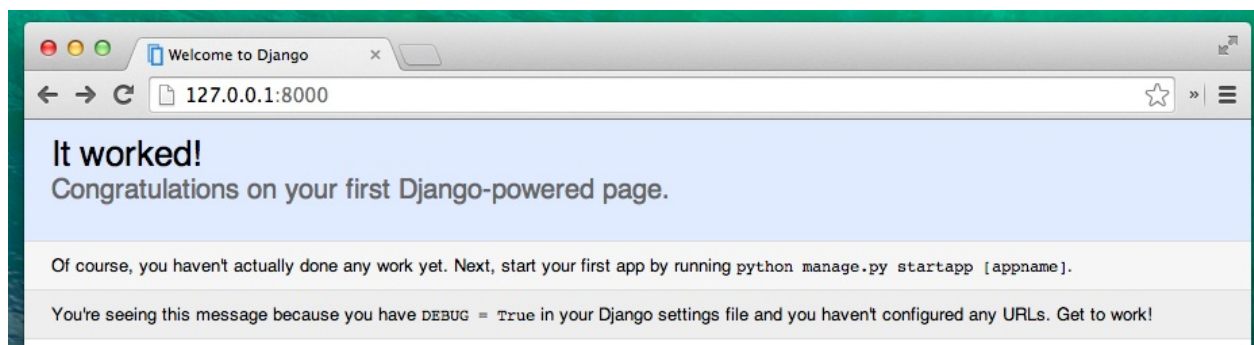
```
(myvenv) ~/djangogirls$ python manage.py runserver
```

Ahora todo lo que debes hacer es controlar que tu sitio esté corriendo - abre tu navegador (Firefox, Chrome, Safari, Internet Explorer o el que utilices) e ingresa la dirección:

```
http://127.0.0.1:8000/
```

El servidor web se apropiará de tu consola hasta que lo termines manualmente: para tipear más comandos o abres una nueva terminal (y no te olvides de activar tu virtualenv allí también), o frena el servidor web yendo a la consola en la que está corriendo y presionando `Ctrl+C` - las teclas Control y C juntas (en Windows, deberás presionar `Ctrl + Break`).

¡Felicitaciones! ¡Has creado tu primer sitio web y lo has ejecutado usando un servidor web!
¿No es genial?



¿Lista para el próximo paso? ¡Es momento de crear algo de contenido!

Modelos en Django

Lo que queremos crear ahora es algo que va a almacenar todos los posts en nuestro blog. Pero para poder hacerlo tenemos que hablar un poco de acerca de algo llamado `objetos`.

Objetos

Hay un concepto en el mundo de la programación llamado `programación orientada a objetos`. La idea es que en lugar de escribir todo como una aburrida secuencia de instrucciones de programación podemos modelar cosas y definir cómo interactúan con las demás.

Entonces ¿Qué es un objeto? Es un conjunto de propiedades y acciones. Suena raro, pero te daremos un ejemplo.

Si queremos un modelar un gato crearemos un objeto `Gato` que tiene algunas propiedades, como son por ejemplo `color`, `edad`, `estado de ánimo` (es decir, bueno, malo, sueño ;)), `dueño` (que es un objeto `Persona` o, tal vez, en el caso de que el gato sea callejero, esta propiedad estará vacía).

Y luego el `Gato` tiene algunas acciones: `ronronear`, `rasguñar` o `alimentarse` (en la cual daremos al gato algunos `ComidaDeGato`, que podría ser un objeto independiente con propiedades, como por ejemplo, `sabor`).

```
Gato
-----
color
edad
humor
dueño
ronronear()
rasguñar()
alimentarse(comida_de_gato)

ComidaDeGato
-----
sabor
```

Básicamente se trata de describir cosas reales en el código con propiedades (llamadas `propiedades del objeto`) y las acciones (llamadas `métodos`).

Y ahora, ¿cómo modelamos los posts en el blog? Queremos construir un blog, ¿no?

Tenemos primero que responder algunas preguntas: ¿Qué es un post de un blog? ¿Qué características debe tener?

Bueno, seguro que nuestros posts necesitan un texto con su contenido y un título, ¿cierto? También sería bueno saber quién lo escribió, así que necesitamos un autor. Por último, queremos saber cuándo el post fue creado y publicado.

```
Post
-----
title
text
author
created_date
published_date
```

¿Qué tipo de cosas podría hacerse con una entrada del blog? Sería bueno tener algún método que publique la entrada, ¿no?

Así que vamos a necesitar el método `publicar`.

Puesto que ya sabemos lo que queremos lograr, ¡podemos empezar a modularlo en Django!

Modelo en Django

Sabiendo qué es un objeto, podemos crear un modelo en Django para nuestros posts en el blog.

Un modelo en Django es un tipo especial de objeto que se guarda en la `base de datos`. Una base de datos es una colección de datos. Allí es el lugar en el cual almacenarás la información sobre usuarios, posts del blog, etc. Utilizaremos una base de datos SQLite para almacenar nuestros datos. Este es el adaptador de base de datos predeterminada en Django -- será suficiente para nosotros por ahora.

Piensa en el modelo en la base de datos como una hoja de cálculo con columnas (campos) y filas (datos).

Creando una aplicación

Para mantener todo en orden, crearemos una aplicación separada dentro de nuestro proyecto. Es muy bueno tener todo organizado desde el principio. Para crear una aplicación, necesitamos ejecutar el siguiente comando en la consola (dentro de la carpeta de `djangogirls` donde está el archivo `manage.py`):

```
(myvenv) ~/djangogirls$ python manage.py startapp blog
```

Vas notar que se crea un nuevo directorio llamado `blog` y contiene una serie de archivos. Nuestros directorios y archivos en nuestro proyecto deberían parecerse a esto:

```
djangogirls
├── mysite
│   ├── __init__.py
│   ├── settings.py
│   ├── urls.py
│   └── wsgi.py
├── manage.py
├── blog
├── migrations
│   ├── __init__.py
│   ├── __init__.py
│   ├── admin.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
```

Después de crear una aplicación también necesitamos decirle a Django que debe utilizarla. Lo hacemos en el archivo `mysite/settings.py`. Tenemos que encontrar `INSTALLED_APPS` y añadir una línea que contiene `'blog'`, justo por encima de `)`. El producto final debe tener este aspecto:

```
INSTALLED_APPS = (
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
)
```

Creando el Modelo Post

En el archivo `blog/models.py` definimos todos los objetos llamados `Models` - este es un lugar en el cual definiremos nuestro modelo post.

Vamos abrir `blog/models.py`, quitamos todo y escribimos un código como este:

```
from django.db import models
from django.utils import timezone

class Post(models.Model):
    author = models.ForeignKey('auth.User')
```



```

title = models.CharField(max_length=200)
text = models.TextField()
created_date = models.DateTimeField(
    default=timezone.now)
published_date = models.DateTimeField(
    blank=True, null=True)

def publish(self):
    self.published_date = timezone.now()
    self.save()

def __str__(self):
    return self.title

```

Vuelve a verificar que usaste dos guiones bajos (`_`) en cada lado del `str` . Estos se utilizan con frecuencia en Python y a veces también los llamamos "dunder" (diminutivo en inglés de "double-underscore").

Da un poco de miedo, ¿verdad? Pero no te preocupes, ¡vamos a explicar qué significan estas líneas!

Todas las líneas que comienzan con `from` o `import` son líneas para añadir algo de otros archivos. Así que en vez de copiar y pegar las mismas cosas en cada archivo, podemos incluir algunas partes con `from... import ...`.

`class Post(models.Model):` - esta línea define nuestro modelo (es un objeto).

- `class` es una palabra clave que indica que estamos definiendo un objeto.
- `Post` es el nombre de nuestro modelo. Podemos darle un nombre diferente (pero debemos evitar espacios en blanco y caracteres especiales). Una clase siempre comienza con su primera letra en mayúscula.
- `models.Model` significa que `Post` es un modelo de Django, así Django sabe que debe guardarlo en la base de datos.

Ahora definimos las propiedades que hablábamos: `title` , `text` , `created_date` , `published_date` y `author` . Para hacer eso tenemos que definir un tipo de campo (¿es texto? ¿un número? ¿una fecha? ¿una relación con otro objeto - es decir, un usuario?).

- `models.CharField` - esto es como defines un texto con un número limitado de caracteres.
- `models.TextField` - esto es para textos largos sin un límite. Será ideal para el contenido de un post, ¿verdad?
- `models.DateTimeField` - esto es fecha y hora.
- `models.ForeignKey` - este es un vínculo con otro modelo.

No vamos a explicar cada pedacito de código, ya que nos tomaría demasiado tiempo. Debes echar un vistazo a la documentación de Django si quieres saber más sobre los

campos de los Modelos y cómo definir cosas diferentes a las descritas anteriormente (<https://docs.djangoproject.com/en/1.8/ref/models/fields/#field-types>).

¿Y qué sobre `def publish(self):`? Es exactamente nuestro método `publish` que mencionamos anteriormente. `def` significa que se trata de una función o método. `publish` es el nombre del método. Puedes cambiarlo, si quieres. La regla es que usamos minúsculas y guiones bajos en lugar de espacios (es decir, si quieres tener un método que calcule el precio medio, este podría llamarse `calculate_average_price`).

Los métodos muy a menudo devuelven algo. Hay un ejemplo de esto en el método `__str__`. En este escenario, cuando llamamos a `__str__()` obtendremos un texto (**string**) con un título de Post.

Si algo todavía no está claro sobre modelos, ¡no dudes en preguntar a tu tutor! Sabemos que es muy complicado, sobre todo cuando estás entendiendo qué funciones y objetos son mientras sigues este documento. Con suerte, ¡todo tiene un poco más sentido para ti ahora!

Crear tablas para los modelos en tu base de datos

El último paso es añadir nuestro nuevo modelo a nuestra base de datos. Primero tenemos que hacer que Django sepa que tenemos algunos cambios en nuestro modelo (acabamos de crearlo), escribe `python manage.py makemigrations blog`. Se verá así:

```
(myvenv) ~/djangogirls$ python manage.py makemigrations blog
Migrations for 'blog':
  0001_initial.py:
    - Create model Post
```

Django preparará un archivo de migración que tenemos que aplicar ahora a nuestra base de datos escribiendo `python manage.py migrate blog`. El resultado debe ser:

```
(myvenv) ~/djangogirls$ python manage.py migrate blog
Operations to perform:
  Apply all migrations: blog
Running migrations:
  Applying blog.0001_initial... OK
```

¡Hurra! Nuestro modelo de Post está ahora en nuestra base de datos. Sería bueno verlo, ¿no? ¡Dirígete al siguiente capítulo para ver cómo luce tu Post!

Administrador de Django

Para agregar, editar y borrar los posts que hemos modelado, utilizaremos el administrador de Django.

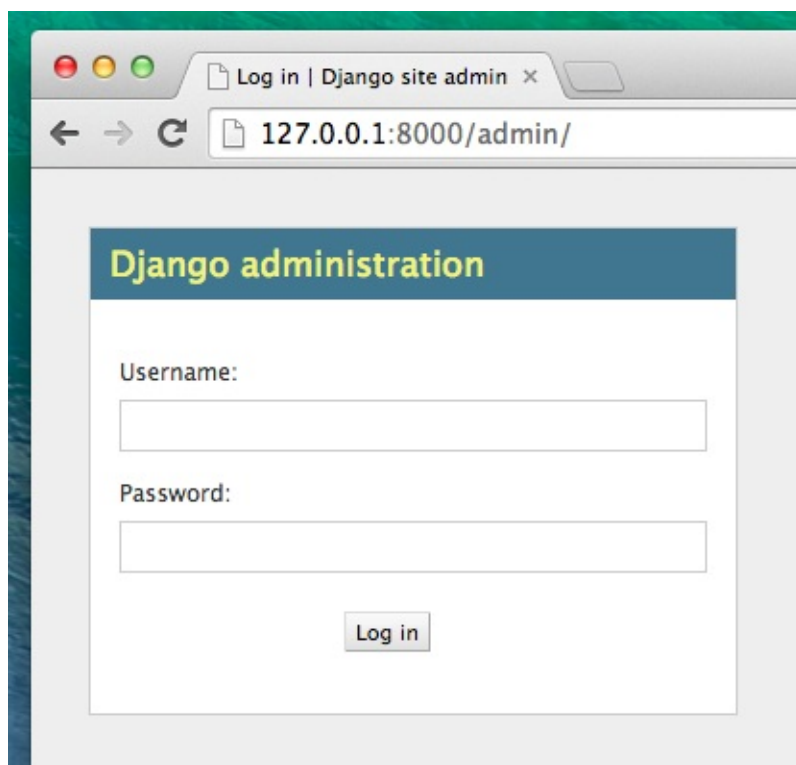
Vamos a abrir el archivo `blog/admin.py` y reemplazar su contenido con esto:

```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Como puedes ver, importamos (incluimos) el modelo `Post` definido en el capítulo anterior. Para hacer nuestro modelo visible en la página del administrador, tenemos que registrar el modelo con `admin.site.register(Post)`.

Ok, es hora de ver tu modelo `Post`. Recuerda ejecutar `python manage.py runserver` en la consola para correr el servidor web. Ve al navegador y tipea la dirección <http://127.0.0.1:8000/admin/>. Verás una página de ingreso como la que sigue:

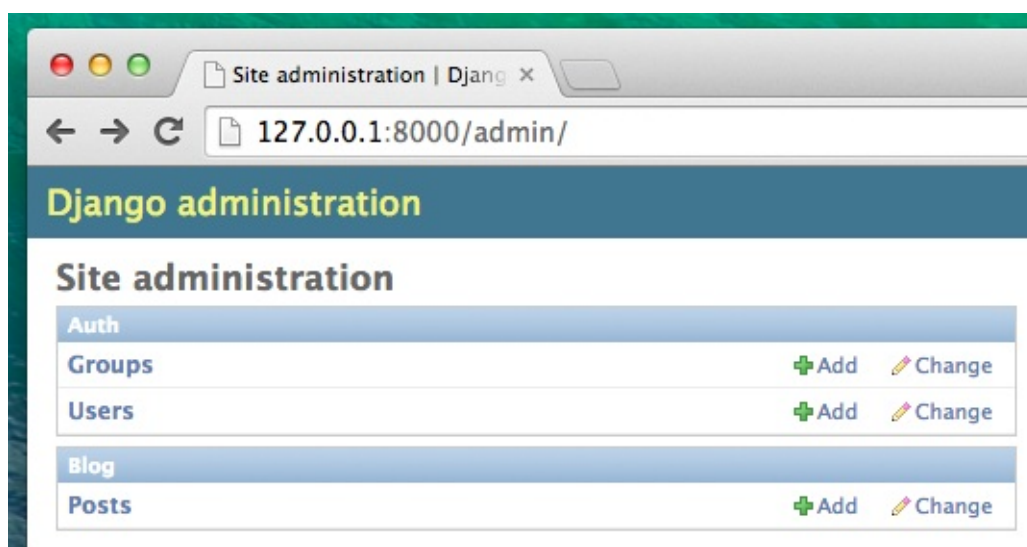


Para poder ingresar deberás crear un *superusuario* - un usuario que tiene control sobre todo lo que hay en el sitio. Vuelve hacia atrás a tu línea de comandos y tipea `python manage.py createsuperuser`, presiona enter y tipea tu nombre de usuario (en minúsculas, sin espacios), dirección de email y contraseña cuando sean requeridos. No te preocupes que no puedes

ver tu contraseña mientras la tipeas - así es como debe ser. Simplemente tipéala y presiona 'Enter' para continuar. La salida de este comando debería verse así (nombre de usuario y email deberían ser los tuyos):

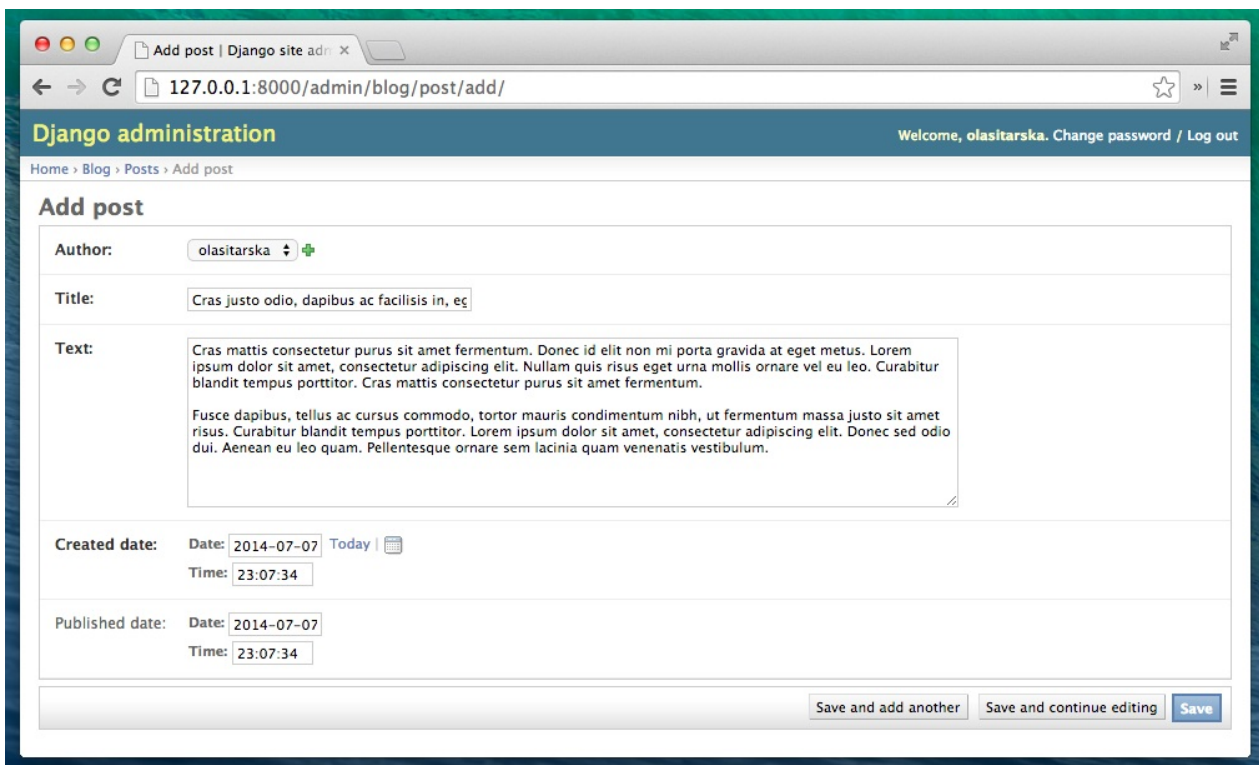
```
(myvenv) ~/djangogirls$ python manage.py createsuperuser
Username: admin
Email address: admin@admin.com
Password:
Password (again):
Superuser created successfully.
```

Vuelve a tu navegador e ingresa con las credenciales de super usuario que elegiste, ahora deberías poder ver el panel de administración de Django.



Ve a Posts y experimenta un poco con esto. Agrega cinco o seis posts del blog. No te preocupes por el contenido - puedes simplemente copiar y pegar texto de este tutorial en el contenido de tus posts para ahorrar tiempo :).

Asegúrate de que por lo menos dos o tres posts (pero no todas) tienen la fecha de publicación. Será útil luego.



The screenshot shows a web browser window with the URL `127.0.0.1:8000/admin/blog/post/add/`. The page title is "Django administration" and the user is logged in as "olasitarska". The breadcrumb trail is "Home > Blog > Posts > Add post". The form is titled "Add post" and contains the following fields:

- Author:** A dropdown menu showing "olasitarska" with a plus icon to add more.
- Title:** A text input field containing "Cras justo odio, dapibus ac facilisis in, egestas eget eget".
- Text:** A large text area containing two paragraphs of Lorem Ipsum text.
- Created date:** A date and time picker showing "Date: 2014-07-07" and "Time: 23:07:34".
- Published date:** A date and time picker showing "Date: 2014-07-07" and "Time: 23:07:34".

At the bottom right of the form, there are three buttons: "Save and add another", "Save and continue editing", and "Save".

Si quieres saber más sobre el administrador de Django, puedes visitar la documentación de Django: <https://docs.djangoproject.com/en/1.8/ref/contrib/admin/>

Este probablemente sea un buen momento para tomar un café (o té) o algo para comer y re-energizarte. Creaste tu primer modelo de Django - ¡mereces un pequeño recreo!

¡Despliega!

Nota: El siguiente capítulo puede ser a veces un poco difícil de superar. Se persistente y acábalo. El despliegue es una parte importante del proceso en el desarrollo web. Este capítulo está situado en el medio del tutorial para que tu tutor pueda ayudarte a poner tu sitio web en línea, lo que puede ser un proceso algo más complicado. Esto significa que podrás acabar el tutorial por tu cuenta si se te acaba el tiempo.

Hasta ahora tu sitio web estaba disponible sólo en tu ordenador, ¡ahora aprenderás cómo desplegarlo! El despliegue es el proceso de publicar tu aplicación en Internet para que la gente pueda acceder y ver tu aplicación :).

Como ya has aprendido, un sitio web tiene que estar en un servidor. Hay muchos proveedores, pero usaremos uno que tiene un proceso de despliegue relativamente simple: [PythonAnywhere](#). PythonAnywhere es gratis para pequeñas aplicaciones que no tienen demasiados visitantes, definitivamente suficiente para este caso.

El otro servicio externo que vamos a utilizar es [GitHub](#), un servicio de alojamiento de código. Hay otras opciones por ahí, pero hoy en día casi todos los programadores tienen una cuenta de GitHub, ¡y ahora tú también la vas a tener!

Usaremos GitHub como paso intermedio para transportar nuestro código desde y hasta PythonAnywhere.

Git

Git es un "sistema de control de versiones" usado por muchos programadores - es un sistema que registra los cambios en los archivos a través del tiempo de forma tal que puedas acceder a versiones específicas cuando lo desees. Es muy similar a la opción de "registrar cambios" en Microsoft Word, pero mucho más poderoso.

Instalar Git

Windows

Puedes descargar Git de git-scm.com. Puedes hacer clic en "Next" para todos los pasos excepto en uno; en el quinto paso titulado "Adjusting your PATH environment", elije "Run Git and associated Unix tools from the Windows command-line" (la última opción). Aparte de

eso, los valores por defecto funcionarán bien. "Checkout Windows-style, commit Unix-style line endings" también está bien.

MacOS

Descarga Git de git-scm.com y sigue las instrucciones.

Linux

Si no lo tienes ya instalado, git debería estar disponible a través del administrador de paquetes, prueba con:

```
sudo apt-get install git
# o
sudo yum install git
```

Iniciar nuestro repositorio Git

Git rastrea los cambios realizados a un grupo determinado de ficheros en lo que llamamos un repositorio de código (o "repo" para abreviar). Iniciemos uno para nuestro proyecto. Abre la consola y ejecuta los siguientes comandos en el directorio de `djangogirls`:

Nota: Comprueba el directorio de trabajo actual con el comando `pwd` (OSX/Linux) o `cd` (Windows) antes de inicializar el repositorio. Deberías estar en la carpeta `djangogirls`.

```
$ git init
Initialized empty Git repository in ~/djangogirls/.git/
$ git config user.name "Tu nombre"
$ git config user.email tú@ejemplo.com
```

Inicializar el repositorio git es algo que sólo necesitamos hacer una vez por proyecto (y no tendrás que volver a poner tu usuario y correo electrónico nunca más)

Git llevará un registro de los cambios realizados en todos los ficheros y carpetas en este directorio, pero hay algunos ficheros que queremos que ignore. Esto lo hacemos creando un fichero llamado `.gitignore` en el directorio base. Abre tu editor y crea un nuevo fichero con el siguiente contenido:

```
*.pyc
__pycache__
myenv
```

```
db.sqlite3
.DS_Store
```

Y guárdalo como `.gitignore` en la primera carpeta "djangogirls".

Nota: ¡El punto al principio del nombre del fichero es importante! Si tienes dificultades para crearlo (a los Mac no les gusta que crees ficheros que empiezan por punto desde Finder, por ejemplo), usa la opción "Guardar como" en tu editor, eso no falla.

Es buena idea utilizar el comando `git status` antes de `git add` o cuando no estés segura de lo que va a hacer, para evitar cualquier sorpresa (por ejemplo, añadir o hacer commit de ficheros no deseados). El comando `git status` devuelve información sobre los ficheros sin seguimiento (untracked), modificados, preparados (staged), el estado de la rama y mucho más. La salida debería ser similar a:

```
$ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore
blog/
manage.py
mysite/

nothing added to commit but untracked files present (use "git add" to track)
```

Y finalmente guardamos nuestros cambios. Ve a la consola y ejecuta estos comandos:

```
$ git add -A .
$ git commit -m "Mi app Django Girls, primer commit"
[...]
13 files changed, 200 insertions(+)
create mode 100644 .gitignore
[...]
create mode 100644 mysite/wsgi.py
```

Enviar nuestro código a GitHub

Visita [GitHub.com](https://github.com) y registra una nueva cuenta de usuario gratuita. Luego, crea un nuevo repositorio con el nombre "my-first-blog". Deja desmarcada la opción "Initialise with a README", deja la opción `.gitignore` en blanco (lo hemos hecho a mano) y deja la licencia como "None".

Owner **Repository name**

hjwp / my-first-blog ✓

Great repository names are short and memorable. Need inspiration? How about **ducking-octo-tyrion**.

Description (optional)

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: **None** Add a license: **None** ⓘ

Create repository

Nota El nombre `my-first-blog` es importante. Podrías elegir otra cosa, pero va a aparecer muchas veces en las instrucciones que siguen y tendrías que sustituirlo cada vez. Probablemente sea más sencillo quedarte con el nombre `my-first-blog`.

En la próxima pantalla verás la URL para clonar tu repositorio. Elige la versión "HTTPS", cópiala y en un momento la pegaremos en la consola:

https://github.com/hjwp/my-first-blog

This repository Search Explore Gist Blog Help hjwp

Unwatch 1 Star 0

Quick setup — if you've done this kind of thing before

or **HTTPS** **SSH** `https://github.com/hjwp/my-first-blog.git`

We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

...or create a new repository on the command line

```
echo "# my-first-blog" >> README.md
git init
git add README.md
git commit -m "first commit"
git remote add origin https://github.com/hjwp/my-first-blog.git
git push -u origin master
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/hjwp/my-first-blog.git
git push -u origin master
```

Code

Issues 0

Pull requests 0

Wiki

Pulse

Graphs

Settings

Ahora tenemos que conectar el repositorio Git de tu ordenador con el que está en GitHub.

```
$ git remote add origin https://github.com/<your-github-username>/my-first-blog.git
$ git push -u origin master
```

Escribe tu nombre de usuario y contraseña de GitHub y deberías ver algo así:

```
Username for 'https://github.com': hjwp
Password for 'https://hjwp@github.com':
Counting objects: 6, done.
Writing objects: 100% (6/6), 200 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/hjwp/my-first-blog.git
* [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Tu código está ahora en GitHub. ¡Ve y míralo! Verás que está en buena compañía; [Django](#), el [Tutorial de Django Girls](#) y muchos otros grandes proyectos de código abierto también alojan su código en GitHub :)

Configurar nuestro blog en PythonAnywhere

Es hora de registrar una cuenta gratuita de tipo "Beginner" en PythonAnywhere.

- www.pythonanywhere.com

Nota: Cuando elijas tu nombre de usuario ten en cuenta que la URL de tu blog tendrá la forma `nombredeusuario.pythonanywhere.com`, así que o bien elije tu propio apodo o bien un nombre que describa sobre qué trata tu blog.

Bajar nuestro código en PythonAnywhere

Cuando te hayas registrado en PythonAnywhere serás redirigida a tu panel de control o página "Consoles". Elige la opción para iniciar una consola "Bash", que es la versión PythonAnywhere de una consola, como la que tienes en tu PC

Nota: PythonAnywhere está basado en Linux, por lo que si estás en Windows la consola será un poco distinta a la que tienes en tu ordenador.

Descarguemos nuestro código desde GitHub a PythonAnywhere mediante la creación de un "clon" del repositorio. Escribe lo siguiente en la consola de PythonAnywhere:

```
$ git clone https://github.com/<tu-usuario-github>/my-first-blog.git
```

Esto va a descargar una copia de tu código en PythonAnywhere. Compruébalo escribiendo:

¡Desplegar!

```
$ tree my-first-blog
my-first-blog/
├── blog
│   ├── __init__.py
│   ├── admin.py
│   ├── migrations
│   │   ├── 0001_initial.py
│   │   └── __init__.py
│   ├── models.py
│   ├── tests.py
│   └── views.py
├── manage.py
└── mysite
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
```

Crear un virtualenv en PythonAnywhere

Tal y como hiciste en tu propio ordenador, puedes crear un virtualenv en PythonAnywhere. En la consola Bash, escribe:

```
20:20 ~ $ cd my-first-blog

20:20 ~ $ virtualenv --python=python3.4 myenv
Running virtualenv with interpreter /usr/bin/python3.4
[...]
Installing setuptools, pip...done.

20:20 ~ $ source myenv/bin/activate

(mvenv)20:20 ~ $ pip install django whitenoise
Collecting django
[...]
Successfully installed django-1.8 whitenoise-1.0.6
```

Recopilar ficheros estáticos

¿Te estabas preguntando qué es eso de "whitenoise"? Es una herramienta para servir los llamados "ficheros estáticos". Los ficheros estáticos funcionan de distinta forma en los servidores en comparación con cómo lo hacen en nuestro propio ordenador y necesitamos una herramienta como "whitenoise" para servirlos.

Aprenderemos un poco más sobre los ficheros estáticos más adelante, cuando editemos el CSS de nuestro sitio.

Por ahora sólo necesitamos ejecutar en el servidor un comando adicional llamado "collectstatic". Le dice a Django que recopile todos los ficheros estáticos que necesita en el

servidor. Por el momento, principalmente son los ficheros estáticos que hacen que el panel de administración esté bonito.

```
20:20 ~ $ python manage.py collectstatic

You have requested to collect static files at the destination
location as specified in your settings:

    /home/edith/my-first-blog/static

This will overwrite existing files!
Are you sure you want to do this?

Type 'yes' to continue, or 'no' to cancel: yes
```

Escribe "yes", ¡y ahí va! ¿No te encanta hacer que las computadoras impriman páginas y páginas de texto imposible de entender? Siempre hago ruiditos para acompañarlo. Brp, brp brp...

```
Copying '/home/edith/.virtualenvs/mvenv/lib/python3.4/site-packages/django/contrib/admin/
Copying '/home/edith/.virtualenvs/mvenv/lib/python3.4/site-packages/django/contrib/admin/
[...]
Copying '/home/edith/.virtualenvs/mvenv/lib/python3.4/site-packages/django/contrib/admin/
Copying '/home/edith/.virtualenvs/mvenv/lib/python3.4/site-packages/django/contrib/admin/
62 static files copied to '/home/edith/my-first-blog/static'.
```

Crear la base de datos en PythonAnywhere

Aquí hay otra cosa que es diferente entre tu ordenador y el servidor: éste utiliza una base de datos diferente. Por lo tanto, las cuentas de usuario y las entradas pueden ser diferentes en el servidor y en tu ordenador.

Así que inicializamos la base de datos en el servidor igual que lo hicimos en nuestro ordenador, con `migrate` y `createsuperuser`:

```
(mvenv)20:20 ~ $ python manage.py migrate
Operations to perform:
[...]
Applying sessions.0001_initial... OK

(mvenv)20:20 ~ $ python manage.py createsuperuser
```

Publicar nuestro blog como una aplicación

web

Ahora que nuestro código está en PythonAnywhere, el virtualenv está listo, los ficheros estáticos han sido recopilados y la base de datos está inicializada, estamos listas para publicarla como una aplicación web.

Haz clic en el logo de PythonAnywhere para volver al panel principal, haz clic en la pestaña **Web** y pincha en **Add a new web app**.

En la ventana de diálogo, después de confirmar el nombre de dominio, elije **manual configuration** (configuración manual) (NB la opción "Django" *no*). Luego, elije **Python 3.4** y haz clic en "Next" para terminar con el asistente.

Nota asegúrate de elegir la opción de "Manual configuration", no la de "Django". Somos demasiado buenas para la configuración por defecto de Django de PythonAnywhere ;-)

Configurar el virtualenv

Serás redirigida a la pantalla de configuración de PythonAnywhere para tu aplicación web, a la que deberás acceder cada vez que quieras hacer cambios en la aplicación del servidor.

The screenshot shows the PythonAnywhere web configuration page. At the top, there's a navigation bar with tabs: Consoles, Files, Web (selected), Schedule, and Databases. Below the tabs, the page title is "Configuration for edith.pythonanywhere.com". Under "Actions:", there are two buttons: "Reload edith.pythonanywhere.com" (green) and "Delete edith.pythonanyw" (red). The "Code:" section shows the source code path, WSGI configuration file, and Python version (3.4). The "Virtualenv:" section includes instructions and a text input field containing the path `/home/edith/my-first-blog/myenv`.

En la sección "Virtualenv", haz clic en el texto rojo que dice "Enter the path to a virtualenv" (Introduce la ruta a un virtualenv) y escribe: `/home//my-first-blog/myvenv/`

Nota: sustituye tu propio nombre de usuario como corresponda. Si cometes un error, PythonAnywhere te mostrará una pequeña advertencia.

Configurar el fichero WSGI

Django funciona utilizando el "protocolo WSGI", un estándar para servir sitios web usando Python, que PythonAnywhere soporta. La forma de configurar PythonAnywhere para que reconozca nuestro blog Django es editar un fichero de configuración WSGI.

Haz clic en el enlace "WSGI configuration file" (en la sección "Code" en la parte de arriba de la página; se llamará algo parecido a `/var/www/<tu-usuario>_pythonanywhere_com_wsgi.py`) y te redirigirá al editor.

Elimina todo el contenido actual y reemplázalo con algo como esto:

```
import os
import sys

path = '/home/<tu-usuario>/my-first-blog' # aquí utiliza tu propio usuario
if path not in sys.path:
    sys.path.append(path)

os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'

from django.core.wsgi import get_wsgi_application
from whitenoise.django import DjangoWhiteNoise
application = DjangoWhiteNoise(get_wsgi_application())
```

Nota no olvides sustituir tu propio nombre de usuario donde dice `<tu-usuario>`

Este fichero se encarga de decirle a PythonAnywhere dónde vive nuestra aplicación web y cómo se llama el fichero de configuración de Django. También configura la herramienta para ficheros estáticos "whitenoise".

Dale a **Save** y vuelve a la pestaña **Web**.

¡Todo listo! Dale al botón verde grande que dice **Reload** y podrás ver tu aplicación. Verás un enlace a ella en la parte de arriba de la página.

Consejos de depuración

Si aparece un error cuando intentas visitar tu sitio, el primer lugar que deberás revisar para obtener información de depuración es el **error log**; encontrarás un enlace a él en la pestaña Web de PythonAnywhere. Mira a ver si hay algún mensaje de error ahí. Los más recientes están al final. Los problemas más comunes incluyen

- olvidar alguno de los pasos que hicimos en la consola: crear el virtualenv, activarlo, instalar Django en él, ejecutar collectstatic, inicializar la base de datos
- cometer un error en la ruta del virtualenv en la pestaña Web; suele haber un mensajito de error de color rojo, si hay algún problema
- cometer un error en el fichero de configuración WSGI; ¿has puesto bien la ruta a la carpeta my-first-blog?

¡Tu tutor está ahí para ayudar!

¡Estás en vivo!

La página por defecto de tu sitio debería decir "Welcome to Django", igual que en tu PC local. Intenta añadir `/admin/` al final de la URL y te redirigirá al panel de administración. Ingresa con tu nombre de usuario y contraseña y verás que puedes añadir nuevas entradas en el servidor.

Date una *GRAN* palmada en la espalda; los despliegues en el servidor son una de las partes más complejas del desarrollo web y muchas veces a la gente le cuesta varios días tenerlo funcionando. Pero tú tienes tu sitio en vivo, en Internet de verdad, ¡así como suena!

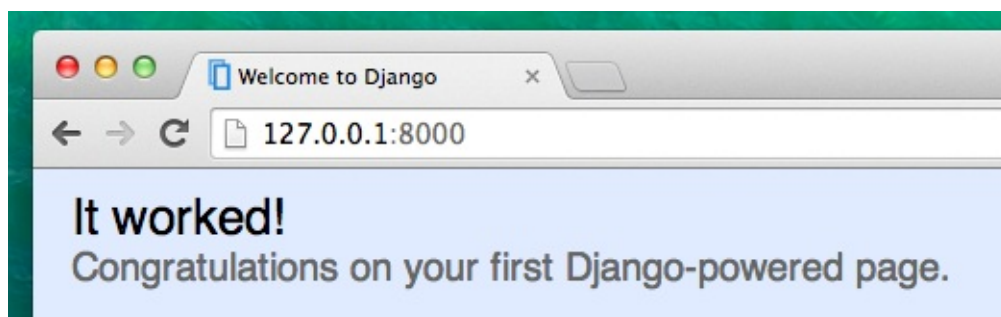
Django urls

Vamos a construir nuestra primera página web -- ¡una página de inicio para tu blog! Pero primero, vamos a aprender un poco sobre Django urls.

¿Qué es una URL?

Una URL es simplemente una dirección web, puedes ver una URL cada vez que visitas cualquier sitio web - es visible en la barra de direcciones de tu navegador (¡Sí!

`127.0.0.1:8000` es una URL. Y <http://djangogirls.com> es también una URL):



Cada página en Internet necesita su propia URL. De esta manera tu aplicación sabe lo que debe mostrar a un usuario que abre una URL. En Django se usa algo llamado `URLconf` (configuración de URL), un conjunto de patrones que Django intentará hacer coincidir con la dirección URL recibida para encontrar la vista correcta.

¿Cómo funcionan las URLs en Django?

Vamos a abrir el archivo `mysite/urls.py` y ver cómo es:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    # Examples:
    # url(r'^$', 'mysite.views.home', name='home'),
    # url(r'^blog/', include('blog.urls')),

    url(r'^admin/', include(admin.site.urls)),
]
```

Como puedes ver, Django ya puso algo aquí para nosotros.

Las líneas que comienzan con `#` son comentarios - significa que esas líneas no serán ejecutadas por Python. Muy útil, ¿verdad?

Ya está aquí la URL de admin, que visitaste en el capítulo anterior:

```
url(r'^admin/', include(admin.site.urls)),
```

Esto significa que para cada URL que empieza con `admin/` Django encontrará su correspondiente *view*. En este caso estamos incluyendo en una sola línea muchas URLs de admin, así no está todo comprimido en este pequeño archivo - es más limpio y legible.

Regex

¿Te preguntas cómo Django coincide las direcciones URL con las vistas? Bueno, esta parte es difícil. Django utiliza `regex` -- expresiones regulares. Regex tiene muchas (¡un montón!) de normas que forman un patrón de búsqueda. Dado que las expresiones regulares son un tema avanzado, no entraremos en detalles sobre su funcionamiento.

Si te interesa entender cómo creamos esos patrones, aquí hay un ejemplo del proceso - solamente necesitaremos un subconjunto de reglas limitado para expresar el patrón que estamos buscando:

```
^ denota el principio del texto
$ denota el final del texto
\d representa un dígito
+ indica que el ítem anterior debería ser repetido <strong>por lo menos</strong> una vez
() para encerrar una parte del patrón
```

Cualquier otra cosa en la definición del URL será tomada literalmente.

Ahora imagina que tienes un sitio web con una dirección como esta:

`http://www.mysite.com/post/12345/`, donde `12345` es el número de post.

Escribir vistas separadas para todos los números de post sería realmente molesto. Con las expresiones regulares podemos crear un patrón que coincidirá la URL y extraerá el número para nosotras: `^post/(\d+)/$`. Analicemos esta expresión parte por parte para entender qué es lo que estamos haciendo aquí:

- `^post/` le está diciendo a Django que tome cualquier cosa que tenga `post/` al principio del URL (justo antes de `^`)
- `(\d+)` significa que habrá un número (de uno o más dígitos) y que queremos que ese

número sea capturado y extraído

- `/` le dice a Django que otro carácter `/` debería venir a continuación
- `$` indica el final del URL, lo que significa que sólo cadenas finalizando con `/` coincidirán con este patrón

¡Tu primer URL de Django!

¡Es hora de crear nuestro primer URL! Queremos que `'http://127.0.0.1:8000/'` sea la página de inicio de nuestro blog y que muestre una lista de posts.

También queremos mantener el archivo `mysite/urls.py` limpio, así que importaremos URLs de nuestro `blog` al archivo `mysite/urls.py` principal.

Elimina las líneas comentadas (líneas comenzando con `#`) y agrega una línea que importará `blog.urls` en el url principal (`' '`).

Tu archivo `mysite/urls.py` debería verse como este:

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'', include('blog.urls')),
]
```

Django ahora redirigirá todo lo que vaya hacia `'http://127.0.0.1:8000/'` a `blog.urls` y buscará más instrucciones allí.

Cuando escribes expresiones regulares en Python acostúmbrate a poner `r` al principio de la cadena - esto es solamente una pista para que Python entienda que la cadena contendrá caracteres especiales que no son para ser interpretados por Python sino que son parte de la expresión regular.

blog.urls

Crea un nuevo archivo vacío `blog/urls.py`. ¡Muy bien! Agrega estas primeras dos líneas:

```
from django.conf.urls import include, url
from . import views
```

Aquí solo estamos importando los métodos de Django y todas nuestras `views` del `blog`

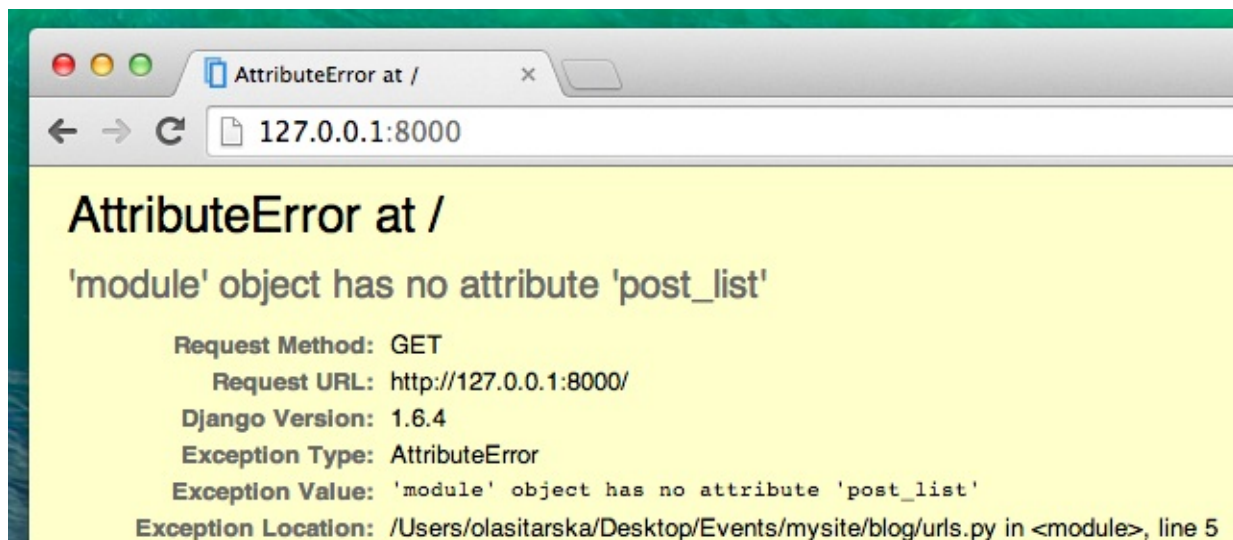
(todavía no tenemos ninguna, pero lo haremos en un minuto)

Luego de esto, podemos agregar nuestro primer patrón URL:

```
urlpatterns = [  
    url(r'^$', views.post_list),  
]
```

Como puedes ver, ahora estamos asignando una `view` llamada `post_list` al URL `^$`. Esta expresión regular coincidirá con `^` (un inicio) seguido de `$` (un final) - por lo tanto, sólo una cadena vacía coincidirá. Y esto es correcto, ya que en los URL resolvers de Django `'http://127.0.0.1:8000/'` no es parte del URL. Este patrón mostrará a Django que `views.post_list` es el lugar correcto al que ir si alguien ingresa a tu sitio web con la dirección `'http://127.0.0.1:8000/'`.

¿Todo bien? Abre <http://127.0.0.1:8000/> en tu navegador para ver el resultado.



No hay más un "It works", ¿verdad? No te preocupes, es solamente una página de error, ¡nada que nos asuste! De hecho, son bastante útiles:

Puedes leer que no hay ningún **atributo 'post_list'**. ¿*post_list* te recuerda algo? ¡Así es como llamamos a nuestra vista! Esto significa que todo está en su lugar, sólo que no creamos nuestra *view* todavía. No te preocupes, ya llegaremos a eso.

Si quieres saber más sobre Django URLconfs, mira la documentación oficial:

<https://docs.djangoproject.com/en/1.8/topics/http/urls/>

Vistas de Django - ¡Es hora de crear!

Es hora de deshacerse del error que hemos creado en el capítulo anterior :)

Una *View* es un lugar donde ponemos la "lógica" de nuestra aplicación. Se solicitará información del `model` que creaste anteriormente y se pasará a una `view` que crearás en el próximo capítulo. Las vistas son sólo métodos de Python que son un poco más complicados que lo que hicimos en el capítulo de **Introducción a Python**.

Las Vistas se colocan en el archivo `views.py`. Agregaremos nuestras *views* al archivo `blog/views.py`.

blog/views.py

Bien, vamos a abrir este archivo y ver lo que contiene:

```
from django.shortcuts import render

# Create your views here.
```

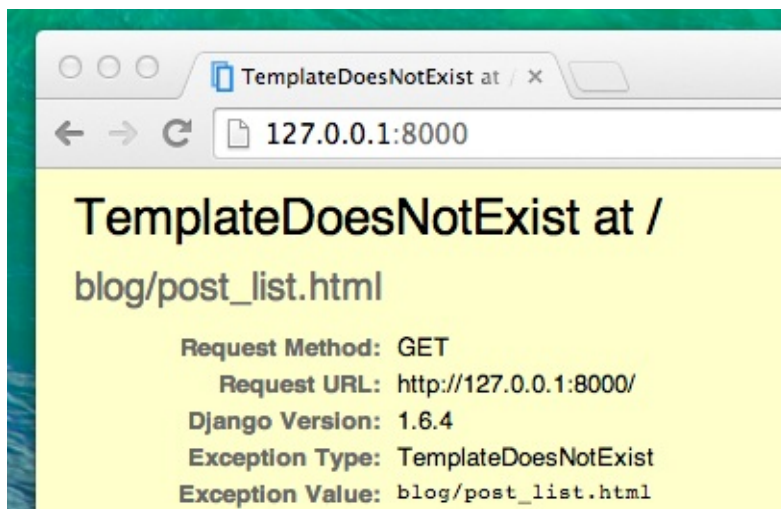
No demasiadas cosas aquí todavía. La *view* más simple puede ser como esto:

```
def post_list(request):
    return render(request, 'blog/post_list.html', {})
```

Como puedes ver, hemos creado un método (`def`) llamado `post_list` que toma un `request` y hace un `return` de un método `render` que renderizará (construirá) nuestra plantilla `blog/post_list.html`.

Guarda el archivo, dirígete a <http://127.0.0.1:8000/> y veamos lo que tenemos ahora.

¡Otro error! Leamos lo que está pasando ahora:



Este es uno fácil: *TemplateDoesNotExist*. ¡Vamos a arreglar este error creando una plantilla en el siguiente capítulo!

Aprende más acerca de las views de Django leyendo la documentación oficial:

<https://docs.djangoproject.com/en/1.8/topics/http/views/>

Introducción a HTML

Te estarás preguntando, ¿qué es una plantilla?

Una plantilla es un archivo que podemos reutilizar para presentar información diferente de forma consistente - por ejemplo, se podría utilizar una plantilla para ayudarte a escribir una carta, porque aunque cada carta puede contener un mensaje distinto y dirigirse a una persona diferente, compartirán el mismo formato.

El formato de una plantilla de Django se describe en un lenguaje llamado HTML (que es el código HTML que mencionamos en el primer capítulo **Cómo funciona Internet**).

¿Qué es HTML?

HTML es un simple código que es interpretado por tu navegador web - como Chrome, Firefox o Safari - para mostrar una página web al usuario.

HTML significa HyperText Markup Language - en español, Lenguaje de Marcas de HyperTexto. **HyperText** significa que es un tipo de texto que soporta hipervínculos entre páginas. **Markup** significa que hemos tomado un documento y lo hemos marcado con código para decirle a algo (en este caso, un navegador) cómo interpretar la página. El código HTML está construido con **etiquetas**, cada una comenzando con `<` y terminando con `>`. Estas etiquetas de marcado son **elementos**.

¡Tu primera plantilla!

Crear una plantilla significa crear un archivo de plantilla. Todo es un archivo, ¿verdad? Probablemente hayas notado esto ya.

Las plantillas se guardan en el directorio de `blog/templates/blog`. Así que primero crea un directorio llamado `templates` dentro de tu directorio `blog`. Luego crea otro directorio llamado `blog` dentro de tu directorio de `templates`:

```
blog
├── templates
│   └── blog
```

(Tal vez te preguntes por qué necesitamos dos directorios llamados `blog` - como descubrirás más adelante, esto es simplemente una útil convención de nomenclatura que

hace la vida más fácil cuando las cosas empiezan a complicarse más.)

Y ahora crea un archivo `post_list.html` (déjalo en blanco por ahora) dentro de la carpeta `blog/templates/blog`.

Mira cómo se ve su sitio web ahora: <http://127.0.0.1:8000/>

Si todavía tienes un error `TemplateDoesNotExist`, intenta reiniciar el servidor. Ve a la línea de comandos, detén el servidor pulsando `Ctrl + C` (teclas Control y C juntas) y comienza de nuevo mediante la ejecución del comando `python manage.py runserver`.



¡Ningún error más! Felicidades :) Sin embargo, por ahora, tu sitio web no está publicando nada excepto una página en blanco, porque la plantilla también está vacía. Tenemos que arreglarlo.

Añade lo siguiente a tu archivo de plantilla:

```
<html>
  <p>Hi there!</p>
  <p>It works!</p>
</html>
```

¿Cómo luce ahora tu sitio web? Haz click para ver: <http://127.0.0.1:8000/>



¡Funcionó! Buen trabajo :)

- La etiqueta más básica, `<html >`, es siempre el principio de cualquier página web y `</html >` es siempre el final. Como puedes ver, todo el contenido de la página web va desde el principio de la etiqueta `<html >` y hasta la etiqueta de cierre `</html >`
- `<p>` es una etiqueta para los elementos de párrafo; `</p>` cierra cada párrafo

Cabeza & cuerpo

Cada página HTML también se divide en dos elementos: **head** y **body**.

- **head** es un elemento que contiene información sobre el documento que no se muestra en la pantalla.
- **body** es un elemento que contiene todo lo que se muestra como parte de la página web.

Usamos `<head>` para decirle al navegador acerca de la configuración de la página y `<body>` para decir lo que realmente está en la página.

Por ejemplo, puedes ponerle un título a la página web dentro de la `<head>`, así:

```
<html>
  <head>
    <title>Ola's blog</title>
  </head>
  <body>
    <p>Hi there!</p>
    <p>It works!</p>
  </body>
</html>
```

Guarda el archivo y actualiza tu página.



¿Observas cómo el navegador ha comprendido que "Ola's blog" es el título de tu página? Ha interpretado `<title>Ola's blog</title>` y colocó el texto en la barra de título de tu navegador (también se utilizará para marcadores y así sucesivamente).

Probablemente también hayas notado que cada etiqueta de apertura coincide con una *etiqueta de cierre*, con un `/`, y que los elementos son *anidados* (es decir, no puedes cerrar una etiqueta particular hasta que todos los que estaban en su interior se hayan cerrado también).

Es como poner cosas en cajas. Tienes una caja grande, `<html></html>`; en su interior hay `<body></body>`, y que contiene las cajas aún más pequeñas: `<p></p>`.

Tienes que seguir estas reglas de etiquetas de *cierre* y de *anidación* de elementos - si no lo haces, el navegador puede no ser capaz de interpretarlos correctamente y tu página se mostrará incorrectamente.

Personaliza tu plantilla

¡Ahora puedes divertirte un poco y tratar de personalizar tu plantilla! Aquí hay algunas etiquetas útiles para eso:

- `<h1>Un título</h1>` - para tu título más importante
- `<h2>Un subtítulo</h2>` - para el título del siguiente nivel
- `<h3>Un subsubtítulo</h3>` - ... y así hasta `<h6>`
- `texto` - pone en cursiva tu texto

- `texto` - pone en negrita tu texto
- `
` - un salto de línea (no puedes colocar nada dentro de br)
- `link` - crea un vínculo
- `primer elementosegundo elemento` - crea una lista, ¡igual que esta!
- `<div></div>` - define una sección de la página

Aquí hay un ejemplo de una plantilla completa:

```
<html>
  <head>
    <title>Django Girls blog</title>
  </head>
  <body>
    <div>
      <h1><a href="">Django Girls Blog</a></h1>
    </div>

    <div>
      <p>published: 14.06.2014, 12:14</p>
      <h2><a href="">My first post</a></h2>
      <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis ves</p>
    </div>

    <div>
      <p>published: 14.06.2014, 12:14</p>
      <h2><a href="">My second post</a></h2>
      <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis ves</p>
    </div>
  </body>
</html>
```

Aquí hemos creado tres secciones `div`.

- El primer elemento `div` contiene el título de nuestro blog - es un encabezado y un enlace
- Otros dos elementos `div` contienen nuestros posts con la fecha de publicación, `h2` con un título que es clickeable y dos `p` (párrafo) de texto, uno para la fecha y uno para nuestro post.

Nos da este efecto:



¡Yaaay! Pero hasta el momento, nuestra plantilla sólo muestra exactamente **la misma información** - considerando que antes hablábamos de plantillas como permitiéndonos mostrar información **diferente** en el **mismo formato**.

Lo que queremos realmente es mostrar posts reales añadidos en nuestra página de administración de Django - y ahí es a donde vamos a continuación.

Una cosa más: ¡despliega!

Sería bueno ver todo esto disponible en Internet, ¿no? Hagamos otro despliegue en PythonAnywhere:

Haz un commit y sube tu código a GitHub

En primer lugar, veamos qué archivos han cambiado desde que hicimos el despliegue por última vez:

```
$ git status
```

Asegúrate de que estás en el directorio `djangogirls` y vamos a decirle a `git` que incluya todos los cambios dentro de este directorio:

```
$ git add -A .
```

Nota `-A` (abreviatura de "all") significa que `git` también reconocerá si has eliminado archivos (por defecto, sólo reconoce archivos nuevos/modificados). También recuerda (del capítulo 3) que `.` significa el directorio actual.

Antes de que subamos todos los archivos, vamos a ver qué es lo que `git` subirá (todos los archivos que `git` cargará deberían aparecer en verde):

```
$ git status
```

Ya casi estamos, ahora es tiempo de decirle que guarde este cambio en su historial. Vamos a darle un "mensaje de commit" donde describimos lo que hemos cambiado. Puedes escribir cualquier cosa que te gustaría en esta etapa, pero es útil escribir algo descriptivo para que puedes recordar lo que has hecho en el futuro.

```
$ git commit -m "Cambie el HTML para la página."
```

Nota Asegúrate de usar comillas dobles alrededor del mensaje de commit.

Una vez que hicimos esto, subimos (push) nuestros cambios a PythonAnywhere:

```
git push
```

Descarga tu nuevo código a PythonAnywhere y actualiza tu aplicación web

- Abre la [página de consolas de PythonAnywhere](#) y ve a tu **consola Bash** (o comienza una nueva). Luego, ejecuta:

```
$ cd ~/my-first-blog $ git pull [...]
```

Y mira cómo tu código se descarga. Si quieres comprobar que ya ha terminado, puedes ir a la pestaña **Files** y ver tu código en PythonAnywhere.

- Finalmente, dirígete a la pestaña [Web](#) y selecciona **Reload** en tu aplicación web.

¡Tu actualización debería estar en línea! Actualiza tu sitio web en el navegador. Ahora deberías poder ver tus cambios :)

ORM de Django y QuerySets

En este capítulo aprenderás cómo Django se conecta a la base de datos y almacena los datos en ella. ¡Vamos a sumergirnos!

¿Qué es un QuerySet?

Un QuerySet es, en esencia, una lista de objetos de un modelo determinado. Un QuerySet te permite leer los datos de base de datos, filtrarlos y ordenarlos.

Es más fácil de aprender con ejemplos. Vamos a intentarlo, ¿de acuerdo?

Django shell

Abre la consola y escribe este comando:

```
(myenv) ~/djangogirls$ python manage.py shell
```

El resultado debería ser:

```
(InteractiveConsole)
>>>
```

Ahora estás en la consola interactiva de Django. Es como la consola de Python, pero con un toque de magia Django :). Puedes utilizar todos los comandos Python aquí también, por supuesto.

Ver todos los objetos

Vamos a mostrar todos nuestros posts primero. Puedes hacerlo con el siguiente comando:

```
>>> Post.objects.all()
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'Post' is not defined
```

¡Uy! Apareció un error. Nos dice que no hay ningún objeto Post. Esto es correcto, ¡nos olvidamos de importarlo primero!

```
>>> from blog.models import Post
```

Esto es simple: importamos el modelo `Post` de `blog.models`. Vamos a intentar mostrar todos los posts nuevamente:

```
>>> Post.objects.all()
[<Post: my post title>, <Post: another post title>]
```

Esta es una lista de las posts creadas anteriormente. Hemos creado estos posts usando la interfaz del administrador de Django. Sin embargo, ahora queremos crear nuevos posts usando Python, ¿cómo lo hacemos?

Crear objetos

Esta es la forma de crear un nuevo objeto `Post` en la base de datos:

```
>>> Post.objects.create(author=me, title='Sample title', text='Test')
```

Pero hay un ingrediente faltante: `me`. Necesitamos pasar una instancia del modelo `User` como autor. ¿Cómo hacemos eso?

Primero importemos el modelo `User`:

```
>>> from django.contrib.auth.models import User
```

¿Qué usuarios tenemos en nuestra base de datos? Veamos:

```
>>> User.objects.all()
[<User: ola>]
```

Este es el super usuario que creamos anteriormente, Vamos a obtener una instancia de ese usuario ahora:

```
me = User.objects.get(username='ola')
```

Como puedes ver, hicimos un `get` de un `User` con el `username` que sea igual a 'ola'. ¡Genial! Acuérdate de poner tu nombre de usuario para obtener tu usuario.

Ahora finalmente podemos crear nuestro primer post:

```
>>> Post.objects.create(author=me, title='Sample title', text='Test')
```

¡Hurra! ¿Quieres probar si funcionó?

```
>>> Post.objects.all()
[<Post: Sample title>]
```

Agrega más posts

Ahora puedes divertirte un poco y añadir más posts para ver cómo funciona. Añade 2 ó 3 más y avanza a la siguiente parte.

Filtrar objetos

Una parte importante de los QuerySets es la habilidad para filtrarlos. Digamos que queremos encontrar todos los posts cuyo autor es el User ola. Usaremos `filter` en vez de `all` en `Post.objects.all()`. En los paréntesis estableceremos qué condición o condiciones deben cumplirse por un post del blog para terminar en nuestro queryset. En nuestro caso sería `author` es igual a `me`. La forma de escribirlo en Django es: `author=me`. Ahora nuestro bloque de código se ve como esto:

```
>>> Post.objects.filter(author=me)
[<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>]
```

¿O tal vez querramos ver todos los posts que contengan la palabra 'title' en el campo `title`?

```
>>> Post.objects.filter(title__contains='title')
[<Post: Sample title>, <Post: 4th title of post>]
```

Nota Hay dos guiones bajos (`_`) entre `title` y `contains`. Django ORM utiliza esta sintaxis para separar los nombres de los campos ("title") y operaciones o filtros ("contains"). Si sólo utilizas un guión bajo, obtendrás un error como "FieldError: Cannot resolve keyword title_contains".

También puedes obtener una lista de todos los posts publicados. Lo hacemos filtrando los posts que tienen el campo `published_date` en el pasado:

```
from django.utils import timezone
Post.objects.filter(published_date__lte=timezone.now()) []
```

Desafortunadamente, ninguno de nuestros posts han sido publicados todavía. ¡Vamos a cambiar esto! Primero obtén una instancia de un post que querramos publicar:

```
>>> post = Post.objects.get(id=1)
```

¡Luego utiliza el método `publish` para publicarlo!

```
>>> post.publish()
```

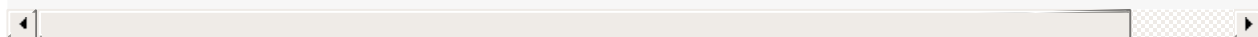
Ahora intenta obtener la lista de posts publicados nuevamente (presiona la tecla con la flecha hacia arriba 3 veces y presiona Enter):

```
>>> Post.objects.filter(published_date__lte=timezone.now())
[<Post: Sample title>]
```

Ordenando objetos

Los QuerySets también te permiten ordenar la lista de objetos. Intentemos ordenarlos por el campo `created_date`:

```
>>> Post.objects.order_by('created_date')
[<Post: Sample title>, <Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of p
```



También podemos invertir el ordenamiento agregando `-` al principio:

```
>>> Post.objects.order_by('-created_date')
[<Post: 4th title of post>, <Post: My 3rd post!>, <Post: Post number 2>, <Post: Sample t
```



¡Genial! ¡Ahora estás lista para la siguiente parte! Para cerrar la consola, tipea:

```
>>> exit()
$
```

Querysets de Django

Tenemos diferentes piezas en su lugar: el modelo `Post` está definido en `models.py`, tenemos a `post_list` en `views.py` y la plantilla agregada. ¿Pero cómo haremos realmente para que nuestros posts aparezcan en nuestra plantilla HTML? Porque eso es lo que queremos hacer: tomar algún contenido (modelos guardados en la base de datos) y mostrarlo adecuadamente en nuestra plantilla, ¿no?

Esto es exactamente lo que las *views* se supone que hacen: conectar modelos con plantillas. En nuestra *view* `post_list` necesitaremos tomar los modelos que deseamos mostrar y pasarlos a una plantilla. Así que básicamente en una *view* decidimos qué (modelo) se mostrará en una plantilla.

Muy bien, ahora ¿cómo lo hacemos?

Necesitamos abrir nuestro archivo `blog/views.py`. Hasta ahora la *view* `post_list` se ve así:

```
from django.shortcuts import render

def post_list(request):
    return render(request, 'blog/post_list.html', {})
```

¿Recuerdas cuando hablamos de incluir código en diferentes archivos? Ahora tenemos que incluir el modelo que definimos en el archivo `models.py`. Agregaremos la línea `from .models import Post` de la siguiente forma:

```
from django.shortcuts import render
from .models import Post
```

El punto después de `from` indica el *directorio actual* o la *aplicación actual*. Como `views.py` y `models.py` están en el mismo directorio, simplemente usamos `.` y el nombre del archivo (sin `.py`). Ahora importamos el nombre del modelo (`Post`).

¿Pero ahora qué sigue? Para tomar publicaciones reales del modelo `Post`, necesitamos algo llamado `querySet` (conjunto de consultas).

QuerySet

Ya debes estar familiarizada con la forma en que funcionan los QuerySets. Hablamos de ellos en el capítulo [Django ORM \(QuerySets\)](#).

Entonces ahora nos interesa obtener una lista de entradas del blog que han sido publicadas y ordenadas por `published_date` (fecha de publicación), ¿no? ¡Ya hicimos eso en el capítulo QuerySets!

```
Post.objects.filter(published_date__lte=timezone.now()).order_by('published_date')
```

Ahora pondremos este bloque de código en el archivo `blog/views.py`, agregándolo a la función `def post_list(request):`:

```
from django.shortcuts import render
from django.utils import timezone
from .models import Post

def post_list(request):
    posts = Post.objects.filter(published_date__lte=timezone.now()).order_by('publish
    return render(request, 'blog/post_list.html', {})
```

Observa que creamos una *variable* en nuestro QuerySet: `posts`. Tómalala como el nombre de nuestro QuerySet. De aquí en adelante vamos a referirnos al QuerySet con ese nombre.

La última parte es pasar el QuerySet `posts` a la plantilla (veremos cómo mostrarla en el siguiente capítulo).

En la función `render` ya tenemos el parámetro `request` (todo lo que recibimos del usuario via Internet) y el archivo `'blog/post_list.html'` como plantilla. El último parámetro, que se ve así: `{}` es un campo en el que podemos agregar algunas cosas para que la plantilla las use. Necesitamos nombrarlos (los seguiremos llamando `'posts'` por ahora :)). Se debería ver así: `{'posts': posts}`. Observa que la parte que va antes de `:` está entre comillas `' '`.

Finalmente nuestro archivo `blog/views.py` debería verse así:

```
from django.shortcuts import render
from django.utils import timezone
from .models import Post

def post_list(request):
    posts = Post.objects.filter(published_date__lte=timezone.now()).order_by('publish
    return render(request, 'blog/post_list.html', {'posts': posts})
```

¡Terminamos! Ahora regresemos a nuestra plantilla y mostremos este QuerySet.

Si quieres leer un poco más acerca de QuerySets en Django, puedes darle un vistazo a: <https://docs.djangoproject.com/en/1.8/ref/models/queries/>

Plantillas de Django

¡Es hora de mostrar algunos datos! Django nos provee las útiles **template tags** para ello.

¿Qué son las template tags?

Verás, en HTML no puedes realmente poner código Python, porque los navegadores no lo entienden. Ellos sólo saben HTML. Sabemos que HTML es algo estático, mientras que Python es mucho más dinámico.

Django template tags nos permiten transferir cosas de Python como cosas en HTML, así que tu puedes construir sitios web dinámicos más rápido y fácil.

Mostrar la plantilla post list

En el capítulo anterior dimos a nuestra plantilla una lista de posts en la variable `posts`. Ahora lo mostraremos en HTML.

Para imprimir una variable en una plantilla de Django, utilizamos llaves dobles con el nombre de la variable dentro, así:

```
{{ posts }}
```

Prueba esto en tu plantilla `blog/templates/blog/post_list.html` (reemplaza el segundo y el tercer par de etiquetas `<div></div>` con la línea `{{ posts }}`), guarda el archivo y actualiza la página para ver los resultados:



Como puedes ver, todo lo que obtenemos es esto:

```
[<Post: Mi segundo post>, <Post: Mi primer post>]
```

Esto significa que Django lo entiende como una lista de objetos. ¿Recuerdas de **Introducción a Python** cómo podemos mostrar listas? Sí, ¡con los ciclos for! En una plantilla de Django, lo haces de esta manera:

```
{% for post in posts %}
    {{ post }}
{% endfor %}
```

Prueba esto en tu plantilla.

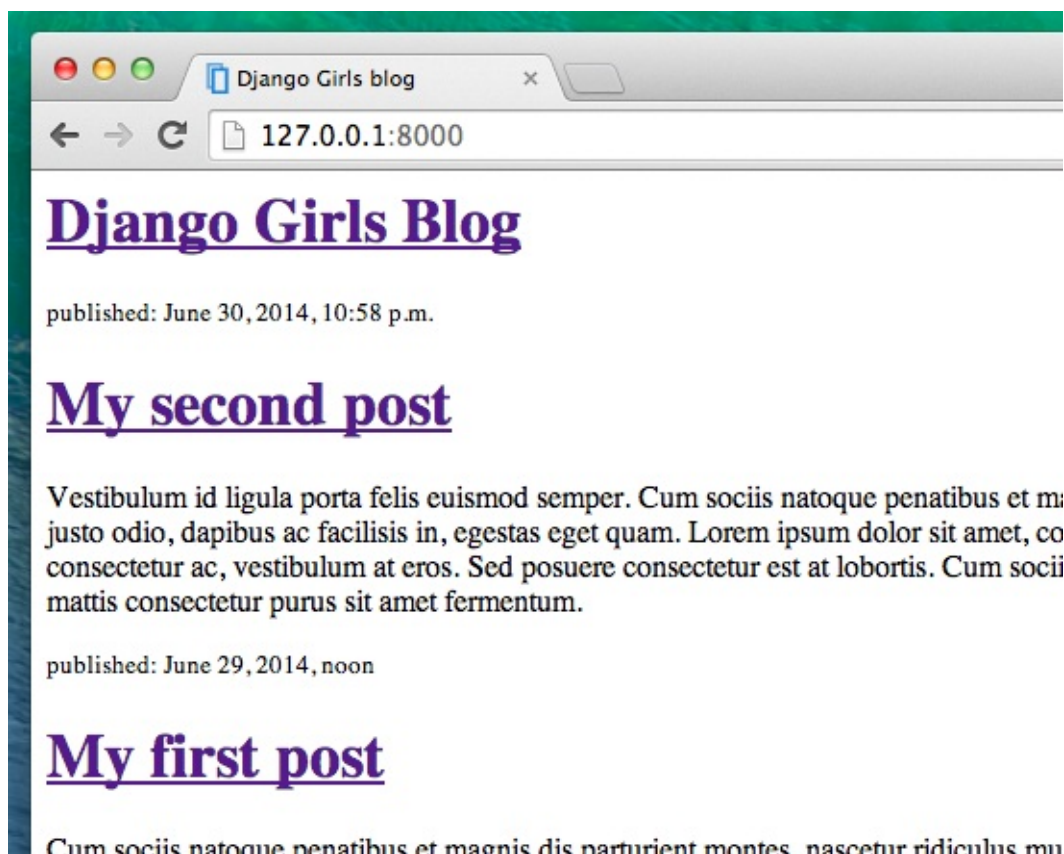


¡Funciona! Pero queremos que se muestren como los posts estáticos que creamos anteriormente en el capítulo de **Introducción a HTML**. Puedes mezclar HTML y template tags. Nuestro `body` se verá así:

```
<div>
    <h1><a href="/">Django Girls Blog</a></h1>
</div>

{% for post in posts %}
    <div>
        <p>published: {{ post.published_date }}</p>
        <h1><a href="">{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaks }}</p>
    </div>
{% endfor %}
```

Todo lo que pones entre `{% for %}` y `{% endfor %}` se repetirá para cada objeto en la lista. Actualiza tu página:



¿Has notado que utilizamos una notación diferente esta vez `{{ post.title }}` o `{{ post.text }}` ? Estamos accediendo a datos en cada uno de los campos definidos en nuestro modelo `Post` . También el `|linebreaks` está dirigiendo el texto de los posts a través de un filtro para convertir saltos de línea en párrafos.

Una cosa más

Sería bueno ver si tu sitio web seguirá funcionando en la Internet pública, ¿verdad? Intentemos desplegarla en PythonAnywhere nuevamente. Aquí te dejamos un ayuda memoria...

- Primero, sube tu código a GitHub

```
$ git status [...] $ git add -A . $ git status [...] $ git commit -m "Added views to create/edit blog post inside the site." [...] $ git push
```

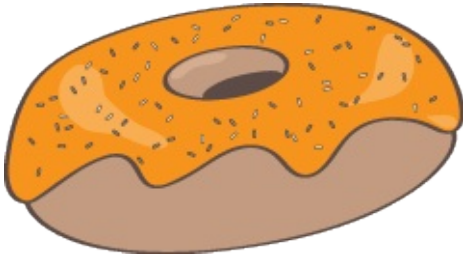
- Luego, identifícate en [PythonAnywhere](#) y ve a tu **consola Bash** (o empieza una nueva), y ejecuta:

```
$ cd my-first-blog $ git pull [...]
```

- Finalmente, ve a la [pestaña Web](#) y presiona **Reload** en tu aplicación web. ¡Tu actualización debería poder verse!

¡Felicidades! Ahora sigue adelante, trata de agregar un nuevo post usando el panel de administrador de Django (¡recuerda añadir `published_date!`) y luego actualiza tu página para ver si aparece tu nuevo post.

¿Funciona como un encanto? ¡Estamos orgullosos! Aléjate de tu computadora por un rato, te has ganado un descanso. :)



CSS - ¡Hazlo bonito!

Nuestro blog todavía se ve bastante feo, ¿verdad? ¡Es hora de hacerlo bonito! Vamos a usar CSS para eso.

¿Qué es CSS?

CSS (Cascading Style Sheets, que significa 'hojas de estilo en cascada') es un lenguaje utilizado para describir el aspecto y el formato de un sitio web escrito en lenguaje de marcado (como HTML). Trátalo como maquillaje para nuestra página web ;).

Pero no queremos empezar de cero otra vez, ¿verdad? Una vez más, usaremos algo que ya ha sido realizado por programadores y publicado en Internet de forma gratuita. Ya sabes, reinventar la rueda no es divertido.

¡Vamos a usar Bootstrap!

Bootstrap es uno de los frameworks HTML y CSS más populares para desarrollar webs bonitas: <http://getbootstrap.com/>

Lo escribieron programadores que trabajaban para Twitter y ahora lo desarrollan voluntarios de todo el mundo.

Instalar Bootstrap

Para instalar Bootstrap tienes que añadir esto al `<head>` de tu fichero `.html` (`blog/templates/blog/post_list.html`):

```
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
```

Esto no añade ningún fichero a tu proyecto. Simplemente apunta a ficheros que existen en Internet. Adelante, abre tu sitio web y actualiza la página. ¡Aquí está!



¡Se ve mucho mejor!

Ficheros estáticos en Django

Finalmente nos vamos a fijar en estas cosas que hemos estado llamando **ficheros estáticos**. Los ficheros estáticos son todos tus CSS e imágenes; ficheros que no son dinámicos, por lo que su contenido no depende del contexto de la petición y serán iguales para todos los usuarios.

Dónde poner los ficheros estáticos para Django

Como has visto cuando hemos ejecutado `collectstatic` en el servidor, Django ya sabe dónde encontrar los ficheros estáticos para la aplicación "admin". Ahora necesitamos añadir algunos ficheros estáticos para nuestra propia aplicación, `blog`.

Esto lo conseguimos creando una carpeta llamada `static` dentro de la aplicación `blog`:

```
djangoirls
├── blog
│   ├── migrations
│   └── static
└── mysite
```

Django encontrará automáticamente cualquier carpeta que se llame "static" dentro de las carpetas de tus aplicaciones y podrá utilizar su contenido como ficheros estáticos.

¡Tu primer fichero CSS!

Creemos un fichero CSS ahora, para añadir tu propio estilo a tu página web. Crear un

nuevo directorio llamado `css` dentro de tu directorio `static` . Después crea un nuevo fichero llamado `blog.css` dentro de este directorio `css` . ¿Lista?

```

djangogirls
├── blog
│   └── static
│       └── css
│           └── blog.css

```

¡Es hora de escribir algo de CSS! Abre el fichero `blog/static/css/blog.css` en tu editor de código.

No vamos a entrar mucho en la personalización y el aprendizaje de CSS aquí porque es bastante fácil y lo puedes aprender por tu cuenta después de este taller. Recomendamos enormemente hacer este [curso de HTML y CSS en Codecademy](#) para aprender todo lo que necesitas saber sobre cómo hacer tus sitios web más bonitos con CSS.

Pero vamos a hacer un poco al menos. ¿Tal vez podríamos cambiar el color de nuestro título? Los ordenadores utilizan códigos especiales para entender los colores. Empiezan con `#` y les siguen 6 letras (A-F) y números (0-9). Puedes encontrar códigos de color, por ejemplo, aquí: <http://www.colorpicker.com/>. También puedes utilizar [colores predefinidos](#) utilizando su nombre en inglés, como `red` y `green` .

En tu fichero `blog/static/css/blog.css` deberías añadir el siguiente código:

```

h1 a {
    color: #FCA205;
}

```

`h1 a` es un selector CSS. Quiere decir que estamos aplicando nuestros estilos a cualquier elemento `a` que se encuentre dentro de un elemento `h1` (por ejemplo cuando tenemos en código algo como: `<h1>enlace</h1>`). En este caso le estamos diciendo que cambie el color a `#FCA205` , que es naranja. Por supuesto, ¡puedes poner tu propio color aquí!

En un fichero CSS definimos los estilos para los elementos del fichero HTML. Los elementos se identifican por el nombre del elemento (es decir, `a` , `h1` , `body`), el atributo `class` (clase) o el atributo `id` (identificador). Class e id son nombres que le asignas tú misma al elemento. Las clases definen grupos de elementos y los ids apuntan a elementos específicos. Por ejemplo, la siguiente etiqueta se puede identificar con CSS usando el nombre `a` , la clase `external_link` o el id `link_to_wiki_page` :

```

<a href="http://en.wikipedia.org/wiki/Django" class="external_link" id="link_to_wiki_

```




Puedes leer más sobre [selectores de CSS en w3schools](#).

También necesitamos decirle a nuestra plantilla HTML que hemos añadido CSS. Abre el fichero `blog/templates/blog/post_list.html` y añade esta línea justo al principio:

```
{% load staticfiles %}
```

Aquí sólo estamos cargando ficheros estáticos :). Luego, entre el `<head>` y `</head>`, después de los enlaces a los ficheros CSS de Bootstrap (el navegador lee los ficheros en el orden en que están, así que nuestro fichero podría sobrescribir partes del código de Bootstrap), añade la siguiente línea:

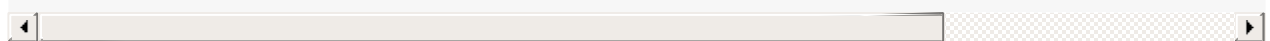
```
<link rel="stylesheet" href="{% static 'css/blog.css' %}">
```

Le acabamos de decir a nuestra plantilla dónde se encuentra nuestro fichero CSS.

Ahora tu fichero debería tener este aspecto:

```
{% load staticfiles %}
<html>
  <head>
    <title>Django Girls blog</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bo
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bo
    <link rel="stylesheet" href="{% static 'css/blog.css' %}">
  </head>
  <body>
    <div>
      <h1><a href="/">Django Girls Blog</a></h1>
    </div>

    {% for post in posts %}
      <div>
        <p>published: {{ post.published_date }}</p>
        <h1><a href="">{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaks }}</p>
      </div>
    {% endfor %}
  </body>
</html>
```



De acuerdo, ¡guarda el fichero y actualiza el sitio!



¡Buen trabajo! ¿Quizá nos gustaría darle un poco de aire a nuestro sitio web y aumentar también el margen en el lado izquierdo? ¡Vamos a intentarlo!

```
body {  
  padding-left: 15px;  
}
```

Añade esto a tu CSS, guarda el fichero y ¡mira cómo funciona!



¿Quizá podríamos personalizar la tipografía del título? Pega esto en la sección `<head>` del fichero `blog/templates/blog/post_list.html`:

```
<link href="http://fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext" re
```

Esta línea va a importar una tipografía llamada *Lobster* de Google Fonts (<https://www.google.com/fonts>).

Ahora añade la línea `font-family: 'Lobster';` en el fichero CSS `blog/static/css/blog.css` dentro del bloque de declaración `h1 a` (el código entre llaves `{` y `}`) y actualiza la página:

```
h1 a {  
    color: #FCA205;  
    font-family: 'Lobster';  
}
```



¡Genial!

Como se mencionó anteriormente, CSS tiene un concepto de clases que básicamente permite nombrar una parte del código HTML y aplicar estilos sólo a esta parte, sin afectar a otras. Es muy útil si tienes dos divs que hacen algo muy diferente (como el encabezado y la entrada) y no quieres que tengan el mismo aspecto.

¡Adelante! Nombra algunas partes del código HTML. Añade una clase llamada `page-header` al `div` que contiene el encabezado, así:

```
<div class="page-header">  
    <h1><a href="/">Django Girls Blog</a></h1>  
</div>
```

Y ahora añade la clase `post` al `div` que contiene una entrada del blog.

```
<div class="post">  
    <p>published: {{ post.published_date }}</p>  
    <h1><a href="">{{ post.title }}</a></h1>
```

```
<p>{{ post.text|linebreaks }}</p>
</div>
```

Ahora añadiremos bloques de declaración a diferentes selectores. Los selectores que comienzan con `.` hacen referencia a las clases. Hay muchos tutoriales y explicaciones sobre CSS en la web que te ayudarán a entender el siguiente código. Por ahora, simplemente copia y pega este bloque de código en tu fichero `blog/static/css/blog.css`:

```
.page-header {
    background-color: #ff9400;
    margin-top: 0;
    padding: 20px 20px 20px 40px;
}

.page-header h1, .page-header h1 a, .page-header h1 a:visited, .page-header h1 a:active {
    color: #ffffff;
    font-size: 36pt;
    text-decoration: none;
}

.content {
    margin-left: 40px;
}

h1, h2, h3, h4 {
    font-family: 'Lobster', cursive;
}

.date {
    float: right;
    color: #828282;
}

.save {
    float: right;
}

.post-form textarea, .post-form input {
    width: 100%;
}

.top-menu, .top-menu:hover, .top-menu:visited {
    color: #ffffff;
    float: right;
    font-size: 26pt;
    margin-right: 20px;
}

.post {
    margin-bottom: 70px;
}

.post h1 a, .post h1 a:visited {
    color: #000000;
}
```

Luego rodea el código HTML que muestra las entradas con las declaraciones de clases. Sustituye esto:

```
{% for post in posts %}
    <div class="post">
        <p>published: {{ post.published_date }}</p>
        <h1><a href="">{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaks }}</p>
    </div>
{% endfor %}
```

en `blog/templates/blog/post_list.html` por esto:

```
<div class="content container">
    <div class="row">
        <div class="col-md-8">
            {% for post in posts %}
                <div class="post">
                    <div class="date">
                        {{ post.published_date }}
                    </div>
                    <h1><a href="">{{ post.title }}</a></h1>
                    <p>{{ post.text|linebreaks }}</p>
                </div>
            {% endfor %}
        </div>
    </div>
</div>
```

Guarda los ficheros y actualiza tu sitio.



¡Bien! Se ve increíble, ¿verdad? En realidad el código que acabamos de pegar no es tan difícil de entender y deberías ser capaz de entender la mayoría sólo con leerlo.

No tengas miedo de jugar un poco con este CSS e intentar cambiar algunas cosas. Si rompes algo, no te preocupes, ¡siempre puedes deshacerlo!

De cualquier forma, te volvemos a recomendar que hagas el [curso de HTML y CSS de Codecademy](#) como una tarea post-taller para que aprendas todo lo que necesitas saber para hacer tus sitios web más bonitos con CSS.

¿Lista para el siguiente capítulo?! :)

Extendiendo Plantillas

Otra cosa buena que Django tiene para tí es la **extensión de plantillas**. ¿Qué significa esto? Significa que puedes usar las mismas partes de tu HTML para diferentes páginas de tu sitio web.

De esta forma no tienes que repetir el código en cada uno de los archivos cuando quieres usar una misma información o un mismo esquema. Y si quieres cambiar algo, no necesitas hacerlo en cada plantilla.

Creando una plantilla base

Una plantilla base es la plantilla más básica que extiendes en cada página de tu sitio web.

Vamos a crear un archivo `base.html` en `blog/templates/blog/`:

```
blog
├── templates
│   └── blog
│       ├── base.html
│       └── post_list.html
```

Luego ábrelo y copia todo lo que hay en `post_list.html` al archivo `base.html`, de la siguiente manera:

```
{% load staticfiles %}
<html>
  <head>
    <title>Django Girls blog</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bo
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bo
    <link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext"
    <link rel="stylesheet" href="{% static "css/blog.css" %}">
  </head>
  <body>
    <div class="page-header">
      <h1><a href="/">Django Girls Blog</a></h1>
    </div>

    <div class="content container">
      <div class="row">
        <div class="col-md-8">
          {% for post in posts %}
            <div class="post">
              <div class="date">
                {{ post.published_date }}
```

```

        </div>
        <h1><a href="">{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaks }}</p>
    </div>
    {% endfor %}
</div>
</div>
</div>
</body>
</html>

```

Luego, en `base.html` reemplaza por completo tu `<body>` (todo lo que haya entre `<body>` and `</body>`) con esto:

```

<body>
    <div class="page-header">
        <h1><a href="/">Django Girls Blog</a></h1>
    </div>
    <div class="content container">
        <div class="row">
            <div class="col-md-8">
                {% block content %}
                {% endblock %}
            </div>
        </div>
    </div>
</body>

```

Básicamente reemplazamos todo entre `{% for post in posts %}{% endfor %}` con:

```

{% block content %}
{% endblock %}

```

¿Qué significa esto? Acabas de crear un `block`, una template tag que te permite insertar HTML en este bloque en otras plantillas que extiendan a `base.html`. Te mostraremos como hacer esto en un momento.

Ahora guárdalo y abre tu archivo `blog/templates/blog/post_list.html` de nuevo. Elimina todo lo que no esté dentro del `body` y luego elimina también `<div class="page-header">` `</div>`, de forma que tu archivo se verá así:

```

{% for post in posts %}
    <div class="post">
        <div class="date">
            {{ post.published_date }}
        </div>
        <h1><a href="">{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaks }}</p>
    </div>

```



```
{% endfor %}
```

Y ahora agrega esta línea al inicio del archivo:

```
{% extends "blog/base.html" %}
```

Significa que ahora estamos extendiendo de la plantilla `base.html` en `post_list.html`. Sólo nos falta una cosa: poner todo (excepto la línea que acabamos de agregar) entre `{% block content %}` y `{% endblock content %}`. Como esto:

```
{% extends "blog/base.html" %}

{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h1><a href="">{{ post.title }}</a></h1>
            <p>{{ post.text|linebreaks }}</p>
        </div>
    {% endfor %}
{% endblock content %}
```

¡Eso es todo! Verifica que tu sitio web aún funcione apropiadamente :)

Si tienes un error `TemplateDoesNotExist` que diga que no hay un archivo `blog/base.html` y tienes `runserver` ejecutándose en la consola, intenta pararlo (presionando Ctrl+C - las teclas Control y C juntas) y reinicialo ejecutando el comando `python manage.py runserver`.

Amplía tu aplicación

Ya hemos completado todos los pasos necesarios para la creación de nuestro sitio web: sabemos cómo escribir un model, url, view y template. También sabemos cómo hacer que nuestro sitio web se vea lindo.

¡Hora de practicar!

Lo primero que necesitamos en nuestro blog es, obviamente, una página para mostrar un post, ¿cierto?

Ya tenemos un modelo `Post`, así que no necesitamos añadir nada a `models.py`.

Crea un enlace en la plantilla

Vamos a empezar añadiendo un enlace dentro del archivo

`blog/templates/blog/post_list.html`. Hasta el momento debería verse así:

```
{% extends "blog/base.html" %}

{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h1><a href="">{{ post.title }}</a></h1>
            <p>{{ post.text|linebreaks }}</p>
        </div>
    {% endfor %}
{% endblock content %}
```

Queremos tener un enlace a una página de detalle sobre el título del post. Vamos a cambiar

`<h1>{{ post.title }}</h1>` dentro del enlace:

```
<h1><a href="{% url 'blog.views.post_detail' pk=post.pk %}">{{ post.title }}</a></h1>
```

Es hora de explicar el misterioso `{% url 'blog.views.post_detail' pk=post.pk %}`. Como probablemente sospeches, la notación `{% %}` significa que estamos utilizando Django template tags. ¡Esta vez vamos a utilizar uno que va a crear una dirección URL para nosotros!

`blog.views.post_detail` es una ruta hacia `post_detail` *view* que queremos crear. Por favor nota: `blog` es el nombre de nuestra aplicación (el `blog` de directorio), `views` es el nombre del archivo `views.py` y `post_detail` es el nombre de la *view*.

Ahora cuando vayamos a: <http://127.0.0.1:8000/> tendremos un error (como era de esperar, ya que no tenemos una dirección URL o una *view* para `post_detail`). Se verá así:



Vamos a crear una URL en `urls.py` para nuestro *view* `post_detail` !

URL: <http://127.0.0.1:8000/post/1/>

Queremos crear una URL que apunte a Django a una *view* denominada `post_detail`, que mostrará una entrada del blog. Agrega la línea `url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail)`, al archivo `blog/urls.py`. Debería tener este aspecto:

```
`` from django.conf.urls import include, url from . import views
```

```
urlpatterns = [
    url(r'^$', views.post_list),
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail),
]
```

Ese da un poco de miedo, pero no te preocupes - lo explicaremos para ti: comienza con `^^`

Eso significa que si entras en `http://127.0.0.1:8000/post/5/` en tu navegador, Django en

`pk` es la abreviación de `primary key`. Este nombre se utiliza a menudo en proyectos de

¡Bien! ¡Actualiza la página: `http://127.0.0.1:8000/` ¡Boom! ¡Sin embargo vemos otro error!

```
![AttributeError][2]
```

```
[2]: images/attribute_error2.png
```

¿Te acuerdas del próximo paso? Por supuesto: ¡agregar una *view*!

```
## post_detail view
```

Esta vez nuestra *view* tomará un parámetro adicional `pk`. ¿Nuestra *view* necesita reci

Ahora, queremos sólo un post del blog. Para ello podemos usar `querysets` como este:

```
Post.objects.get(pk=pk)
```

Pero este código tiene un problema. Si no hay ningún `Post` con `llave primaria` (`pk`) t

```
![DoesNotExist error][3]
```

```
[3]: images/does_not_exist2.png
```

¡No queremos eso! Pero, por supuesto, Django viene con algo que se encargará de ese problema.

```
![Page not found][4]
```

```
[4]: images/404_2.png
```

La buena noticia es que puedes crear tu propia página `Page Not Found` y diseñarla como quieras.

¡Es hora de agregar una *view* a nuestro archivo `views.py`!

Deberíamos abrir `blog/views.py` y agregar el siguiente código:

```
from django.shortcuts import render, get_object_or_404
```

Cerca de otras líneas `from`. Y en el final del archivo añadimos nuestra *view*:

```
def post_detail(request, pk):
    post = get_object_or_404(Post, pk=pk)
    return render(request, 'blog/post_detail.html', {'post': post})
```

Sí. Es hora de actualizar la página: <http://127.0.0.1:8000/>

¡Funcionó! Pero ¿qué pasa cuando haces click en un enlace en el título del post?

¡Oh no! ¡Otro error! Pero ya sabemos cómo lidiar con eso, ¿no? ¡Tenemos que añadir una plantilla!

Crearemos un archivo en `blog/templates/blog` llamado `post_detail.html`.

Se verá así:

```
{% extends "blog/base.html" %}

{% block content %}
    <div class="post">
        {% if post.published_date %}
            <div class="date">
                {{ post.published_date }}
            </div>
        {% endif %}
        <h1>{{ post.title }}</h1>
        <p>{{ post.text|linebreaks }}</p>
    </div>
{% endblock %}
```

Una vez más estamos extendiendo `base.html`. En el bloque `content` queremos mostrar la fecha de publicación (si existe), título y texto de nuestros posts. Pero deberíamos discutir algunas cosas importantes, ¿cierto?

`{% if ... %} ... {% endif %}` es un template tag que podemos usar cuando querramos ver algo (¿recuerdas `if ... else...` del capítulo de **Introducción a Python?**). En este escenario queremos comprobar si el campo `published_date` de un post no está vacío.

Bien, podemos actualizar nuestra página y ver si `Page Not Found` se ha ido.



¡Yay! ¡Funciona!

Una cosa más: ¡Tiempo de implementación!

Sería bueno verificar que tu sitio web aún funcionará en PythonAnywhere, ¿cierto? Intentemos desplegar de nuevo.

```
$ git status
$ git add -A .
$ git status
$ git commit -m "Added views to create/edit blog post inside the site."
$ git push
```

- Luego, en una [consola Bash de PythonAnywhere](#)

```
$ cd my-first-blog $ git pull [...]
```

- Finalmente, ve a la pestaña [Web](#) y haz click en **Reload**.

¡Y eso debería ser todo! Felicidades :)

Formularios en Django

Lo último que haremos en nuestro website es crear un apartado para agregar y editar posts en el blog. Django `admin` está bien, pero es bastante difícil de personalizar y hacerlo bonito. Con `forms` tendremos un poder absoluto sobre nuestra interfaz - podemos hacer casi cualquier cosa que podamos imaginar!

Lo bueno de Django forms es que podemos definirlo desde cero o creando un `ModelForm` y se guardará el resultado del formulario en el modelo.

Esto es exactamente lo que queremos hacer: crearemos un formulario para nuestro modelo `Post`.

Como cada parte importante de Django, forms tienen su propio archivo: `forms.py`.

Tenemos que crear un archivo con este nombre en el directorio `blog`.

```
blog
└─ forms.py
```

Ok, vamos abrirlo y vamos a escribir el siguiente código:

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

    class Meta:
        model = Post
        fields = ('title', 'text',)
```

Primero necesitamos importar Django forms (`from django import forms`) y, obviamente, nuestro modelo `Post` (`from .models import Post`).

`PostForm`, como probablemente sospechas, es el nombre del formulario. Necesitamos decirle a Django que este formulario es un `ModelForm` (así Django hará algo de magia para nosotros) - `forms.ModelForm` es responsable de ello.

A continuación, tenemos `class Meta`, donde le decimos a Django qué modelo debe utilizar para crear este formulario (`model = Post`).

Finalmente, podemos decir que campo(s) queremos en nuestro formulario. En este

escenario sólo queremos `title` y `text` - `author` será la persona que se ha autenticado (¡tú!) y `created_date` se definirá automáticamente cuando creamos un post (es decir en el código), ¿sí?

¡Y eso es todo! Todo lo que necesitamos hacer ahora es usar el formulario en una *view* y mostrarla en una plantilla.

Una vez más vamos a crear: un enlace a la página, una dirección URL, una vista y una plantilla.

Enlace a una página con el formulario

Es hora de abrir `blog/templates/blog/base.html`. Vamos a añadir un enlace en `div` llamado `page-header`:

```
<a href="{% url 'blog.views.post_new' %}" class="top-menu"><span class="glyphicon gly
```

Ten en cuenta que queremos llamar a nuestra nueva vista `post_new`.

Después de agregar la línea, tu archivo html debería tener este aspecto:

```
{% load staticfiles %}
<html>
  <head>
    <title>Django Girls blog</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bo
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bo
    <link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext'
    <link rel="stylesheet" href="{% static 'css/blog.css' %}">
  </head>
  <body>
    <div class="page-header">
      <a href="{% url 'blog.views.post_new' %}" class="top-menu"><span class="g
      <h1><a href="/">Django Girls Blog</a></h1>
    </div>
    <div class="content container">
      <div class="row">
        <div class="col-md-8">
          {% block content %}
          {% endblock %}
        </div>
      </div>
    </div>
  </body>
</html>
```

Luego de guardar y actualizar la página `http://127.0.0.1:8000` obviamente verás un error `NoReverseMatch` familiar, ¿verdad?

URL

Abrimos `blog/urls.py` y añadimos una línea:

```
url(r'^post/new/$', views.post_new, name='post_new'),
```

Y el código final tendrá este aspecto:

```
from django.conf.urls import include, url
from . import views

urlpatterns = [
    url(r'^$', views.post_list),
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail),
    url(r'^post/new/$', views.post_new, name='post_new'),
]
```

Después de actualizar el sitio, veremos un `AttributeError`, puesto que no tenemos la vista `post_new` implementada. Vamos a añadirla ahora.

Vista post_new

Es el momento de abrir el archivo `blog/views.py` y agregar las siguientes líneas al resto de las filas `from`:

```
from .forms import PostForm
```

y nuestra *vista*:

```
def post_new(request):
    form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

Para crear un nuevo formulario `Post`, tenemos que llamar a `PostForm()` y pasarlo a la plantilla. Volveremos a esta *vista* pero, por ahora, vamos a crear rápidamente una plantilla para el formulario.

Plantilla

Tenemos que crear un archivo `post_edit.html` en el directorio `blog/templates/blog`. Para hacer que un formulario funcione necesitamos varias cosas:

- tenemos que mostrar el formulario. Podemos hacerlo, por ejemplo, con un simple ``.
- la línea anterior tiene que estar dentro de una etiqueta de formulario HTML: `<form method="POST">...</form>`
- necesitamos un botón `Guardar`. Lo hacemos con un botón HTML: `<button type='submit'>Save</button>`
- y finalmente justo después de la apertura de `<form...>` necesitamos agregar `{% csrf_token %}`. ¡Esto es muy importante ya que hace que tus formularios sean seguros! Django se quejará si te olvidas de esta parte cuando intente guardar el formulario.

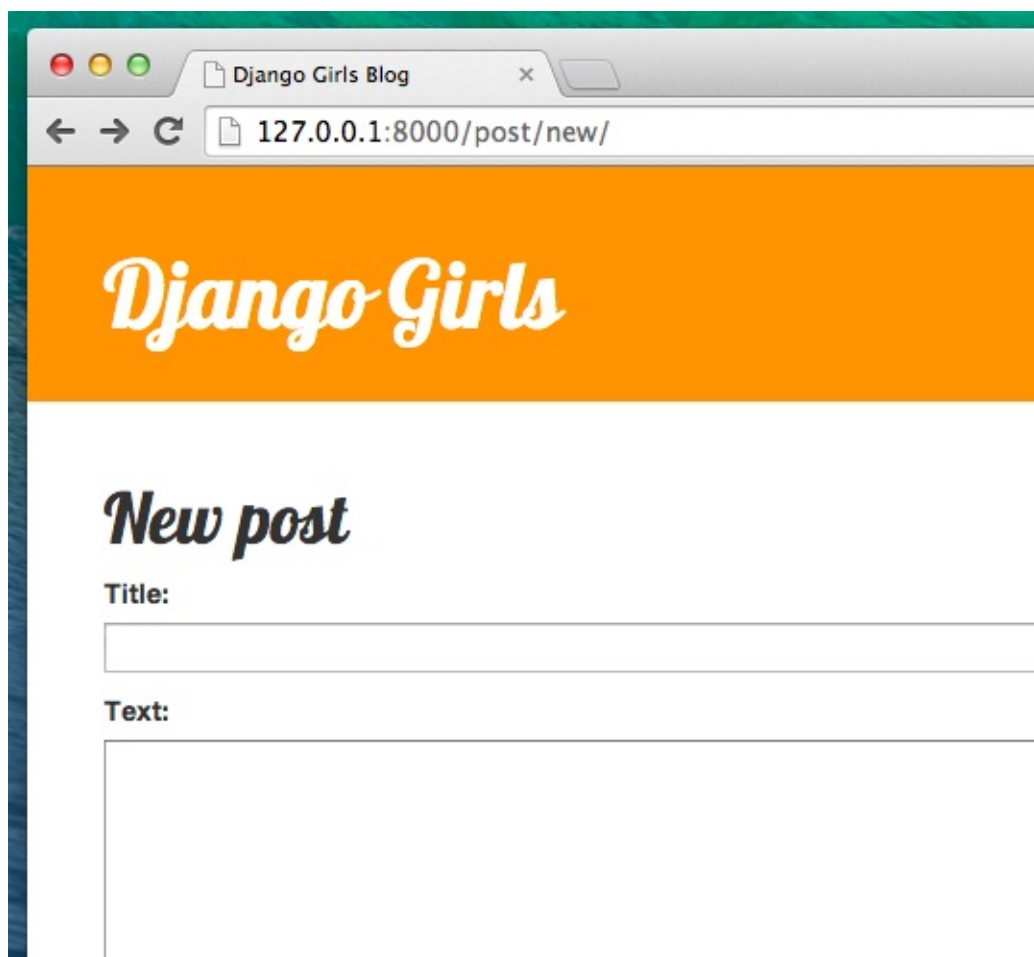


Bueno, vamos a ver cómo quedará el HTML en `post_edit.html`:

```
{% extends "blog/base.html" %}

{% block content %}
    <h1>New post</h1>
    <form method="POST" class="post-form">{% raw %}{% csrf_token %}{% endraw %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
    </form>
{% endblock %}
```

¡Es hora de actualizar! ¡Si! ¡Tu formulario se muestra!



Pero, ¡un momento! Si escribes algo en los campos `title` y `text` y tratas de guardar los cambios - ¿qué pasará?

¡Nada! Una vez más estamos en la misma página y el texto se ha ido... no se añade ningún post nuevo. Entonces, ¿qué ha ido mal?

La respuesta es: nada. Tenemos que trabajar un poco más en nuestra *vista*.

Guardar el formulario

Abre `blog/views.py` una vez más. Actualmente, lo que tenemos en la vista `post_new` es:

```
def post_new(request):
    form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

Cuando enviamos el formulario somos redirigidos a la misma vista, pero esta vez tenemos algunos datos adicionales en `request`, más específicamente en `request.POST` (el nombre no tiene nada que ver con un post del blog, se refiere a que estamos "publicando" -en inglés, posting- datos). ¿Recuerdas que en el archivo HTML la definición de `<form>` tenía la

variable `method="POST"` ? Todos los campos del formulario estan ahora en `request.POST` . No deberías renombrar la variable `POST` (el único nombre que también es válido para la variable `method` es `GET` , pero no tenemos tiempo para explicar cuál es la diferencia).

En nuestra *view* tenemos dos posibles situaciones a contemplar. Primero: cuando accedemos a la página por primera vez y queremos un formulario en blanco. Segundo: cuando volvemos a la *view* con los datos del formulario que acabamos de escribir. Así que tenemos que añadir una condición (utilizaremos `if` para eso).

```
if request.method == "POST":
    [...]
else:
    form = PostForm()
```

Es hora de llenar los puntos `[...]` . Si el `method` es `POST` queremos construir el `PostForm` con los datos del formulario, ¿no? Lo haremos con:

```
form = PostForm(request.POST)
```

Fácil! Lo siguiente es verificar si el formulario es correcto (todos los campos necesarios están definidos y no hay valores incorrectos). Lo hacemos con `form.is_valid()` .

Comprobamos que el formulario es válido y, si es así, ¡lo podemos salvar!

```
if form.is_valid():
    post = form.save(commit=False)
    post.author = request.user
    post.save()
```

Básicamente, tenemos que hacer dos cosas aquí: guardamos el formulario con `form.save` y añadimos un autor (ya que no había ningún campo de `author` en el `PostForm` y este campo es obligatorio). `commit=False` significa que no queremos guardar el modelo `Post` todavía - queremos añadir el autor primero. La mayoría de las veces utilizarás `form.save()` , sin `commit=False` , pero en este caso, tenemos que hacerlo. `post.save()` conservará los cambios (añadiendo a autor) y se creará una nuevo post en el blog.

Por último, sería genial si podemos inmediatamente ir a la página `post_detail` del nuevo post de blog, ¿no? Para hacerlo necesitamos importar algo más:

```
from django.shortcuts import redirect
```

Agrégalo al principio del archivo. Y ahora podemos decir: vé a la página `post_detail` del

post recién creado.

```
return redirect('blog.views.post_detail', pk=post.pk)
```

`blog.views.post_detail` es el nombre de la vista a la que queremos ir. ¿Recuerdas que esta *view* requiere una variable `pk` ? Para pasarlo a las vistas utilizamos `pk=post.pk` , donde `post` es el post recién creado.

Bien, hablamos mucho, pero probablemente queremos ver como se ve ahora la *vista*, ¿verdad?

```
def post_new(request):
    if request.method == "POST":
        form = PostForm(request.POST)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.save()
            return redirect('blog.views.post_detail', pk=post.pk)
    else:
        form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

Vamos a ver si funciona. Ve a la página <http://127.0.0.1:8000/post/new/>, añade un `title` y un `text` , guardalo... ¡y voilà! Se añade el nuevo post al blog y se nos redirige a la página de `post_detail` .

Probablemente has visto que no hemos definido la fecha de publicación. Vamos a introducir un *botón publicar* en **Django Girls Tutorial: Extensions**.

¡Eso es genial!

Validación de formularios

Ahora, vamos a enseñarte qué tan bueno es Django forms. Un post del blog debe tener los campos `title` y `text` . En nuestro modelo `Post` no dijimos (a diferencia de `published_date`) que estos campos son requeridos, así que Django, por defecto, espera que estén definidos.

Trata de guardar el formulario sin `title` y `text` . ¡Adivina qué pasará!

Django Girls Blog

127.0.0.1:8000/post/new/

Django Girls

New post

- This field is required.

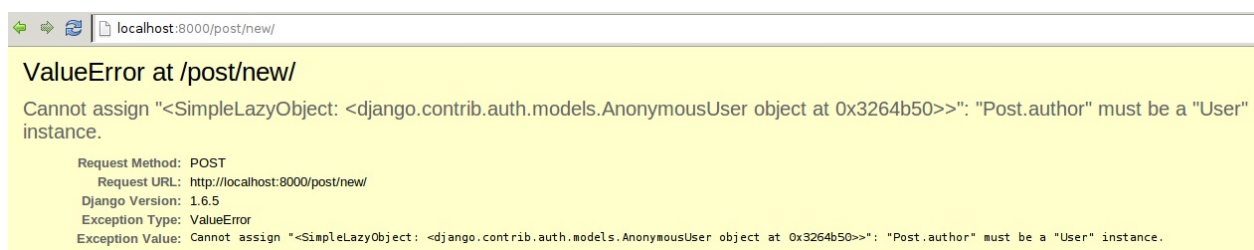
Title:

- This field is required.

Text:

Django se encarga de validar que todos los campos en el formulario estén correctos. ¿No es genial?

Como recientemente hemos utilizado la interfaz de administrador de Django, el sistema piensa que estamos conectadas. Hay algunas situaciones que podrían llevarnos a desconectarnos (cerrando el navegador, reiniciando la base de datos, etc.). Si estás recibiendo errores al crear un post que indican la falta de inicio de sesión de usuario, dirígete a la página de administración <http://127.0.0.1:8000/admin> e inicia sesión nuevamente. Esto resolverá el problema temporalmente. Hay un arreglo permanente esperándote en el capítulo **Tarea: ¡Añadir seguridad a tu sitio web!** después del tutorial principal.



Editar formulario

Ahora sabemos cómo agregar un nuevo formulario. Pero, ¿qué pasa si queremos editar uno existente? Es muy similar a lo que acabamos de hacer. Vamos a crear algunas cosas importantes rápidamente (si no entiendes algo, pregúntale a tu tutor o revisa los capítulos anteriores, son temas que ya hemos cubierto).

Abre el archivo `blog/templates/blog/post_detail.html` y añade esta línea:

```
<a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyph
```

para que la plantilla quede:

```
{% extends "blog/base.html" %}

{% block content %}
    <div class="date">
        {% if post.published_date %}
            {{ post.published_date }}
        {% endif %}
        <a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="g
    </div>
    <h1>{{ post.title }}</h1>
    <p>{{ post.text|linebreaks }}</p>
{% endblock %}
```

En el archivo `blog/urls.py` añadimos esta línea:

```
url(r'^post/(?P<pk>[0-9]+)/edit/$', views.post_edit, name='post_edit'),
```

Vamos a reusar la plantilla `blog/templates/blog/post_edit.html`, así que lo último que nos falta es una *view*.

Abramos el archivo `blog/views.py` y añadamos al final esta línea:

```
def post_edit(request, pk):
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = PostForm(request.POST, instance=post)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.save()
            return redirect('blog.views.post_detail', pk=post.pk)
    else:
```

```
form = PostForm(instance=post)
return render(request, 'blog/post_edit.html', {'form': form})
```

Esto se ve casi exactamente igual a nuestra view `post_new`, ¿no? Pero no del todo.

Primero: pasamos un parámetro extra `pk` de los urls. Luego: obtenemos el modelo `Post` que queremos editar con `get_object_or_404(Post, pk=pk)` y después, al crear el formulario pasamos este post como una `instancia` tanto al guardar el formulario:

```
form = PostForm(request.POST, instance=post)
```

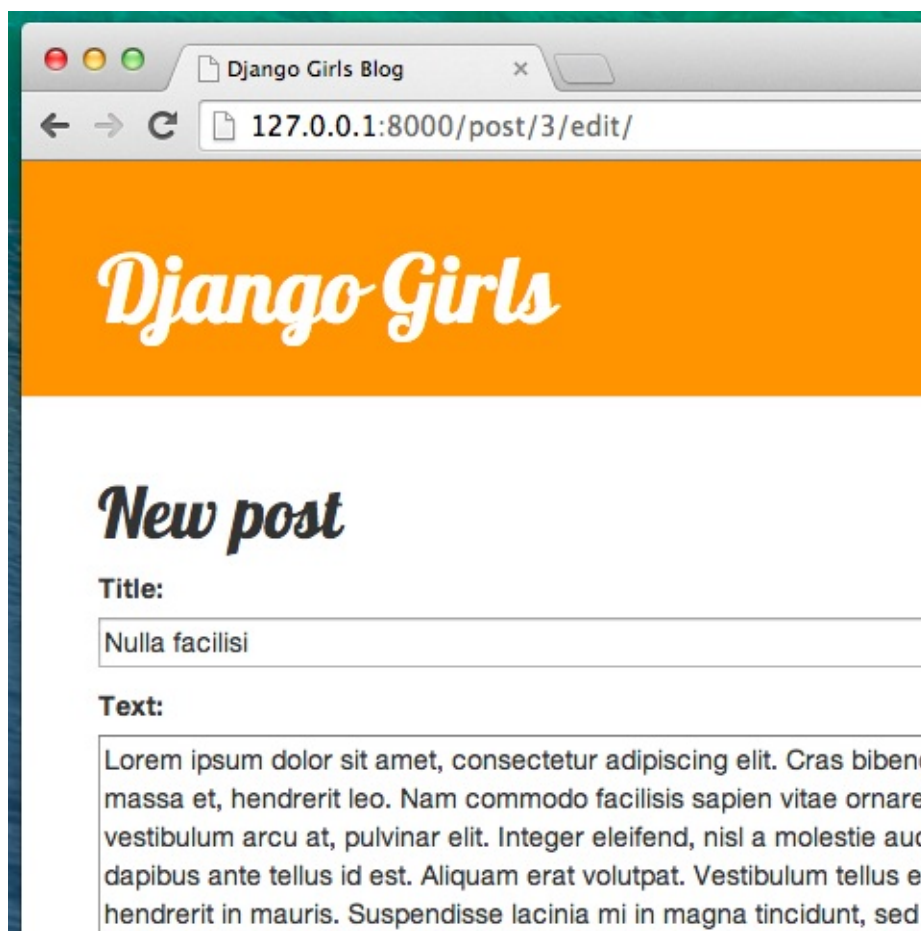
como al abrir un formulario con este post para editarlo:

```
form = PostForm(instance=post)
```

Ok, ¡vamos a probar si funciona! Dirígete a la página `post_detail`. Debe haber ahí un botón para editar en la esquina superior derecha:



Al dar click ahí, debes ver el formulario con nuestro post del blog:



¡Siéntete libre de cambiar el título o el texto y guarda los cambios!

¡Felicitaciones! ¡Tu aplicación está cada vez más completa!

Si necesitas más información sobre los formularios de Django, debes leer la documentación: <https://docs.djangoproject.com/en/1.8/topics/forms/>

Una cosa más: ¡Tiempo de implementación!

Veamos si todo esto funciona en PythonAnywhere. ¡Tiempo de hacer otro despliegue!

- Primero, haz un commit con tu nuevo código y súbelo a GitHub

```
$ git status $ git add -A $ git status $ git commit -m "Added views to create/edit blog post inside the site." $ git push
```

- Luego, en una [consola Bash de PythonAnywhere](#)

```
$ cd my-first-blog $ git pull [...]
```

- Finalmente, ve a la pestaña [Web](#) y haz click en **Reload**.

¡Y eso debería ser todo! Felicidades :)

Dominio

PythonAnywhere te ha dado un dominio gratuito, pero tal vez no quieras tener ".pythonanywhere.com" al final de la URL de tu blog. Quizás quiera que tu blog viva en "www.infinite-kitten-pictures.org" o "www.3d-printed-steam-engine-parts.com" o "www.antique-buttons.com" o "www.mutant-unicornz.net", o lo que quieras que sea.

Aquí hablaremos brevemente sobre cómo obtener un dominio y veremos cómo vincularlo a tu aplicación web en PythonAnywhere. Sin embargo, deberías saber que la mayoría de los dominios son pagos, y PythonAnywhere te cobrará un valor adicional para usar tu propio nombre de dominio -- no es demasiado dinero en total, pero es probablemente algo que quieras hacer sólo si estás muy comprometida con la causa.

¿Donde registrar un dominio?

Un dominio típico cuesta alrededor de 15 dólares estadounidenses anuales. Hay opciones más baratas y más caras, dependiendo del proveedor. Hay una gran cantidad de empresas a las que puedes comprar un dominio: una simple [búsqueda en google](#) dará cientos de opciones.

Nuestra opción favorita es [I want my name](#). Ellos se promocionan como una opción "indolora para el manejo de dominios" y realmente lo son.

También puedes obtener dominios gratuitos. [dot.tk](#) es un lugar para obtener uno, pero deberíamos advertirte que los dominios gratuitos a veces se sienten algo "baratos" -- si tu sitio va a ser un sitio para un negocio profesional, seguramente quieras considerar comprar un dominio "apropiado" que termine en `.com`.

Cómo apuntar tu dominio a PythonAnywhere

Si elegiste la opción de *iwantmyname.com*, haz click en `Domains` en el menú y elije tu nuevo dominio. Luego encuentra el vínculo a `manage DNS records`:



Ahora necesitas encontrar este formulario:



Y completarlo con los siguientes detalles: - Hostname: `www` - Type: `CNAME` - Value: tu

dominio de PythonAnywhere (por ejemplo, `djangogirls.pythonanywhere.com`) - TTL: 60



En la parte inferior, haz click en el botón "Agregar" y guarda los cambios.

Nota Si utilizaste un proveedor de dominios diferente, la interfaz exacta para encontrar tus configuraciones de DNS / CNAME será diferente, pero tu objetivo es el mismo: configurar un CNAME que apunte a tu nuevo dominio en

`yourusername.pythonanywhere.com`.

Puede tomar unos minutos para tu dominio para empezar a funcionar, ¡sé paciente!

Configura el dominio a través de la aplicación web en PythonAnywhere

También necesitarás decirle a PythonAnywhere que quieres usar tu dominio personalizado.

Ve a la página [PythonAnywhere Accounts](#) y haz una actualización del tipo de cuenta. La opción más económica (el plan "Hacker") está bien para empezar. Siempre puedes elegir un plan con mayores prestaciones después cuando te vuelvas super-famosa y tengas millones de visitas.

Luego, ve a la pestaña [Web](#) y anota un par de cosas:

- Copia la **ruta a tu virtualenv** y ponla en algún lugar seguro
- Abre tu **archivo de configuración WSGI**, copia el contenido, y pégalo en algún lugar seguro

Luego, haz click en **Delete** de tu vieja aplicación web. No te preocupes, esto no eliminará tu código, solamente cambiará el dominio `yourusername.pythonanywhere.com`. Luego, crea una nueva aplicación web y sigue estos pasos:

- Ingresa tu nombre de dominio
- Elige "configuración manual"
- Elige Python 3.4
- ¡Y listo!

Cuando seas redirigida a la pestaña web.

- Pega la ruta al virtualenv que guardaste anteriormente
- Abre tu nuevo archivo de configuración WSGI y pega el contenido de tu viejo archivo de configuración

Haz click en actualizar, ¡deberías ver que tu sitio está en línea en el nuevo dominio!

Si te encuentras con algún problema, haz click en el vínculo "Send feedback" en el sitio de PythonAnywhere, y uno de sus amigables administradores te contactará para ayudarte en breve.

¿Qué sigue?

¡Date muchas felicitaciones! ¡Eres increíble!. ¡Estamos orgullosos! <3

¿Qué hacer ahora?

Toma un descanso y relájate. Acabas de hacer algo realmente grande.

Después de eso, asegúrate de:

- Seguir a Django Girls en [Facebook](#) o [Twitter](#) para estar al día

¿Me puedes recomendar recursos adicionales?

¡Sí! En primer lugar, sigue adelante y prueba nuestro libro llamado [Django Girls Tutorial: Extensiones](#).

Más adelante, puedes intentar los recursos listados a continuación. ¡Están todos muy recomendados!

- [Django's official tutorial](#)
- [New Coder tutorials](#)
- [Code Academy Python course](#)
- [Code Academy HTML & CSS course](#)
- [Django Carrots tutorial](#)
- [Learn Python The Hard Way book](#)
- [Getting Started With Django video lessons](#)
- [Two Scoops of Django: Best Practices for Django book](#)