
Table of Contents

Introducción	1.1
¿Cómo funciona Internet?	1.2
Introducción a la Línea de comandos	1.3
Instalación de Python	1.4
Editor de código	1.5
Introducción a Python	1.6
¿Qué es Django?	1.7
Instalación de Django	1.8
¡Tu primer proyecto en Django!	1.9
Modelos en Django	1.10
Administrador de Django	1.11
¡Desplegar!	1.12
URLs en Django	1.13
Vistas de Django - ¡Es hora de crear!	1.14
Introducción a HTML	1.15
ORM de Django (Querysets)	1.16
Datos dinámicos en plantillas	1.17
Plantillas de Django	1.18
CSS – Hazlo bonito	1.19
Extender plantillas	1.20
Extiende tu aplicación	1.21
Formularios en Django	1.22
¿Qué sigue?	1.23
Instalación	1.24

Tutorial de Django Girls

[gitter](#) [join chat](#)

Este trabajo está bajo la licencia internacional Creative Commons Attribution-ShareAlike 4.0.

Para ver una copia de esta licencia, visita el siguiente enlace

<https://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, Geraldina Garcia Alvarez and Manuel Kaufmann.

Introducción

¿Alguna vez has sentido que el mundo se relaciona cada vez más a la tecnología y de alguna manera 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 parece y queremos mostrarte lo divertido puede llegar a ser.

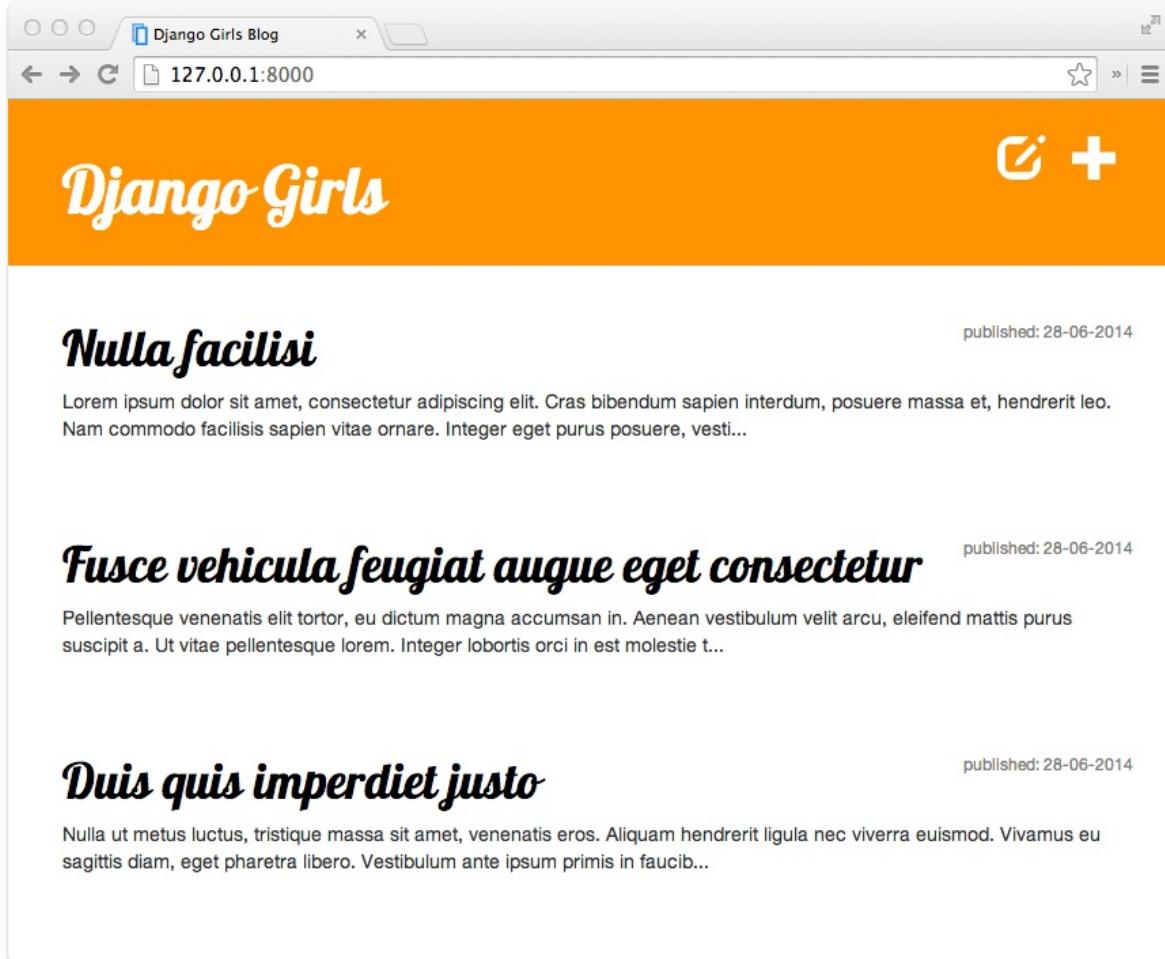
Este tutorial no te convertirá en programadora por arte de magia. Sí quieres ser buena en esto, necesitarás meses o incluso años de aprendizaje y práctica. Sin embargo queremos mostrarte que programar o crear sitios web no es tan complicado como parece. Intentaremos explicar pequeñas partes lo mejor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

¡Esperamos poder hacerte amar la tecnología tanto como nosotras!

¿Qué aprenderás con este tutorial?

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

Se verá (más o menos) como ésta:



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 otras con el tutorial! ¡No temas dejar tus preguntas allí!

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

Sobre nosotras 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?

Para los lectores en casa: este capítulo está cubierto en el video [How the Internet Works](#) (en Inglés).

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 <https://djangogirls.org> en tu navegador y presionas 'Enter'?

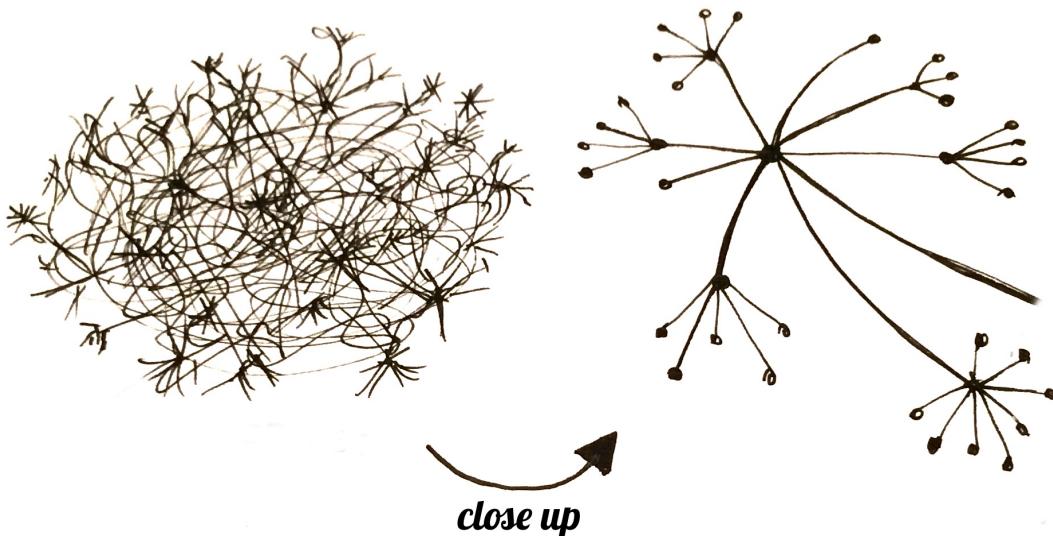
Lo primero que tienes que entender es que un sitio web es sólo un montón de archivos guardados en un disco duro. Al igual que tus películas, música o fotos. Sin embargo, los sitios web poseen una peculiaridad: incluyen un código informático llamado HTML.

Si no estás familiarizada con la programación, puede ser difícil de captar HTML a la primera, pero tus navegadores web (como Chrome, Safari, Firefox, etc.) lo aman. Los navegadores están diseñados para entender este código, seguir sus instrucciones y presentar estos archivos de los cuales está hecho tu sitio web, exactamente de la forma que quieras.

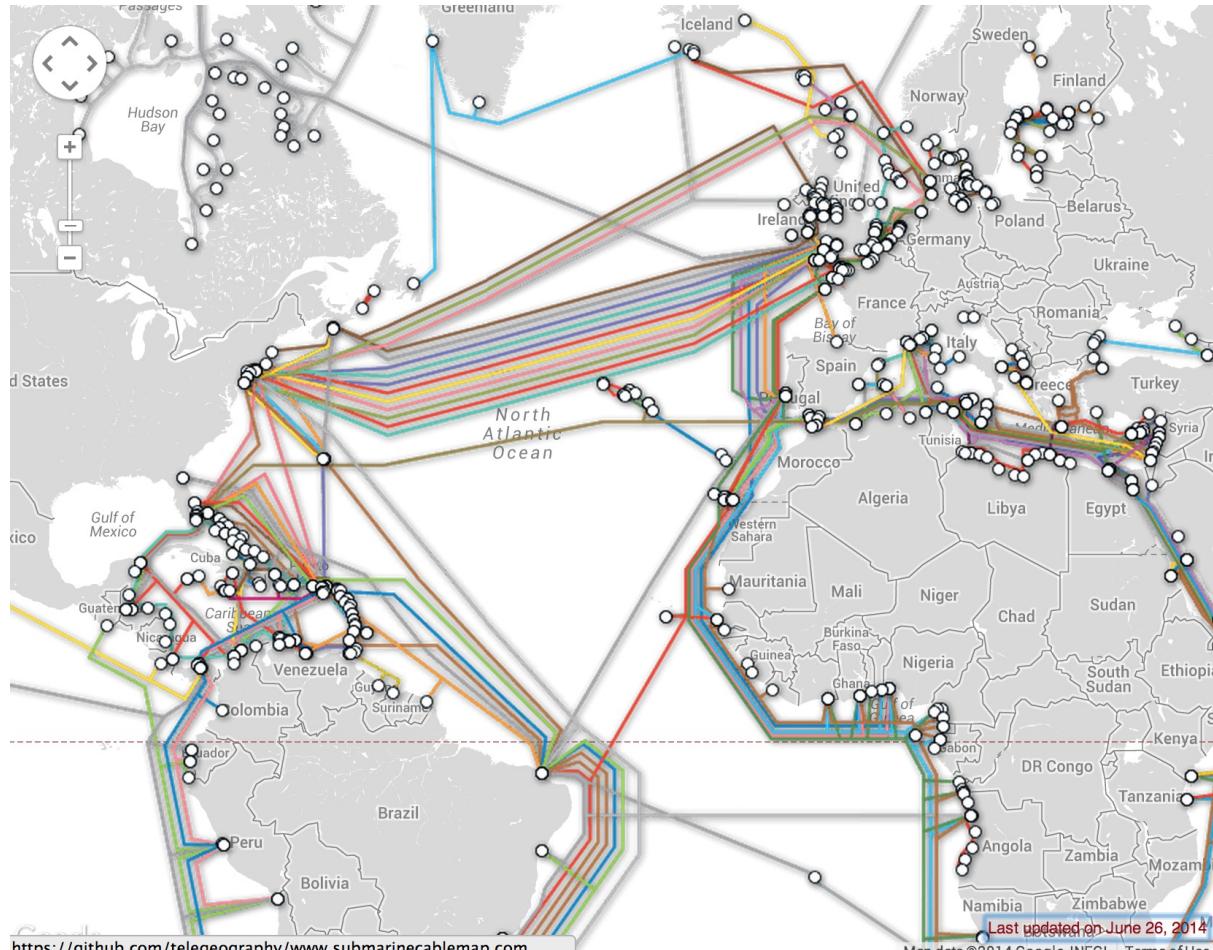
Como cualquier otro archivo, tenemos que guardar los archivos HTML en algún lugar de un disco duro. Para Internet, usamos computadoras especiales y poderosas llamadas *servidores*. Estos no tienen una pantalla, ratón o teclado, debido a que su propósito es almacenar datos y servirlos. Por esa razón son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

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

¡Te hemos hecho una imagen! Luce algo así:

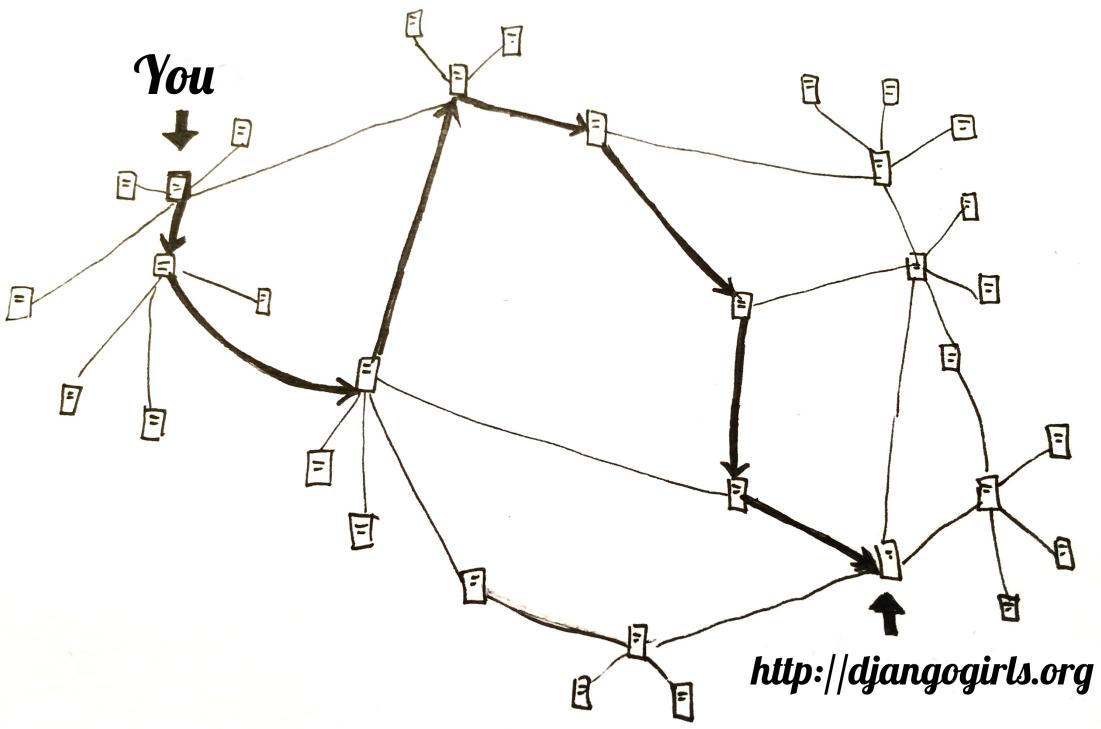


Parece un lío, ¿no? En realidad es una red de máquinas conectadas (los mencionados *servidores*). ¡Cientos de miles de máquinas! ¡Muchos, muchos kilómetros de cables alrededor del mundo! Puedes visitar el sitio web Submarine Cable Map (<http://submarinecablemap.com/>) y ver lo complicada que es la red.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



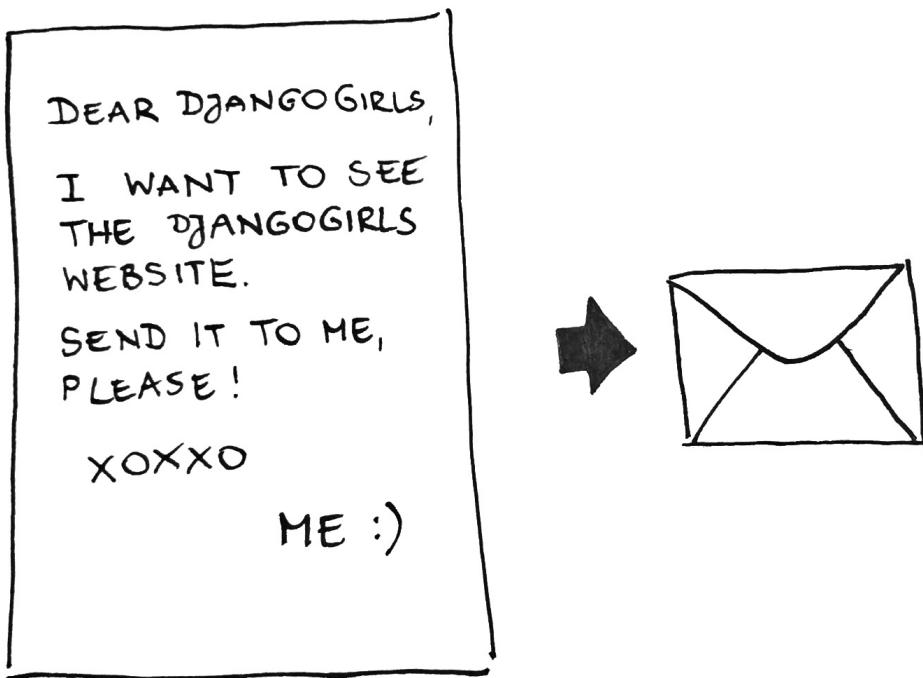
Es fascinante, ¿no? Pero, obviamente, no es posible tener un cable entre cada máquina conectada a Internet. Así que, para llegar a una máquina (por ejemplo la que aloja a <https://djangogirls.org>) tenemos que pasar una solicitud a través de muchas máquinas diferentes.

Se parece a esto:



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

Tu carta va hacia la oficina de correo más cercana. Luego va a otra que es un poco más cerca de su destinatario, luego otra y otra hasta que es entregada a su destino. La única cosa diferente es que si envías muchas cartas (*paquetes de datos*) al mismo lugar, cada una podría ir a través de oficinas de correos totalmente distintas (*routers*). Esto depende 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 djangogirls.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. Cuando el *servidor* recibe una *peticIÓN entrante* (en una carta), este envía su sitio de Internet (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

Para los lectores en casa: este capítulo está cubierto en el video [Your new friend: Command Line](#) (en Inglés).

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.

Nota Ten en cuenta que a lo largo de este libro usamos los términos 'directorio' y 'carpeta' indistintamente pero son la misma cosa.

¿Qué es la línea de comandos?

La ventana que se llama generalmente la **línea de comandos** o la **interfaz de línea de comandos**, es una aplicación basada en texto para ver, manejar y manipular archivos en tu computadora. Al igual que Windows Explorer 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

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.

OS X y Linux

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

Terminal

```
$
```

Windows

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

Terminal

```
>
```

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 puede que haya algo como `C:\Users\ola>` o `olas-MacBook-Air:~ ola$` antes del prompt y eso está perfecto. En este tutorial lo simplificaremos lo más posible.

La parte hasta e incluyendo el `$` o el `>` se llama la *línea de comandos* o *prompt*. Está a la espera de que escribas algo ahí.

En el tutorial, cuando queremos que escribas un comando, vamos a incluir el `$` o `>`, y en algunas ocasiones, algo más a la izquierda. Puedes ignorar la parte izquierda y solo escribir el comando que comienza después del prompt.

Tu primer comando (¡YAY!)

Vamos a empezar con algo simple. Escribe este comando:

OS X y Linux

Terminal

```
$ whoami
```

Windows

Terminal

```
> whoami
```

Y pulsa `intro`. Este es nuestro resultado:

Terminal

```
$ 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!

Fundamentos

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

Estaría bien saber dónde estamos ahora, ¿verdad? Vamos a ver. Escribe este comando y pulsa `intro`:

OS X y Linux

Terminal

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

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

Windows

Terminal

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

Nota: 'cd' significa 'current directory' - en español, 'directorio actual'.

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

Listar archivos y directorios

¿Qué hay aquí? Sería bueno saber. Veamos:

OS X y Linux

Terminal

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

Windows

Terminal

```
> 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

Ahora, vayamos a nuestro directorio Desktop, el escritorio:

OS X y Linux

Terminal

```
$ cd Desktop
```

Windows

Terminal

```
> cd Desktop
```

Comprueba si realmente ha cambiado:

OS X y Linux

Terminal

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

Windows

Terminal

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

¡Aquí está!

Truco pro: si escribes `cd D` y luego pulsas `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 práctica en el escritorio? Lo puedes hacer de esta manera:

OS X y Linux

Terminal

```
$ mkdir practice
```

Windows

Terminal

```
> mkdir practice
```

Este pequeño comando creará una carpeta con el nombre `practice` en el escritorio. ¡Puedes comprobar si está ahí mirando en el escritorio o ejecutando el comando `ls` o `dir`! ¡Inténtalo :)

Truco pro: Si no quieres escribir una y otra vez los mismos comandos, prueba pulsando la flecha arriba y la flecha abajo de tu teclado para ir pasando por los comandos utilizados recientemente.

¡Ejercicios!

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

Solución:

OS X y Linux

Terminal

```
$ cd practice
$ mkdir test
$ ls
test
```

Windows

Terminal

```
> cd practice
> mkdir test
> dir
05/08/2014 07:28 PM <DIR>      test
```

¡Felicitaciones! :)

Limpieza

No queremos dejar un lío, así que vamos a eliminar todo lo que hemos hecho hasta este momento.

En primer lugar, tenemos que volver al escritorio:

OS X y Linux

Terminal

```
$ cd ..
```

Windows

Terminal

```
> cd ..
```

Usar `..` con el comando `cd` hará que cambie el directorio actual al directorio padre (es el que contiene el directorio actual).

Revisa dónde estás:

OS X y Linux

Terminal

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

Windows

Terminal

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

Es el momento de eliminar el directorio `practice`:

Atención: Eliminar archivos utilizando `del`, `rmdir` o `rm` hace que no puedan recuperarse, lo que significa que los *archivos borrados desaparecerán para siempre*. Así que ten mucho cuidado con este comando.

OS X y Linux

Terminal

```
$ rm -r practice
```

Windows

Terminal

```
> rmdir /S practice
practice, Are you sure <Y/N>? Y
```

¡Hecho! Para asegurarnos de que realmente se ha eliminado, vamos a comprobarlo:

OS X y Linux

Terminal

```
$ ls
```

Windows

Terminal

```
> dir
```

Salida

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

OS X y Linux

Terminal

```
$ exit
```

Windows

Terminal

```
> exit
```

Genial, ¿no? :)

Resumen

Aquí hay una lista de algunos comandos útiles:

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

Estos son sólo algunos de los comandos que puedes ejecutar en la línea de comandos pero hoy no vas a utilizar ninguno más.

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, déjanos decirte qué es Python. Python es un lenguaje de programación muy popular que puede ser usado para crear sitios web, juegos, software científico, gráficos, y más, mucho más.

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

Instalación de Python

Nota Si estás utilizando una Chromebook, salta este capítulo y asegúrate de seguir las instrucciones de [Chromebook Setup](#).

Nota Si ya has seguido los pasos de instalación, no hay necesidad de hacerlo nuevamente - ¡puedes saltar y continuar al siguiente capítulo!

Para lectores en casa: esta parte está cubierta en el video [Installing Python & Code Editor](#).

Esta sección está basada en un tutorial por Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>)

Django está escrito en Python. Necesitamos Python para hacer cualquier cosa en Django. ¡Vamos a empezar con la instalación! Queremos que instales Python 3.5, así que si tienes alguna versión anterior, deberás actualizarla.

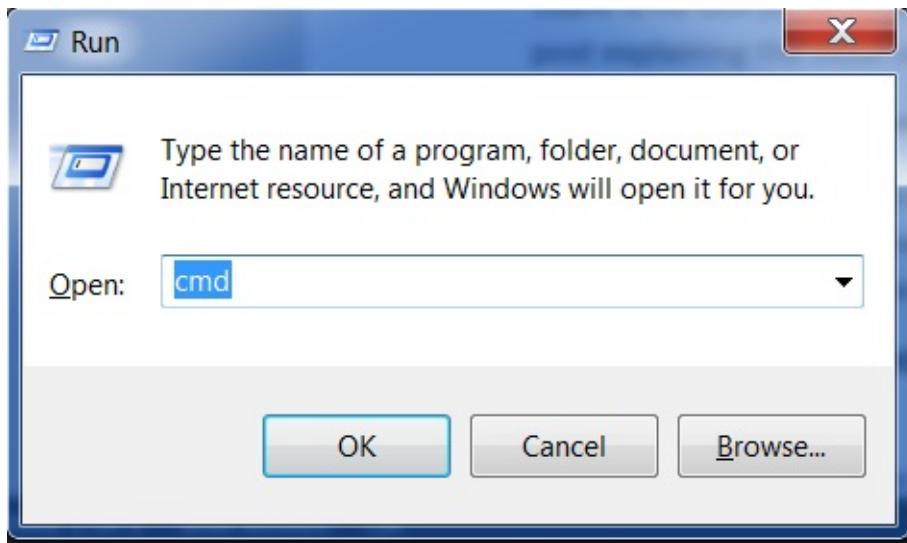
Windows

Primero comprueba si tu computadora está corriendo una versión de 32-bit o una de 64-bit de Windows en <https://support.microsoft.com/en-au/kb/827218>. Puedes descargar Python para Windows desde el sitio web <https://www.python.org/downloads/windows/>. Haz click en link "Latest Python 3 Release - Python x.x.x". Si tu computadora está corriendo una versión de **64-bit** de Windows, descarga el **Windows x86-64 executable installer**. De lo contrario, descarga el **Windows x86 executable installer**. Luego de descargar el instalador, debes ejecutarlo (doble click en él) y seguir las instrucciones.

Algo para tener en cuenta: Durante la instalación notarás una ventana llamada "Setup". Asegúrate de tildar la opción "Add Python 3.5 to PATH" y luego hacer click en "Install Now", como se muestra aquí:



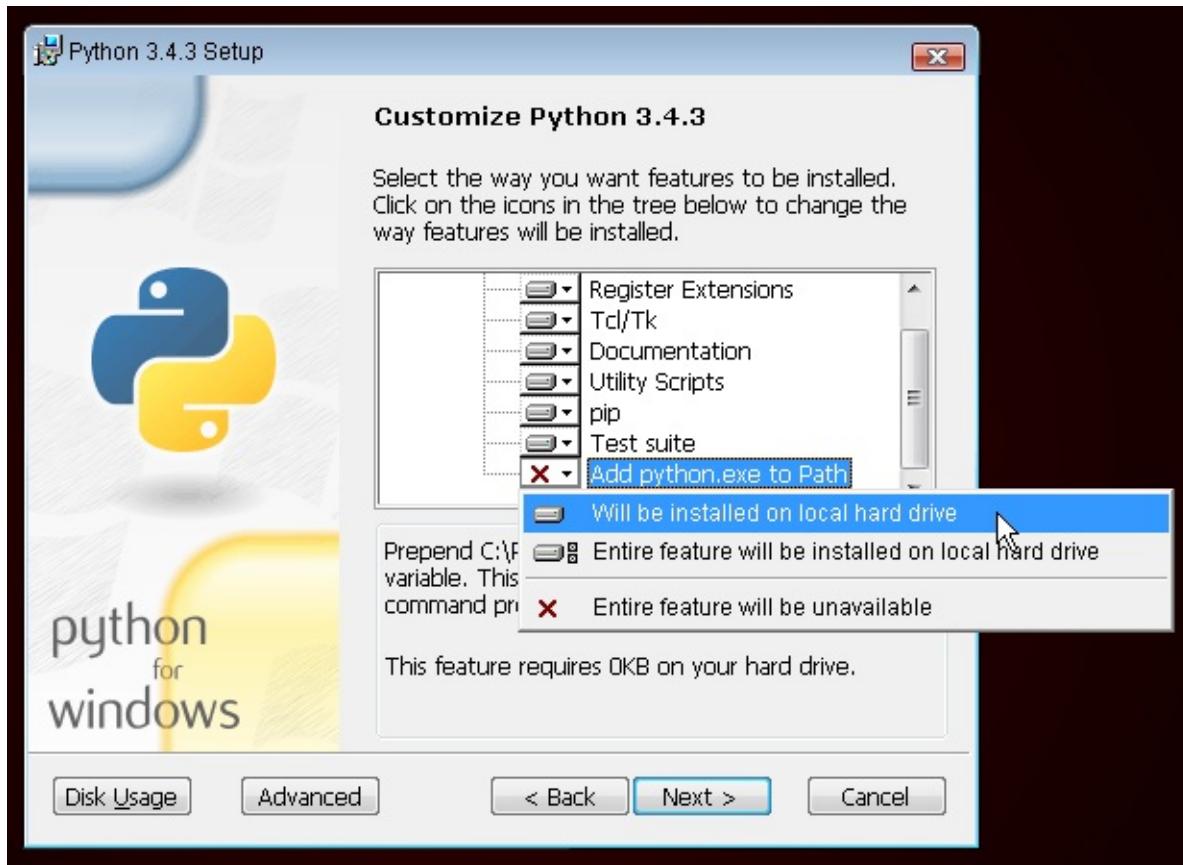
En los siguientes pasos, vamos a usar la Línea de Comandos de Windows (ya te vamos a contar sobre ella). Por ahora, si necesitas escribir algún comando, ve a Menu → Todos los programas → Accesorios → Símbolo del sistema. También puedes mantener presionada la tecla de Windows y presionar la tecla "R" hasta que aparezca la ventanita "Run". Para abrir la Línea de Comandos de Windows, escribe "cmd" y presiona `intro` la ventana "Run" (en las nuevas versiones de Windows, quizás tengas que buscar "Línea de Comandos" ya que a veces está escondido)



Nota: si estás usando una versión más vieja de Windows (7, Vista o cualquiera más vieja) y el instalador de Python 3.5.x falla con un error, puedes probar lo siguiente:

1. instala todos las Actualizaciones de Windows y prueba volver a instalar Python 3.5; o
2. instala una [versión más vieja de Python](#), por ejemplo la [3.4.4](#).

Si instalas una versión más vieja de Python, es posible que la pantalla del instalador se vea un poquito diferente que la mostrada anteriormente. Asegúrate de ir hacia el final y ver "Add python.exe to Path", luego hacer click en el botón y seleccionar "Will be installed on local hard drive":



OS X

Nota Antes de que instales Python en OS X, debes asegurarte de que las configuraciones de tu Mac te permitan instalar paquetes que no son del App Store. Ve a Preferencias del Sistema (está en la carpeta Aplicaciones), haz click en "Seguridad y Privacidad", y luego en la pestaña "General". Si yo "Permitir aplicaciones descargadas desde:" está configurado como "Mac App Store", cámbialo a "Mac App Store and identified developers"

Necesitas ir a la página web <https://www.python.org/downloads/release/python-351/> y descargar el instalador de Python:

- Descarga el archivo *Mac OS X 64-bit/32-bit installer*,
- Haz doble click en *python-3.5.1-macosx10.6.pkg* para ejecutar el instalador.

Linux

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

Terminal

```
$ python3 --version  
Python 3.5.1
```

Si tienes una 'micro versión' diferente de Python instalado, por ejemplo 3.5.0, no necesitas actualizar. Si no tienes instalado Python o si deseas una versión diferente, puedes instalarla de la siguiente manera:

Debian o Ubuntu

Escribe este comando en tu consola:

Terminal

```
$ sudo apt-get install python3.5
```

Fedora (hasta 21)

Usa este comando en tu consola:

Terminal

```
$ sudo yum install python3
```

Fedora (22+)

Usa este comando en tu consola:

Terminal

```
$ sudo dnf install python3
```

openSUSE

Use this command in your console:

Terminal

```
$ sudo zypper install python3
```

Verifica que la instalación fue correcta abriendo la aplicación de *Terminal* y ejecutando el comando

```
python3 :
```

Terminal

```
$ python3 --version  
Python 3.5.1
```

NOTA: Si estás en Windows y obtienes un mensaje de error diciendo `python3` no se reconoce como comando, intenta con `python` (sin el `3`) y comprueba si todavía es una versión de Python 3.5.

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

Editor de código

Para lectores en casa: este capítulo está cubierto en el video [Installing Python & Code Editor](#) (en inglés).

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

Nota Si estás usando una Chromebook, salta este capítulo y asegúrate de seguir las [instrucciones de instalación para Chromebook](#) instructions.

Nota Es posible que ya hayas hecho esto en el capítulo de instalación. Si es así, ¡puedes avanzar directamente al siguiente capítulo!

Existe una gran cantidad de editores diferentes y la elección queda reducida en gran medida a las preferencias personales. La mayoría de programadoras de Python usan IDEs (Entornos de Desarrollo Integrados) complejos pero muy poderosos, como PyCharm. Sin embargo, como principiante, probablemente no es muy adecuado; nuestras recomendaciones son igualmente poderosas pero mucho más simples.

Nuestras sugerencias están listadas abajo, pero siéntete libre de preguntarle a tu tutora o 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 3

Sublime Text es un editor muy popular con un periodo de prueba gratis. Es fácil de instalar y de usar, 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 no producen texto plano. Lo que generan es texto enriquecido (con fuentes y formato), usando formatos propios como [RTF \(Rich Text Format\)](#) o en español, "Formato de Texto Enriquecido".

La segunda razón es que los editores de código son herramientas especiales para editar código, porque proveen características útiles como resaltar el 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!

El prompt de Python

Para lectores en casa: esta parte está cubierta en el video [Python Basics: Integers, Strings, Lists, Variables and Errors](#) (en Inglés).

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

Una vez que estés lista, sigue las siguientes instrucciones.

Queremos abrir una consola de Python, así que escribe `python` en Windows o `python3` en Mac OS/Linux y pulsa `intro`.

Terminal

```
$ python3
Python 3.5.1 (...)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

¡Tu primer comando en Python!

Después de ejecutar el comando de Python, el cursor cambia a `>>>`. Para nosotras 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 `>>>`.

Por ahora no queremos salir de la consola de Python. Queremos aprender más sobre ella. Empecemos con algo muy sencillo. Por ejemplo, intenta escribir algo matemático, como `2 + 3` y presiona `intro`.

Terminal

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

¡Bien! ¿Ves como salió la respuesta? ¡Python sabe matemáticas! Podrías intentar otros comandos como:

- `4 * 5`
- `5 - 1`
- `40 / 2`

Para realizar cálculos exponentiales, por ejemplo, 2 a la potencia de 3, escribimos:

Terminal

```
>>> 2 ** 3  
8
```

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...

Cadenas de caracteres

¿Cuál es tu nombre? Escribe tu nombre de pila entre comillas así:

Terminal

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

¡Has creado tu primera cadena! Es una secuencia de caracteres que puede ser procesada por una computadora. Una cadena, "string" en inglés, siempre debe comenzar y terminar con el mismo carácter. Pueden ser comillas simples (') o dobles (") (no hay diferencia!) Las comillas le dicen a Python que lo que está dentro de ellas es una cadena.

Las cadenas pueden ser concatenadas. Prueba esto:

Terminal

```
>>> "Hi there " + "ola"  
'Hi there ola'
```

También puedes multiplicar las cadenas por un número:

Terminal

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

Si necesitas poner un apóstrofe dentro de la cadena, tienes dos formas de hacerlo.

Usar comillas dobles:

Terminal

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

o escapar el apóstrofe con una barra invertida (\):

Terminal

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

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

Terminal

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

¡Acabas de usar la función `upper` sobre 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 hay una función para eso!

Terminal

```
>>> 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 `*`, combinan valores para producir uno nuevo
- **funciones** - como `upper()` y `len()`, realizan operaciones 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

Vamos a intentar algo nuevo. ¿Podemos obtener la longitud de un número de la misma manera que pudimos averiguar la longitud de nuestro nombre? Escribe `len(304023)` y pulsa `intro`:

Terminal

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

¡Tenemos nuestro primer error! Dice que los objetos de tipo "int" (números enteros) no tienen longitud. ¿Qué podemos hacer ahora? ¿Quizá podamos escribir el número como una cadena? Las cadenas tienen longitud, ¿verdad?

Terminal

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

¡Funcionó! Hemos utilizado la función `str` dentro de la función `len`. `str()` convierte todo en cadenas.

- La función `str` convierte cosas en cadenas, **strings**
- La función `int` convierte cosas en enteros, **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 algo de forma que puedas usarlo más tarde. Los programadores usan estas variables para almacenar datos, hacer su código más legible y para no tener que recordar qué es cada cosa.

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

Terminal

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

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

Como habrás notado, el programa no devuelve nada como lo hacía antes. ¿Cómo sabemos que la variable existe realmente? Simplemente escribe `name` y pulsa `intro`:

Terminal

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

¡Genial! ¡Tu primera variable :)! Siempre puedes cambiar a lo que se refiere:

Terminal

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

También puedes usarla dentro de funciones:

Terminal

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

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

Prueba esto:

Terminal

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

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

Terminal

```
>>> 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 un rato y descubre qué puedes hacer!

La función print

Intenta esto:

Terminal

```
>>> name = 'Maria'  
>>> name  
'Maria'  
>>> print(name)  
Maria
```

Cuando sólo escribes `name`, el intérprete de Python responde con la *representación* en forma de cadena 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 más claro.

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

Anímate y crea una lista:

Terminal

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

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 la pondremos también en una variable:

Terminal

```
>>> 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 lo sabes!

Terminal

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

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

Terminal

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

No devuelve nada, sólo ha cambiado el orden en que los números aparecen en la lista. Vamos a imprimirla otra vez y ver que ha pasado:

Terminal

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

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

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

Terminal

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

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

Terminal

```
>>> 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. Los programadores prefieren comenzar a contar desde 0, por lo tanto el primer objeto en tu lista está en el índice 0, el próximo está en el 1, y así sucesivamente. Intenta esto:

Terminal

```
>>> 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 borrar algo de tu lista necesitas usar **índices** como aprendimos anteriormente y el método `pop()`. Vamos a ver un ejemplo y reforzar lo que aprendimos anteriormente; vamos a borrar el primer número de nuestra lista.

Terminal

```
>>> print(lottery)
[59, 42, 30, 19, 12, 3, 199]
>>> print(lottery[0])
59
>>> lottery.pop(0)
>>> print(lottery)
[42, 30, 19, 12, 3, 199]
```

¡Funcionó de maravilla!

Para diversión adicional, prueba algunos otros índices: 6, 7, 1000, -1, -6 ó -1000. A ver si se puedes 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

Para lectores en casa: esta parte está cubierta en el video [Python Basics: Dictionaries](#) (en Inglés).

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:

Terminal

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

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

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

Terminal

```
>>> 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:

Terminal

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

Mira, 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!

Terminal

```
>>> 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.

¿Cuando deberías usar un diccionario o una lista? Bueno, es un buen punto para reflexionar. Simplemente ten una solución en mente antes de buscar una 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 quiere decir que pueden ser modificados después de ser creados. Puedes agregar nuevos pares clave/valor a un diccionario luego de crearlo, como:

Terminal

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

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

Terminal

```
>>> 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 método `pop()` 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:

Terminal

```
>>> participant.pop('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:

Terminal

```
>>> 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

Para lectores en casa: esta parte está cubierta en el video [Python Basics: Comparisons](#) (en Inglés).

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

Terminal

```
>>> 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:

Terminal

```
>>> 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:

Terminal

```
>>> 6 > 2 and 2 < 3
True
>>> 3 > 2 and 2 < 1
```

```

False
>>> 3 > 2 or 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:

Terminal

```

>>> 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 comparar.

Boolean

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

Hay sólo dos objetos booleanos:

- `True`
- `False`

Pero para que Python entienda esto, siempre necesitas escribir los como 'True' (la primera letra en mayúscula, con el resto de las letras en minúscula). `true`, `TRUE`, `tRUE` no funcionarán -- solo `True` es correcto. (Lo mismo aplica para 'False', por supuesto.)

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

Terminal

```

>>> a = True
>>> a
True

```

También puedes hacerlo de esta manera:

Terminal

```
>>> 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

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

¡Guárdalo!

Hasta ahora hemos escrito todo nuestro código Python en el intérprete, lo cual nos limita a ingresar 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() :
```

Terminal

```
>>> 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:

editor

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

Obviamente, ahora eres una desarrolladora Python muy experimentada, así que sintete 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 como queramos, pero la parte importante es asegurarse de que termina en **.py**. La extensión **.py** le dice a nuestro sistema operativo que es un **archivo ejecutable de Python** y que Python puede correrlo.

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, ahora deberías ver que la función `print` es de un color diferente de las cadenas dentro de ella. Esto se denomina "sintaxis resaltada", y es una característica muy útil cuando se programa. El color de las cosas te dará pistas, como cadenas no cerradas o errores tipográficos en un nombre clave (como la función `def`, que veremos a continuación). Esta es una de las razones por las cuales usar un editor de código :)

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.

OS X

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

Terminal

```
$ cd ~/Desktop
```

Linux

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

Terminal

```
$ cd ~/Desktop
```

Windows

Y en Windows, será así:

Terminal

```
> cd %HomePath%\Desktop
```

Si te quedas atascada, sólo pide ayuda.

Ahora usa Python para ejecutar el código en el archivo así:

Terminal

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

Nota: en Windows 'python3' no es reconocido como un comando. En cambio, utiliza 'python' para ejecutar el archivo:

Terminal

```
> python python_intro.py
```

¡Muy bien! Ejecutaste tu primer programa de Python desde un archivo. ¿No te sientes 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** con esto:

python_intro.py

```
if 3 > 2:
```

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

Terminal

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

Python espera que le demos más instrucciones las cuales se ejecutan si la condición `3 > 2` es verdadera (o `True`). Intentemos hacer que Python imprima "It works!". Cambia tu código en el archivo **python_intro.py** para que se vea como esto:

python_intro.py

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

¿Observas cómo hemos indentado la siguiente línea de código con 4 espacios? Necesitamos hacer esto para que Python sepa que código ejecutar si es resultado es verdadero. Puedes poner un espacio, pero casi todas las programadoras y los programadores Python hacen 4 espacios para hacer que el código sea más legible. Un solo tab también cuenta como 4 espacios.

Guárdalo y ejecútalo de nuevo:

Terminal

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

Note: Recuerda que en Windows, 'python3' no es reconocido como un comando. De ahora en adelante, reemplaza 'python3' por 'python' para ejecutar el archivo.

¿Qué pasa si una 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 :

python_intro.py

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

Cuando esto se ejecute imprimirá:

Terminal

```
$ 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 :

python_intro.py

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

y al ejecutarlo:

Terminal

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

¿Ves lo que pasó allí? `elif` te permite agregar condiciones adicionales que se ejecutan si la condición previa falla.

Puedes agregar tantas sentencias `elif` como quieras después de la sentencia `if` inicial. Por ejemplo:

`python_intro.py`

```
volume = 57  
if volume < 20:  
    print("It's kinda quiet.")  
elif 20 <= volume < 40:  
    print("It's nice for background music")  
elif 40 <= volume < 60:  
    print("Perfect, I can hear all the details")  
elif 60 <= volume < 80:  
    print("Nice for parties")  
elif 80 <= volume < 100:  
    print("A bit loud!")  
else:  
    print("My ears are hurting! :(")
```

Python corre a través de cada prueba en secuencia e imprime:

Terminal

```
$ python3 python_intro.py  
Perfect, I can hear all the details
```

Comentarios

Las líneas que comienzan con `#` son comentarios. Puedes escribir lo que quieras luego de `#` y Python lo ignorará. Los comentarios pueden hacer que tu código sea más fácil de entender para otras personas.

`python_intro.py`

```
# Change the volume if it's too loud or too quiet  
if volume < 20 or volume > 80:  
    volume = 50  
    print("That's better!")
```

No necesitas escribir un comentario por cada línea de código, pero son útiles para explicar porqué tu código está haciendo algo, o proveer un resumen cuando se está haciendo algo complejo.

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!

Para lectores en casa: esta parte está cubierta en el video [Python Basics: Functions](#) (en Inglés).

¿Recuerdas funciones como `len()` que puedes ejecutar en Python? Bueno, te tenemos buenas noticias, ¡ahora aprenderás a 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:

`python_intro.py`

```
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:

Terminal

```
$ python3 python_intro.py
```

```
Hi there!  
How are you?
```

Nota: si no funciona, ¡no te desesperes!. El resultado te ayudará a entender porqué:

- Si recibiste un `NameError`, eso probablemente significa que has tipeado algo de forma incorrecta. Deberías comprobar que utilizaste el mismo nombre cuando creaste la función con `def hi():` y cuando la llamaste con `hi()`.
- Si recibiste un `IndentationError`, comprueba que ambas líneas `print` tengan la misma cantidad de espacios blancos desde el principio de la línea: Python quiere todo el código dentro de la función alineado correctamente.
- Si no hay ninguna respuesta, comprueba que el último `hi()` no esté indentado - si lo está, esa línea será parte de la función también y nunca será ejecutada.

Construyamos nuestra primera función con parámetros. Utilizaremos el ejemplo anterior - una función que dice 'Hi' a la persona que ejecuta el programa - con un nombre:

python_intro.py

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

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

python_intro.py

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

Recuerda: la función `print` está indentada cuatro espacios dentro de la condición `if`. Esto es porque la función se ejecutan cuando la condición se cumple. Vamos a ver cómo funciona:

Terminal

```
$ 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:

python_intro.py

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

Y lo ejecutamos de nuevo:

Terminal

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

¿Y si cambiamos el nombre?

python_intro.py

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

Y lo ejecutamos:

Terminal

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

Ahora, ¿qué crees que suceda si escribes otro nombre ahí? (No Ola ni Sonja). Inténtalo y ve si tienes razón. Debería imprimir esto:

Terminal

```
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?

python_intro.py

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

Ahora vamos a llamar al código:

Terminal

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

¡Felicitaciones! Acabas de aprender cómo escribir funciones :)

Bucles

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

A los programadores 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:

python_intro.py

```
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:

python_intro.py

```
for name in girls:
```

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

Aquí está el código completo que estará en el archivo:

python_intro.py

```
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:

Terminal

```
$ 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`:

`python_intro.py`

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

Lo que imprimirá:

Terminal

```
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). Eso es porque "range" está medio-aberto, y con eso queremos decir que incluye el primer valor, pero no el último.

Resumen

Eso es todo. ¡Eres genial! Este fue un capítulo difícil, por lo que debes sentirte orgullosa de ti misma. ¡Nosotras estamos orgullosas de ti porque has llegado lejos!

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/ˈdʒæŋgou/jang-goh*) es un framework para aplicaciones web gratuito y open source, escrito en Python. Un framework web es un conjunto de componentes que te ayudan a desarrollar sitios web más fácil y rápidamente.

Cuando construyes un sitio web, siempre 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 atrás varias personas notaron que los desarrolladores web enfrentan problemas similares cuando construyen un sitio nuevo, por eso se unieron 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 nuevo.

¿Por qué necesitas un framework?

Para entender para que es Django, necesitamos mirar mas de cerca a los servidores. Lo primero que el servidor necesita saber es 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 quien 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 pasada a Django quien 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 (ten 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 hacer coincidir la URL. Django comprueba los patrones de arriba hacia abajo y si algo coincide entonces Django le pasa la solicitud a la función asociada (que se llama `vista`).

Imagina a una cartera 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 permiso para hacerlo, actualizar la descripción de tu trabajo y devolver un mensaje: "¡Hecho!". Entonces la *vista* genera una respuesta y Django puede enviarla al navegador web del usuario.

Por supuesto, la descripción anterior está un poco simplificada, pero no necesitas saber todas las cosas técnicas aún. Tener una idea general es suficiente.

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

Instalación de Django

Nota Si estás utilizando una Chromebook, salta este capítulo y asegúrate de seguir las instrucciones de [Chromebook Setup](#)

Nota Si ya has realizado los pasos de instalación, esto ya lo has hecho. ¡Puedes avanzar directamente al siguiente capítulo!

Parte de esta sección está basada en tutoriales de Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Parte de esta sección se basa en el [django-marcador tutorial](#) bajo licencia 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 tu computadora. Es posible saltarse este paso, pero es altamente recomendable. ¡Empezar con la mejor configuración posible te ahorrará muchos problemas en el futuro!

Así que, vamos a crear un **entorno virtual** (también llamado un *virtualenv*). Virtualenv aísla tu configuración de Python/Django por cada proyecto. Esto quiere decir que cualquier cambio que hagas en un sitio web no afectará a ningún otro que estés desarrollando. Genial, ¿no?

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

NOTA: En Windows, asegúrate que este directorio no contenga acentos ni caracteres especiales; si tu nombre contiene acentos, utiliza un directorio diferente, por ejemplo `C:\djangogirls`.

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

Terminal

```
$ mkdir djangogirls  
$ cd djangogirls
```

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

Terminal

```
$ 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:\Python35\python -m venv myvenv`. Se verá así:

Terminal

```
C:\Users\Name\djangogirls> C:\Python35\python -m venv myvenv
```

en donde `C:\Python35\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 usar 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:

Terminal

```
$ python3 -m venv myvenv
```

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

NOTA: En algunas versiones de Debian/Ubuntu quizás recibas el siguiente error:

Terminal

```
The virtual environment was not created successfully because ensurepip is not available. On Debian/Ubuntu systems, you need to install the python3-venv package using the following command.  
apt-get install python3-venv  
You may need to use sudo with that command. After installing the python3-venv package, recreate your virtual environment.
```

En este caso, sigue las instrucciones mencionadas arriba e instala el paquete `python3-venv` así:

Terminal

```
$ sudo apt-get install python3-venv
```

NOTA: En algunas versiones de Debian/Ubuntu inicializar el entorno virtual puede producir el siguiente error:

Terminal

```
Error: Command '['/home/eddie/Slask/tmp/venv/bin/python3', '-Im', 'ensurepip', '--upgrade', '--default-pip']}' returned non-zero exit status 1
```

Para solucionar esto, usa el comando `virtualenv` en cambio.

Terminal

```
$ sudo apt-get install python-virtualenv  
$ virtualenv --python=python3.5 myvenv
```

NOTA: Si obtienes un error como este

Terminal

```
E: Unable to locate package python3-venv
```

intenta ejecutar el siguiente comando en cambio:

Terminal

```
sudo apt install python3.5-venv
```

Trabajar con `virtualenv`

El comando anterior creará un directorio llamado `myvenv` (o cualquier nombre que hayas elegido) que contiene nuestro entorno virtual (básicamente un montón de archivos y carpetas).

Windows

Inicia el entorno virtual ejecutando:

Terminal

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

NOTA: en Windows 10 quizás obtengas un error en la consola Window PowerShell diciendo `execution of scripts is disabled on this system`. En ese caso, abre otra ventana Windows PowerShell con la opción "Run as Administrator". Luego intenta escribir lo siguiente antes de iniciar tu entorno virtual:

Terminal

```
C:\WINDOWS\system32> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned  
Execution Policy Change  
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the about_Execution_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy? [Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): A
```

Linux y OS X

Inicia el entorno virtual ejecutando:

Terminal

```
$ source myvenv/bin/activate
```

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

NOTA: a veces `source` podría no estar disponible. En ese caso trata hacerlo de esta forma:

Terminal

```
$ . myvenv/bin/activate
```

Sabrás que tienes `virtualenv` iniciado cuando veas que la línea de comando tiene este prefijo `(myvenv)`.

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

OK, tenemos todas las dependencias importantes en su lugar. ¡Finalmente podemos instalar Django!

Instalar Django

Ahora que tienes tu `virtualenv` iniciado, puedes instalar Django.

Antes de hacer eso, deberíamos asegurarnos de que tenemos la última versión de `pip`, el programa que usamos para instalar Django:

Terminal

```
(myvenv) ~$ pip install --upgrade pip
```

Luego ejecuta `pip install django~=1.10.0` (nota que usamos la tilde seguido de un signo igual: `~=`) para instalar Django.

Terminal

```
(myvenv) ~$ pip install django~=1.10.0
Collecting django~=1.10.0
  Downloading Django-1.10.4-py2.py3-none-any.whl (6.8MB)
Installing collected packages: django
Successfully installed django-1.10.4
```

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`). Crea un nuevo entorno virtual en ese directorio, luego elimina el viejo y vuelve a probar el comando anterior (mover el directorio del virtualenv no funcionará debido a que virtualenv usa rutas absolutas)

Windows 8 y Windows 10

Tu línea de comandos quizás se congele luego de que intentas instalar Django. Si esto sucede, intenta este comando en cambio:

Terminal

```
C:\Users\Name\djangogirls> python -m pip install django~=1.10.0
```

Linux

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

¡Eso es todo! Ahora estás lista (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 está basado en el [tutorial django-marcador](#) 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 blog sencillo!

El primer paso es iniciar un nuevo proyecto de Django. Básicamente, significa que vamos a lanzar unos scripts proporcionados por Django que nos crearán el esqueleto de un proyecto de Django. Son solo un montón de directorios y archivos que usaremos más tarde.

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 buena idea. Django necesita mantener una cierta estructura para poder encontrar cosas importantes.

Recuerda ejecutar todo en el virtualenv. Si no ves un prefijo `(myvenv)` en tu consola necesitas activar tu virtualenv. Explicamos cómo hacerlo en el capítulo de [Instalación de Django](#) en la sección [Trabajar con virtualenv](#). Basta con escribir `myvenv\Scripts\activate` en Windows o `source myvenv/bin/activate` en Mac OS X o Linux.

OS X or Linux

En Mac OS X o Linux deberías ejecutar el siguiente comando en la consola. ¡No te olvides de agregar el punto `.` al final!

Terminal

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

El punto `.` es crucial porque le dice al script que instale Django en el directorio actual (para el cual el punto `.` sirve de abreviatura)

Nota Cuando escribas los comandos de arriba acuérdate de que sólo tienes que escribir la parte que empieza por `django-admin`. La parte `(myvenv) ~/djangogirls$` que mostramos aquí es solo un ejemplo del mensaje que aparecerá en tu línea de comandos.

Windows

En Windows deberías ejecutar el siguiente comando en la consola. ¡No te olvides de agregar el punto . al final!

Terminal

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

El punto . es crucial porque le dice al script que instale Django en el directorio actual (para el cual el punto . sirve de abreviatura)

Nota Cuando escribas los comandos de arriba acuérdate de que sólo tienes que escribir la parte que empieza por django-admin . La parte (myvenv) C:\Users\Name\djangogirls> que mostramos aquí es solo un ejemplo del mensaje que aparecerá en tu línea de comandos.

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

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

Nota: en la estructura de tus directorios también verás un directorio llamado venv que creamos anteriormente.

manage.py es un script que ayuda con la administración del sitio. Con él podremos (entre otras cosas) iniciar un servidor web en nuestra computadora sin necesidad de instalar nada más.

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

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

Por ahora vamos a ignorar el resto de archivos porque no los vamos a cambiar. ¡Sólo acuérdate de no borrarlos accidentalmente!

Cambiar 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 modifícalo para elegir tu zona horaria. Por ejemplo:

`mysite/settings.py`

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

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 línea de `STATIC_URL`, agrega una nueva línea con `STATIC_ROOT`:

`mysite/settings.py`

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

Cuando `DEBUG` está en `True` y `ALLOWED_HOSTS` está vacío, el host es validado entre `['localhost', '127.0.0.1', '[::1]']`. Esto no coincidirá con nuestro nombre de host en PythonAnywhere una vez que despleguemos nuestra aplicación, así cambiémoslo a lo siguiente:

`mysite/settings.py`

```
ALLOWED_HOSTS = ['127.0.0.1', '.pythonanywhere.com']
```

Nota: Si estás utilizando una Chromebook, agrega esta línea al final de tu archivo

```
settings.py : MESSAGE_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'
```

Configurar una base de datos

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

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

`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:

Terminal

```
(myvenv) ~/djangogirls$ python manage.py migrate
Operations to perform:
  Apply all migrations: auth, admin, contenttypes, sessions
Running migrations:
  Rendering model states... DONE
    Applying contenttypes.0001_initial... OK
    Applying auth.0001_initial... OK
    Applying admin.0001_initial... OK
    Applying admin.0002_logentry_remove_auto_add... OK
    Applying contenttypes.0002_remove_content_type_name... OK
    Applying auth.0002_alter_permission_name_max_length... OK
    Applying auth.0003_alter_user_email_max_length... OK
    Applying auth.0004_alter_user_username_opts... OK
    Applying auth.0005_alter_user_last_login_null... OK
    Applying auth.0006_require_contenttypes_0002... OK
    Applying auth.0007_alter_validators_add_error_messages... OK
    Applying sessions.0001_initial... OK
```

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

Iniciar el servidor web

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`:

Terminal

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

Si estás en una Chromebook, usa este comando en cambio:

Cloud 9

```
(myvenv) ~/djangogirls$ python manage.py runserver 0.0.0.0:8080
```

Si estás en Windows y te falla con un error `UnicodeDecodeError`, utiliza en su lugar este comando:

Terminal

```
(myvenv) ~/djangogirls$ python manage.py runserver 0:8000
```

Ahora todo lo que necesitas hacer es comprobar que tu sitio se esté ejecutando. Abre el navegador (Firefox, Chrome, Safari, Internet Explorer o el que utilices) y escribe la dirección:

browser

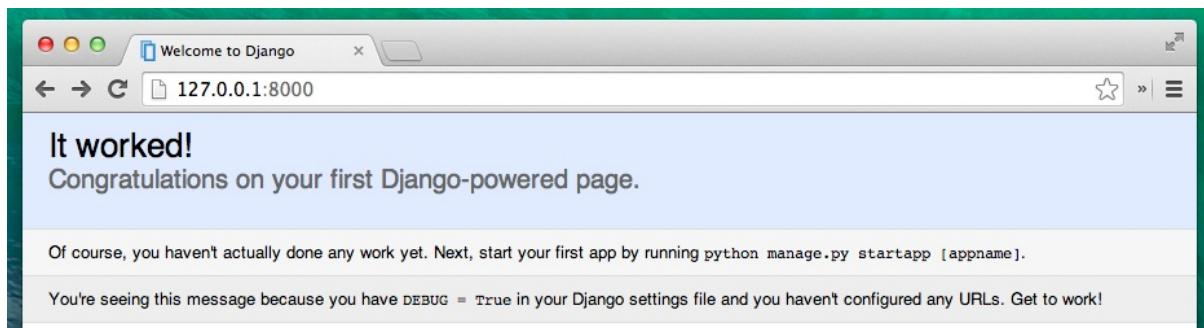
```
http://127.0.0.1:8000/
```

Si estás en una Chromebook, siempre deberás visitar tu servidor de pruebas accediendo así:

browser

```
https://django-girls-<your cloud9 username>.c9users.io
```

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



Mientras el servidor web esté corriendo, no verás un nuevo prompt en la línea de comandos para ingresar más comandos. La terminal aceptará nuevo texto ingresado pero no ejecutará los comandos. Esto es porque el servidor web se ejecuta constantemente para escuchar nuevas las nuevas peticiones que ingresen.

Hemos visto como funcionan los servidor web en el capítulo [¿Cómo funciona internet?](#)

Para escribir más comandos mientras el servidor web está funcionando, abre una nueva consola y activa el virtualenv. Para parar el servidor web, pasa a la ventana donde se esté ejecutando y pulsa Ctrl+C, las teclas Control y C a la vez (en Windows puede que tengas que pulsar Ctrl+Pausa).

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

Modelos en Django

Lo que queremos crear ahora es algo que almacene todas los posts de nuestro blog. Pero para poder hacerlo tenemos que hablar un poco sobre 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 entre ellas.

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

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

Además el `Gato` tiene algunas acciones: `ronronear`, `arañar` o `alimentarse` (en la cual daremos al gato algo de `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?

Necesitamos responder a la pregunta: ¿Qué es un post de un blog? ¿Qué propiedades debería tener?

Bueno, seguro que nuestras 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 se creó y publicó la entrada.

```
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 `publish` (publicar en Inglés).

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

Modelos 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. Es un lugar en el cual almacenarás la información sobre usuarios, tus entradas de blog, etc. Utilizaremos una base de datos SQLite para almacenar nuestros datos. Este es el adaptador de base de datos predeterminado en Django -- será suficiente para nosotros por ahora.

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

Crear 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`):

Terminal

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

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

```
djangogirls
```

```

└── blog
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
├── db.sqlite3
├── manage.py
└── mysite
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.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 agregar una línea que contiene `'blog'`, justo por encima de `]`. El producto final debe tener este aspecto:

`mysite/settings.py`

```

INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',
]

```

Crear el modelo Post

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

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

`blog/models.py`

```

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

```

Comprueba que has usado dos guiones bajos (`_`) en cada lado del `str`. Esta convención se usa en Python con mucha frecuencia y a veces también se llaman "dunder" (abreviatura de "double-underscore" o, en español, "doble guión bajo").

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

Todas las líneas que comienzan con `from` o `import` son líneas para agregar 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). Empieza siempre el nombre de una clase con una letra 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 de las que hablábamos: `title`, `text`, `created_date`, `published_date` y `author`. Para ello tenemos que definir un tipo de campo (¿es texto? ¿un número? ¿una fecha? ¿una relación con otro objeto, es decir, un usuario?).

- `models.CharField`, así es como defines un texto con un número limitado de caracteres.
- `models.TextField`, este es para texto largo sin límite. Suena perfecto para el contenido de la entrada del blog, ¿no?
- `models.DateTimeField`, este es una fecha y hora.
- `models.ForeignKey`, este es una relación con otro modelo.

No vamos a explicar aquí cada pedacito de código porque nos tomaría demasiado tiempo. Deberías echar un vistazo a la documentación de Django si quieras saber más sobre los campos de los Modelos y cómo definir cosas diferentes a las descritas anteriormente (<https://docs.djangoproject.com/en/1.10/ref/models/fields/#field-types>).

¿Y qué sobre `def publish(self):`? Es exactamente el método `publish` que mencionábamos antes. `def` significa que es una función/método y `publish` es el nombre del método. Puedes cambiar el nombre del método, si quieras. La regla de nomenclatura es utilizar minúsculas y guiones bajos en lugar de espacios. Por ejemplo, un método que calcule el precio medio se podría llamar `calcular_precio_medio`.

Los métodos suelen devolver (`return`, en inglés) algo. Hay un ejemplo de esto en el método `__str__`. En este escenario, cuando llamemos a `__str__()` obtendremos un texto (`string`) con un título de Post.

Nota también que `def publish(self):` y `def __str__(self):` están indentadas dentro de nuestra clase. Debido a que Python es sensible a los espacios en blanco, necesitamos indentar nuestros métodos dentro de las clases. De otra forma, el método no pertenecerá a la clase, y puedes obtener un comportamiento inesperado.

Si algo todavía no está claro sobre modelos, ¡no dudes en preguntar a tu guía! Sabemos que es complicado, sobre todo cuando aprendes lo que son funciones y objetos al mismo tiempo. Pero con suerte, ¡todo tiene un poco más de sentido para ti ahora!

Crear tablas para los modelos en tu base de datos

El último paso aquí es agregar nuestro nuevo modelo a la base de datos. Primero tenemos que hacer saber a Django que hemos hecho cambios en nuestro modelo (¡lo acabamos de crear!).

Escribe `python manage.py makemigrations blog`. Se verá así:

Terminal

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

Nota: Recuerda guardar los archivos que edites. De lo contrario, tu computadora ejecutará la versión anterior lo cuál quizás te muestre mensajes inesperados.

Django preparó un archivo de migración que ahora tenemos que aplicar a nuestra base de datos.

Escribe `python manage.py migrate blog` y el resultado debería ser:

Terminal

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

¡Hurra! ¡Nuestro modelo Post ya está en nuestra base de datos! Estaría bien verlo, ¿no? ¡Salta al siguiente capítulo para ver qué aspecto tiene 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:

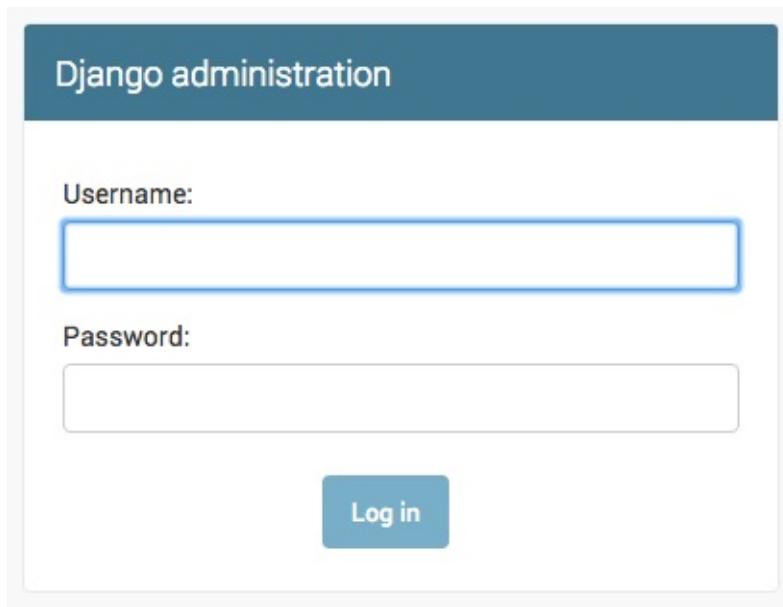
`blog/admin.py`

```
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 nuestro modelo Post. Recuerda ejecutar `python manage.py runserver` en la consola para correr el servidor web. Ve al navegador y escribe la dirección <http://127.0.0.1:8000/admin/>. Verás una página de inicio de sesión como la que sigue:



Para iniciar sesión, deberás crear un *superusuario*, que es un usuario que tiene control sobre todo el sitio. Vuelve a la línea de comandos, escribe `python manage.py createsuperuser` y pulsa `intro`.

Recuerda, para escribir nuevos comandos mientras el servidor web se está ejecutando, abre una nueva terminal y activa tu virtualenv. Vimos como escribir nuevos comandos en el capítulo ¡Tu primer proyecto Django!, en la sección Inicial el servidor web.

Cuando te lo pida, escribe tu nombre de usuario (en minúscula, sin espacios), email y contraseña. No te preocupes si no puedes ver la contraseña mientras la escribes, así es como se supone que debe ser. Simplemente escríbela y pulsa `intro` para continuar. La salida de este comando debería verse así (nombre de usuario y dirección de correo electrónico deberían ser los tuyos):

Terminal

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

Vuelve al navegador. Inicia sesión con las credenciales de súper usuario que elegiste; deberías ver el panel de administración de Django.

Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION		
Groups	+ Add	Change
Users	+ Add	Change
BLOG		
Posts	+ Add	Change

Ve a Posts y experimenta un poco con estos. Agrega cinco o seis Post al blog. No te preocupes por el contenido, puedes copiar y pegar cualquier texto de este tutorial para ahorrar tiempo :).

Asegúrate que al menos dos o tres entradas (pero no todas) tengan la fecha de publicación. Será de ayuda más tarde.

Django administration

WELCOME, KOJO. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home > Blog > Posts > Add post

Add post

Author: kojo

Title:

Text:

Created date: Date: 2015-12-25 Today Time: 20:50.01 Now

Published date: Date: Today Time: Now

Si quieres saber más sobre el administrador de Django, puedes visitar la documentación de Django:
<https://docs.djangoproject.com/en/1.10/ref/contrib/admin/>

Este probablemente sea un buen momento para tomar un café (o té) o algo para comer y reenergizarte. Crea este primer modelo de Django - ¡mereces un pequeño recreo!

¡Desplegar!

Nota El siguiente capítulo puede ser, a veces, un poco difícil de seguir. 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 guía 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 computadora, ¡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 de servidores disponibles en Internet. Vamos a utilizar uno que tiene un proceso de despliegue relativamente sencillo: [PythonAnywhere](#). PythonAnywhere es gratis para pequeñas aplicaciones que no tienen demasiados visitantes, lo que es 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!

Estos tres lugares serán importantes. Tu computadora local será el lugar de desarrollo y pruebas. Cuando estés contenta con los cambios, pondrás una copia de tu programa en Github. Tu sitio web estará en PythonAnywhere y lo actualizarás obteniendo una nueva copia del código desde GitHub.

Git

Nota Si ya has hecho los pasos de instalación, no hace falta que hagas esto otra vez. Puedes avanzar a la siguiente sección y empezar a crear tu repositorio Git.

Git es un "sistema de control de versiones" que utilizan muchas programadoras y programadores. Este software puede rastrear los cambios realizados en archivos a lo largo del tiempo de forma que puedas recuperar una versión específica más tarde. Es un poco parecido a la opción de "control de cambios" de Microsoft Word, pero mucho más potente.

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 están bien. "Checkout Windows-style, commit Unix-style line endings" también está bien.

OS X

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

Nota Si estás en OS X 10.6, 10.7 o 10.8, necesitarás instalar la versión de Git desde aquí: [instalador Git para OS X Snow Leopard](#)

Debian o Ubuntu

Terminal

```
$ sudo apt-get install git
```

Fedora (hasta 21)

Terminal

```
$ sudo yum install git
```

Fedora 22+

Terminal

```
$ sudo dnf install git
```

openSUSE

Terminal

```
$ sudo zypper install git
```

Iniciar nuestro repositorio Git

Git rastrea los cambios realizados a un grupo determinado de archivos en lo que llamamos un repositorio de código (abreviado "repo"). 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`.

Terminal

```
$ git init  
Initialized empty Git repository in ~/djangogirls/.git/  
$ git config --global user.name "Your Name"  
$ git config --global user.email you@example.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 archivos y carpetas en este directorio, pero hay algunos archivos que queremos que ignore. Esto lo hacemos creando un archivo llamado `.gitignore` en el directorio base. Abre tu editor y crea un nuevo archivo con el siguiente contenido:

`.gitignore`

```
* .pyc  
*~  
__pycache__  
myvenv  
db.sqlite3  
/static  
.DS_Store
```

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

Nota ¡El punto al principio del nombre del archivo 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.

Nota Uno de los archivos especificados en tu `.gitignore` es `db.sqlite3`. Este archivo es tu base de datos local, donde todos los posts son guardados. No queremos agregarlo al repositorio ya que tu sitio web de PythonAnywhere usará una base de datos diferente. Esta base de datos puede ser SQLite, como en nuestra computadora de desarrollo, pero normalmente usarás algo llamado MySQL que permite manejar muchos más visitantes que SQLite. De cualquier forma, al ignorar tu base de datos SQLite para la copia de GitHub, significa que todos los post que crees van a estar disponibles de forma local únicamente, y tendrás que agregarlos manualmente de nuevo en producción. Debes pensar en tu base de datos local como un buen lugar donde jugar y probar diferentes cosas sin miedo de que puedas borrar los posts reales de tu blog.

Es una buena idea utilizar el comando `git status` antes de `git add` o en cualquier momento en que no estés segura de lo que ha cambiado. Esto ayudará a evitar cualquier sorpresa, como agregar o hacer commit de archivos equivocados. El comando `git status` devuelve información sobre los archivos sin seguimiento ("untracked"), modificados, preparados ("staged"), el estado de la rama y mucho más. La salida debería ser similar a:

Terminal

```
$ 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 tu consola y ejecuta estos comandos:

Terminal

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

Enviar nuestro código a GitHub

Ve a [GitHub.com](#) y registra una nueva cuenta gratuita. (Si ya lo hiciste en la preparación del taller, ¡genial!)

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.

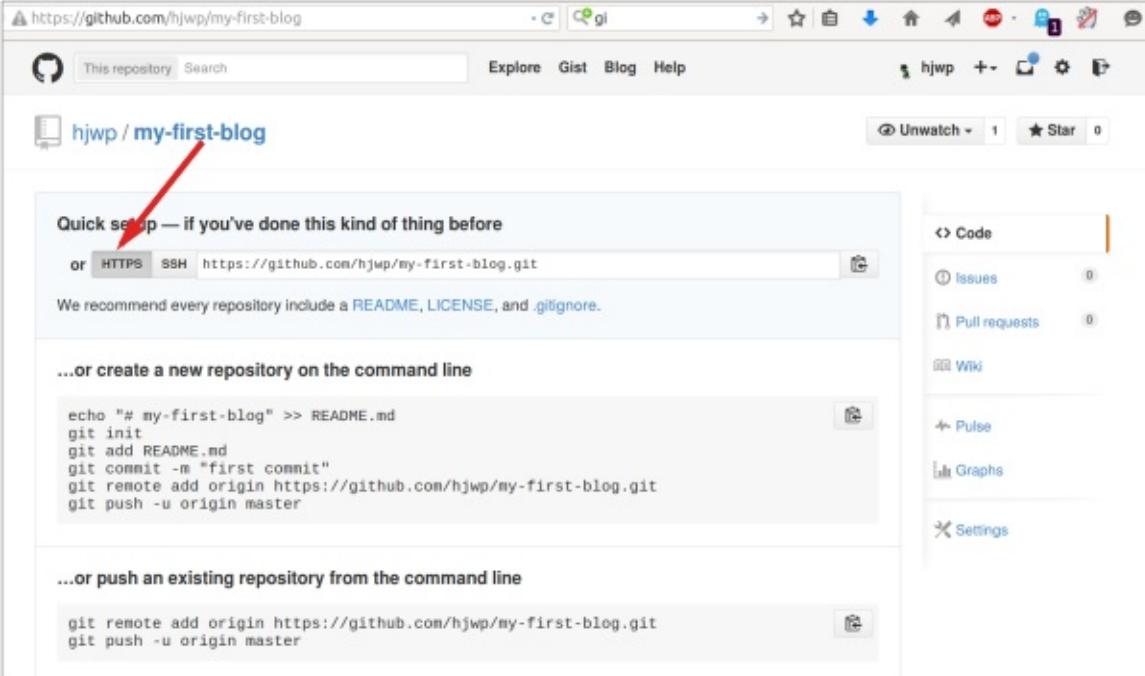
Add .gitignore: **None** Add a license: **None** ⓘ

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 tendrás 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 + ⌂ ⓘ

hjwp / my-first-blog Unwatch 1 Star 0

Quick setup — if you've done this kind of thing before
 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 Pull requests Wiki Pulse Graphs Settings

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

Escribe lo siguiente en la consola (sustituye `<tu-usuario-github>` por el nombre de usuario que elegiste al crear tu cuenta de GitHub, pero sin los signos de mayor y menor):

Terminal

```
$ 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:

Terminal

```
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

Nota Puede que ya hayas creado una cuenta en PythonAnywhere durante los paso de instalación. Si es así, no necesitas hacerlo de nuevo.

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 elige tu propio apodo o bien un nombre que describa sobre qué trata tu blog.

Cargar 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". Esta 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 computadora.

Vamos a descargar nuestro código de GitHub a PythonAnywhere mediante la creación de un "clon" del repositorio. Escribe lo siguiente en la consola de PythonAnywhere (no te olvides de utilizar tu nombre de usuario de GitHub en lugar de <tu-usuario-github>):

Terminal PythonAnywhere

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

Esto va a descargar una copia de tu código en PythonAnywhere. Compruébalo escribiendo `tree my-first-blog` :

Terminal PythonAnywhere

```
$ 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 (entorno virtual) en PythonAnywhere

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

Terminal PythonAnywhere

```
$ cd my-first-blog

$ virtualenv --python=python3.5 myvenv
Running virtualenv with interpreter /usr/bin/python3.5
[...]
Installing setuptools, pip...done.

$ source myvenv/bin/activate

(myvenv) $ pip install django~=1.10.0
Collecting django
[...]
Successfully installed django-1.10.4
```

Nota El paso `pip install` puede llevar un par de minutos. ¡Paciencia, paciencia! Pero si tarda más de 5 minutos, algo va mal. Pregunta a tu tutora o tutor.

Crear la base de datos en PythonAnywhere

Aquí hay otra cosa que es diferente entre tu computadora 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 computadora.

Podemos inicializar la base de datos en el servidor igual que lo hicimos en nuestra computadora, con `migrate` y `createsuperuser`:

Terminal PythonAnywhere

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

Publicar nuestro blog como una aplicación web

Ahora nuestro código está en PythonAnywhere, el virtualenv está listo 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 y haz clic en la pestaña **Web**. Por último, pincha en **Add a new web app**.

Después de confirmar tu nombre de dominio, elige **manual configuration** o "configuración manual" (N.B. – la opción "Django" *no*) en el diálogo. Luego elige **Python 3.5** 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.

edith.pythonanywhere.com

Actions:

Code:
What your site is running.

Source code: You can use the [Files tab](#) to navigate to your app's source code.

WSGI configuration file: `/var/www/edith_pythonanywhere_com_wsgi.py`

Python version: 3.4

Virtualenv:
Use a virtualenv to get different versions of flask, django etc from our default system ones. More info [here](#). You need to Reload your web app to activate it; NB - we do nothing if the virtualenv does not exist.

`/home/edith/my-first-blog/myvenv`

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/<tu-usuario>/my-first-blog/myvenv/`. Haz clic en el cuadro azul seleccionado para guardar la ruta antes de continuar.

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

Configurar el archivo WSGI

Django funciona utilizando el "protocolo WSGI", un estándar para servir sitios web que usan Python, el cual es soportado por PythonAnywhere. La forma de configurar PythonAnywhere para que reconozca nuestro blog Django es editando un archivo 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.

Borra todo el contenido y reemplázalo con algo como esto:

`<your-username>_pythonanywhere_com_wsgi.py`

```
import os
import sys

path = '/home/<your-PythonAnywhere-username>/my-first-blog' # use your own PythonAnywhere
username here
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 django.contrib.staticfiles.handlers import StaticFilesHandler
application = StaticFilesHandler(get_wsgi_application())
```

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

Nota En la línea cuatro, asegúrate de que PythonAnywhere sepa como encontrar tu aplicación.

Es muy importante que esa ruta sea correcta, y especialmente que no haya espacios extras.

De otra forma, verás un "ImportError" en el registro de errores.

Este archivo se encarga de decirle a PythonAnywhere donde vive nuestra aplicación web y como se llama el archivo de configuración de Django.

El `StaticFilesHandler` es para manejar nuestro CSS. Esto lo hace automáticamente en nuestro desarrollo local el comando `runserver`. Encontraremos un poquito más sobre los archivos estáticos luego en este tutorial, cuando editemos los archivos CSS para nuestro sitio.

Haz click en **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 ves un error cuando intentas visitar tu página, el primer lugar donde buscar información de depuración es el **registro de errores**. Encontrarás un enlace a él en la [pestaña Web](#) de PythonAnywhere. Mira 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, 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`?
- ¿Has elegido la misma versión de Python para el `virtualenv` y para la aplicación web? Ambas deberían ser 3.5.

Hay algunos [consejos generales de depuración](#) en el wiki de Pythonanywhere

Y recuerda, ¡tu tutora está ahí para ayudarte!

¡Estás en vivo!

La página predeterminada para su sitio web debe decir "Bienvenido a Django", al igual que lo hace en su computadora. Intenta agregar `/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 agregar nuevas entradas en el servidor.

Una vez que hayas creado algunos posts en tu blog, puedes volver a tu instancia local (no la de PythonAnywhere). Desde ahí podrías trabajar en tu instancia local para hacer algunos cambios. Estos son los pasos comunes en el desarrollo web (hacer cambios locales, enviarlos a GitHub, descargarlos en el servidor web). Esto te permite trabajar y experimentar sin romper tu sitio web en vivo. Muy bueno, ¿no?

¡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!

URLs en Django

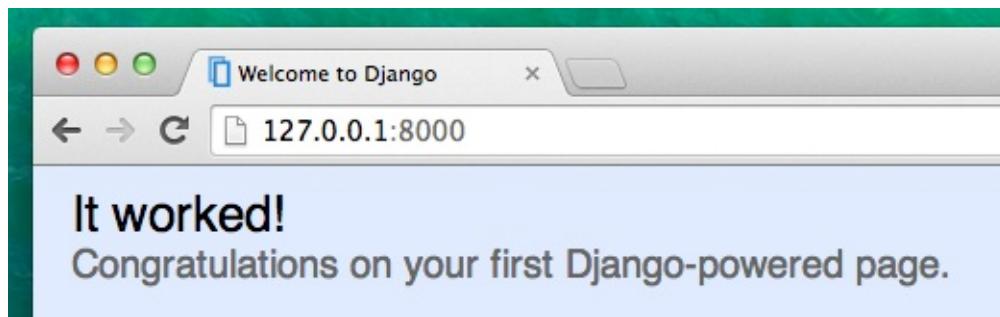
Estamos a punto de construir nuestra primera página web: ¡una página de inicio para el blog! Pero primero, vamos a aprender un poco acerca de las urls de Django.

¿Qué es una URL?

Una URL es simplemente una dirección web. Puedes ver una URL cada vez que visitas una página.

Se ve en la barra de direcciones del navegador (¡sí! ¡ 127.0.0.1:8000 es una URL! Y

`https://djangogirls.com` también es 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 utilizamos algo que se llama `URLconf` (configuración de URL). URLconf es un conjunto de patrones que Django intentará comparar con la URL recibida para encontrar la vista correcta.

¿Cómo funcionan las URLs en Django?

Vamos a abrir el archivo `mysite/urls.py` en el editor de código de tu elección y veamos lo que tiene:

`mysite/urls.py`

```
"""mysite URL Configuration

[...]
"""

from django.conf.urls import url
from django.contrib import admin

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

Como puedes ver, Django ya puso algo aquí por nosotras.

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

La URL del admin, que visitaste en el capítulo anterior, ya está aquí:

mysite/urls.py

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

Esto significa que para cada URL que empieza con `admin/` Django encontrará su correspondiente `view`. En este caso estamos incluyendo muchas URLs del admin así que está todo empaquetado en un pequeño archivo. Es más limpio y legible.

Regex

¿Te preguntas cómo compara Django las direcciones URL con las vistas? Bueno, esta parte es complicada. Django usa `regex`, abreviatura de "expresiones regulares". Regex tiene muchas normas (¡un montón!) que forman un patrón de búsqueda. Como las regex son un tema avanzado, no vamos a entrar en detalle de cómo funcionan.

Si aún así quieras entender cómo hemos creado los patrones, aquí hay un ejemplo del proceso. Sólo necesitaremos un grupo limitado de reglas para expresar el patrón que estamos buscando, en concreto:

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

Cualquier otra cosa en la definición de la 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 de la 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 de la URL, lo que significa que sólo coincidirán con este patrón las cadenas que terminen en `/`

¡Tu primera URL de Django!

¡Es hora de crear nuestra primera URL! Queremos que '<http://127.0.0.1:8000/>' sea la página de inicio del blog y que muestre una lista de entradas.

También queremos mantener limpio el archivo `mysite/urls.py`, así que vamos a importar las URLs de nuestra aplicación `blog` en el archivo principal `mysite/urls.py`.

Adelante, agrega la línea que importará `blog.urls`. Nota que estamos usando la función `include` aquí así que **tendrás que agregar eso a la primera línea de import del archivo**.

El archivo `mysite/urls.py` debería verse ahora así:

`mysite/urls.py`

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

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

Ahora Django redirigirá todo lo que entre a '<http://127.0.0.1:8000/>' hacia `blog.urls` y buscará más instrucciones allí.

Cuando se escriben expresiones regulares en Python siempre se pone `r` delante de la cadena. Esto le da una pista a Python de que la cadena puede contener caracteres especiales que no son para Python en sí, sino para la expresión regular.

blog.urls

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

`blog/urls.py`

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

Aquí sólo estamos importando la función `url` de Django y todas nuestras `vistas` de la aplicación `blog` (aún no tenemos ninguna, ¡pero llegaremos a eso en un minuto!)

Luego de esto, podemos agregar nuestro primer patrón URL:

`blog/urls.py`

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

]

Como puedes ver, ahora estamos asignando una vista `view` llamada `post_list` a la URL `^$`. Esta expresión regular coincidirá con `^` (un inicio) seguido de `$` (un final), por lo tanto sólo una cadena vacía coincidirá. Es correcto, porque en el sistema de resolución de URL de Django, '<http://127.0.0.1:8000/>' no forma parte de la URL. Este patrón le dirá a Django que `views.post_list` es el lugar correcto al que ir si alguien entra a tu sitio web con la dirección '<http://127.0.0.1:8000/>'.

La última parte `name='post_list'` es el nombre de la URL que se utilizará para identificar a la vista. Puede coincidir con el nombre de la vista pero también puede ser algo completamente distinto. Utilizaremos las URL con nombre más delante en el proyecto así que es importante darle un nombre a cada URL de la aplicación. También deberíamos intentar mantener los nombres de las URL únicos y fáciles de recordar.

Si intentas visitar <http://127.0.0.1:8000/> ahora, encontrarás una especie de mensaje diciendo 'página web no disponible'. Esto es porque el servidor (recordaste escribir `runserver`?) no está ejecutándose más. Mira en la ventana de la consola para ver porqué.

```
return _bootstrap._gcd_import(name[level:], package, level)
File "<frozen importlib._bootstrap>", line 2254, in _gcd_import
File "<frozen importlib._bootstrap>", line 2237, in _find_and_load
File "<frozen importlib._bootstrap>", line 2226, in _find_and_load_unlocked
File "<frozen importlib._bootstrap>", line 1200, in _load_unlocked
File "<frozen importlib._bootstrap>", line 1129, in _exec
File "<frozen importlib._bootstrap>", line 1471, in exec_module
File "<frozen importlib._bootstrap>", line 321, in _call_with_frames_removed
File "/Users/dana/Dana-Files/Codes/djangogirls/blog/urls.py", line 5, in <module>
    url(r'^$', views.post_list, name='post_list'),
AttributeError: 'module' object has no attribute 'post_list'
```

Tu consola está mostrando un error, pero no te preocupes. De hecho, es de mucha ayuda: te está diciendo que no hay ningún **atributo 'post_list'**. Ese es el nombre de la *vista* que Django está tratando de encontrar y usar, pero no la hemos creado aún. En este momento tu `/admin/` tampoco funcionará. No te preocupes, ya llegaremos a eso.

Si quieres saber más sobre Django URLconfs, mira la documentación oficial:
<https://docs.djangoproject.com/en/1.10/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. Pedirá información del `modelo` que has creado antes y se la pasará a la `plantilla`. Crearemos una plantilla en el próximo capítulo. Las vistas son sólo métodos de Python que son un poco más complicados que los que escribimos 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 abrir este archivo y ver lo que contiene:

`blog/views.py`

```
from django.shortcuts import render

# Create your views here.
```

No demasiadas cosas aquí todavía.

Recuerda que las líneas que comienzan con `#` son comentarios. Esto significa que esas líneas no será ejecutadas por Python.

La *view* más simple puede ser como esto:

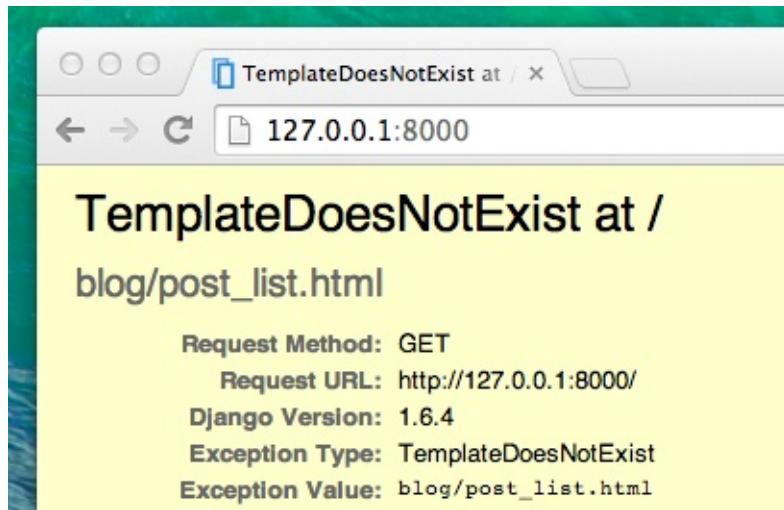
`blog/views.py`

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

Como puedes ver, hemos creado un método (`def`) llamado `post_list` que recibe una petición, `request`, y devuelve, `return`, un método `render` que renderizará (construirá) la plantilla `blog/post_list.html`.

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

¡Otro error! Leamos lo que está pasando ahora:



Este muestra que el servidor está corriendo nuevamente, al menos, pero todavía no se ve del todo bien, ¿no?. No te preocupes, es solo otra página de error, nada de que asustarse. Igualmente que el mensaje de error en la consola, estos también son muy útiles. Puedes leer *TemplateDoesNotExist* en él. ¡Vamos a arreglar este error y crear una plantilla en el siguiente capítulo!

Aprende más sobre las vistas de Django leyendo la documentación oficial:

<https://docs.djangoproject.com/en/1.10/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 marca con código para decirte cómo interpretar la página (en este caso, un navegador). El código HTML está construido con **etiquetas**, cada una comenzando con `<` y terminando con `>`. Estas etiquetas representan **elementos** de marcado.

¡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`:

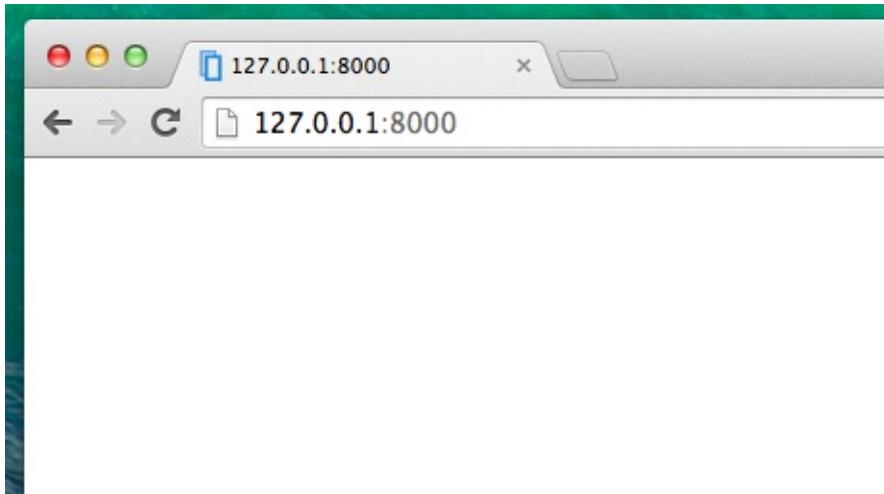
```
blog
└── templates
    └── blog
```

(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`.



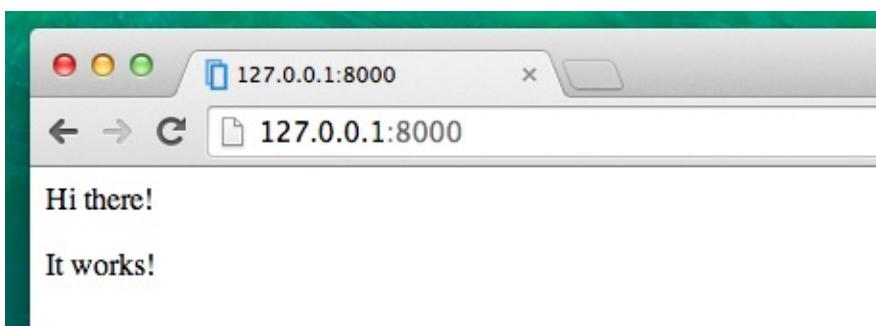
¡Ningún error más! Felicidades :) Sin embargo, por ahora, tu sitio web no está publicando nada excepto una página en blanco, porque la plantilla también está vacía. Tenemos que arreglarlo.

Agrega lo siguiente a tu archivo de plantilla:

blog/templates/blog/post_list.html

```
<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 y 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.

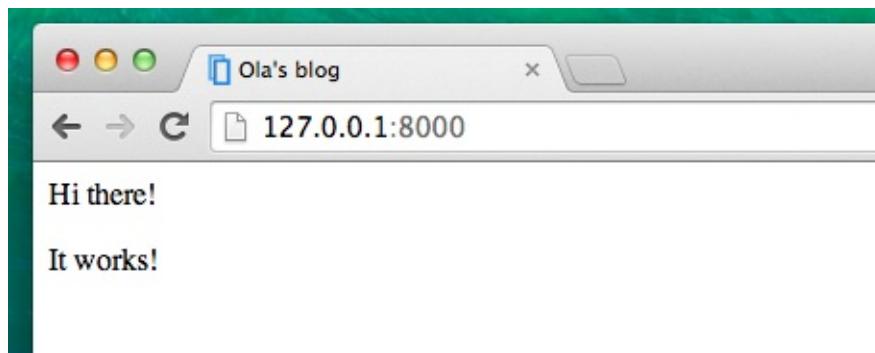
Usamos `<head>` para decirle al navegador acerca de la configuración de la página y `<body>` para decir lo que realmente está en la página.

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

`blog/templates/blog/post_list.html`

```
<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 tambié 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, copia y pégalo en el archivo
`blog/templates/blog/post_list.html`

`blog/templates/blog/post_list.html`

```

<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 vestibulum. Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit amet risus.</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 vestibulum. Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut f.</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: ¡desplegar!

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, vamos a ver qué archivos han cambiado desde la última puesta en marcha (ejecute estos comandos localmente, no en PythonAnywhere):

Terminal

```
$ 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:

Terminal

```
$ git add --all .
```

Nota `--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):

Terminal

```
$ 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.

Terminal

```
$ git commit -m "Changed the HTML for the site."
```

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

Una vez hemos hecho esto, subamos (push) los cambios a Github:

Terminal

```
$ 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:

Terminal

```
$ 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 líneal! Actualiza tu sitio web en el navegador. Ahora deberías poder ver tus cambiosss :)

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 una base de datos, filtrarlos y ordenarlos.

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

Django shell

Abre tu consola local (no la de PythonAnywhere) y escribe este comando:

Terminal

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

El resultado debería ser:

Terminal

```
(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.

Todos los objetos

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

Terminal

```
>>> 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!

Terminal

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

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

Terminal

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

¡Es una lista de posts que hemos creado antes! Las hemos creado 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:

Terminal

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

Pero nos falta un ingrediente aquí: `me`. Necesitamos pasar una instancia del modelo `User` como autor. ¿Eso cómo se hace?

Primero importemos el modelo `User`:

Terminal

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

¿Qué usuarios tenemos en nuestra base de datos? Prueba esto:

Terminal

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

¡Es el súper usuario que hemos creado antes! Vamos a obtener una instancia de ese usuario ahora:

Terminal

```
>>> 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 post:

Terminal

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

¡Hurra! ¿Quieres comprobar si ha funcionado?

Terminal

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

¡Ahí está, una entrada de blog más en la lista!

Agrega más posts

Ahora puedes divertirte un poco y agregar más posts para ver cómo funciona. Agrega 2 o 3 más y avanza a la siguiente parte.

Filtrar objetos

Una parte importante de los QuerySets es la habilidad para filtrar los resultados. Digamos que queremos encontrar todas las entradas cuyo autor sea el User ola. Usaremos `filter` en vez de `all` en `Post.objects.all()`. Entre paréntesis estableceremos qué condición (o condiciones) debe cumplir una entrada del blog para aparecer como resultado 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 tiene este aspecto:

Terminal

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

¿O quizás queremos ver todas las entradas que contengan la palabra 'title' en el campo `title` ?

Terminal

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

Nota Hay dos guiones bajos (`_`) entre `title` y `contains`. El ORM de Django utiliza esta sintaxis para separar los nombres de los campos ("title") de las 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 todas las entradas publicadas. Lo hacemos filtrando las entradas que tienen la fecha de publicación, `published_date`, en el pasado:

Terminal

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

Por desgracia, la entrada que hemos añadido desde la consola de Python no está publicada todavía. ¡Eso lo podemos cambiar! Primero obtenemos una instancia de la entrada que queremos publicar:

Terminal

```
>>> post = Post.objects.get(title="Sample title")
```

¡Y luego públicala con nuestro método `publish`!

Terminal

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

Ahora vuelve a intentar obtener la lista de entradas publicadas (pulsa la tecla de "flecha arriba" 3 veces y pulsa `intro`):

Terminal

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

Ordenar objetos

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

Terminal

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

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

Terminal

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

Encadenar QuerySets

También puedes combinar QuerySets **encadenando** uno con otro:

Terminal

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

Es muy potente y te permite escribir consultas bastante complejas.

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

Terminal

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

Datos dinámicos en plantillas

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: 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 pasarlo 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:

`blog/views.py`

```
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:

`blog/views.py`

```
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 \(QuerySets\)](#).

Así que ahora nos interesa tener una lista de entradas del blog que han sido publicadas y ordenadas por `published_date` (fecha de publicación), ¿no? ¡Ya lo hicimos en el capítulo [QuerySets!](#)

blog/views.py

```
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) :
```

blog/views.py

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

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

Fíjate en que creamos una *variable* para el QuerySet: `posts`. Trátala como si fuera el nombre de nuestro QuerySet. De aquí en adelante vamos a referirnos al QuerySet con ese nombre.

Además, este código utiliza la función `timezone.now()`, así que necesitamos agregar una importación de `timezone`.

En la función `render` ya tenemos el parámetro `request` (todo lo que recibimos del usuario vía Internet) y el archivo `'blog/post_list.html'` como plantilla. El último parámetro, que se ve así: `{}` es un lugar 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}`. Fíjate en que la parte antes de `:` es una cadena; tienes que envolverla con comillas `''`.

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

blog/views.py

```
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('published_date')
    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/1.10/ref/models/querysets/>

Plantillas de Django

¡Es hora de mostrar algunos datos! Para ello Django incorpora unas etiquetas de plantillas, **template tags**, muy útiles.

¿Qué son las template tags?

Verás, en HTML no se puede escribir código Python porque los navegadores no lo entienden. Sólo saben HTML. Sabemos que HTML es bastante estático, mientras que Python es mucho más dinámico.

Las **template tags de Django** nos permiten comunicar elementos de Python a HTML, para que puedas construir sitios web dinámicos más rápido y fácil.

Mostrar la plantilla post list

En el capítulo anterior le dimos a nuestra plantilla una lista de entradas en la variable `posts`. Ahora la vamos a mostrar en HTML.

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

`blog/templates/blog/post_list.html`

```
 {{ posts }}
```

Prueba esto en la plantilla `blog/templates/blog/post_list.html`. Sustituye todo desde el segundo `<div>` al tercer `</div>` por `{{ posts }}` . Guarda el archivo y refresca la página para ver los resultados:



Como puedes ver, lo que hemos conseguido es esto:

`blog/templates/blog/post_list.html`

```
[<Post: My second post>, <Post: My first post>]
```

Esto significa que Django lo entiende como una lista de objetos. ¿Recuerdas de [Introducción a Python](#) cómo podemos mostrar listas? Sí, ¡con bucles for! En una plantilla de Django se hacen así:

blog/templates/blog/post_list.html

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

Prueba esto en tu plantilla.



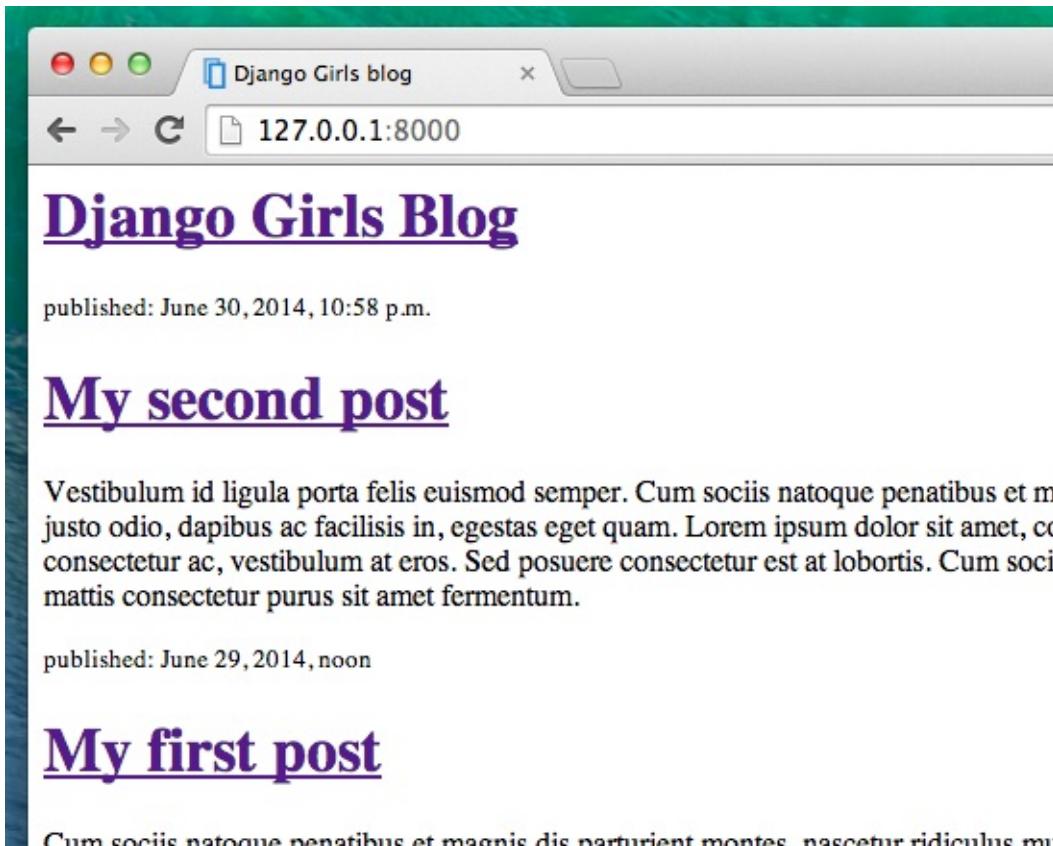
¡Funciona! Pero queremos que se muestren como las entradas de blog estáticas que creamos anteriormente en el capítulo de [Introducción a HTML](#). Puedes mezclar HTML y etiquetas de plantilla. Nuestro `body` se verá así:

blog/templates/blog/post_list.html

```
<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 }}>{{ post.title }}</a></h1>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endfor %}
```

Todo lo que pongas entre `{% for %}` y `{% endfor %}` se repetirá para cada objeto de la lista. Refresca la página:



Una cosa más

Sería bueno ver si tu sitio web seguirá funcionando en la Internet pública, ¿no? Vamos a intentar desplegar de nuevo en PythonAnywhere. Aquí va un resumen de los pasos...

- Primero, sube tu código a Github

Terminal

```
$ git status
[...]
$ git add --all .
$ git status
[...]
$ git commit -m "Modified templates to display posts from database."
[...]
$ git push
```

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

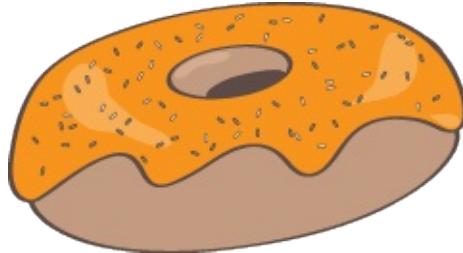
Terminal

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

- Finalmente, ve a la [pestaña Web](#) y presiona **Reload** en tu aplicación web. ¡Tu actualización debería verse! Si las entradas en tu sitio web de PythonAnywhere no coinciden con las entradas del blog de tu servidor local, está bien. La base de datos de tu computadora local y la de PythonAnywhere no se sincronizan con el resto de tus archivos.

¡Felicitades! Ahora sigue adelante y trata de agregar una nueva entrada usando el panel de administrador de Django (¡recuerda agregar `published_date`!). Asegúrate de estar en el Django Admin de tu sitio de PythonAnywhere, <https://yourname.pythonanywhere.com/admin>. Luego actualiza tu página para ver si aparece tu nuevo post.

¿Funciona de maravilla? ¡Estamos orgullosas! Aléjate un rato de la computadora, 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 hecho 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 de HTML y CSS mas populares para desarrollar sitios bonitos:

<http://getbootstrap.com/>

Fue escrito por programadores que trabajaban para Twitter y ahora está siendo desarrollado por voluntarios de todo el mundo.

Instalar Bootstrap

Para instalar Bootstrap, tienes que agregar esto al `<head>` en tu archivo `.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 agrega archivos a tu proyecto, sólo hace referencia a archivos que existen en internet.

Adelante, abre tu sitio y actualiza la página. ¡Ahí lo tienes!



¡Se ve mucho mejor!

Archivos estáticos en Django

Finalmente nos vamos a fijar en estas cosas que hemos estado llamando **archivos 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 archivos estáticos para Django

Django ya sabe dónde encontrar los archivos estáticos para la aplicación "admin". Ahora necesitamos agregar algunos archivos estáticos para nuestra propia aplicación, `blog`.

Hacemos esto creando una carpeta llamada `static` dentro de la estructura de la aplicación que llamamos `blog`:

```
djangogirls
├── 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 archivos estáticos.

¡Tu primer archivo CSS!

Ahora creamos un archivo CSS para agregar tu propio estilo a tu página web. Crea un nuevo directorio llamado `css` dentro de tu directorio `static`. Despues crea un nuevo archivo llamado `blog.css` dentro de este directorio `css`. ¿Lista?

```
djangogirls
└── blog
    └── static
        └── css
            └── blog.css
```

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

No nos adentraremos mucho en la personalización y aprendizaje sobre CSS aquí. Es muy fácil y lo puedes aprender por tu cuenta después de este taller. Hay una recomendación sobre un curso gratuito para aprender más al final de esta página.

Pero vamos a hacer un poco al menos. ¿Tal vez podríamos cambiar el color de nuestro título? Las computadoras utilizan códigos especiales para entender los colores. Estos códigos empiezan con `#` y les siguen 6 letras (A-F) y números (0-9). Por ejemplo, el color azul es `#0000FF`. Puedes encontrar estos códigos para muchos colores aquí: <http://www.colorpicker.com/>. También puedes utilizar [colores predefinidos](#) utilizando su nombre en inglés, como `red` y `green`.

En tu archivo `blog/static/css/blog.css` deberías agregar el siguiente código:

`blog/static/css/blog.css`

```
h1 a {
    color: #FCA205;
}
```

`h1 a` es un selector CSS. Esto significa que estamos aplicando nuestros estilos a cualquier elemento `a` dentro de un elemento `h1` (por ejemplo cuando tenemos en código como: `<h1>link</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 el archivo CSS se definen los estilos de los elementos que se encuentran en el archivo HTML. Los elementos se identifican por el nombre del elemento (por ejemplo `a` . `h1` , `body`), el atributo `class` o el atributo `id` . "class" y "id" son nombres que le asignas tú misma al elemento. Las "class" definen grupos de elementos y los "id" apuntan a elementos específicos. Por ejemplo, la siguiente etiqueta puede identificarse mediante CSS usando el nombre de etiqueta `a` , la clase `external_link` o el id `link_to_wiki_page` :

```
<a href="https://en.wikipedia.org/wiki/Django" class="external_link" id="link_to_wiki_page">
```

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

También necesitamos decirle a nuestra plantilla HTML que hemos agregado algunos CSS. Abre el archivo `blog/templates/blog/post_list.html` y agrega esta línea al principio del todo:

`blog/templates/blog/post_list.html`

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

Aquí sólo estamos cargando archivos estáticos :). Luego, entre las etiquetas `<head>` y `</head>`, después de los enlaces a los archivos CSS de Bootstrap, agrega la siguiente línea:

blog/templates/blog/post_list.html

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

El navegador lee los archivos en el orden que aparecen, de ese modo el código en nuestro archivo puede sobrescribir el código en los archivos de Bootstrap. Le acabamos de decir a nuestra plantilla dónde se encuentra nuestro archivo CSS.

Ahora tu archivo se debe ver así:

blog/templates/blog/post_list.html

```
{% load staticfiles %}

<html>
  <head>
    <title>Django Girls blog</title>
    <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">
      <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|linebreaksbr }}</p>
      </div>
    {% endfor %}
  </body>
</html>
```

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

The screenshot shows a Mac OS X style browser window titled "Django Girls blog". The address bar shows "127.0.0.1:8000". The page content includes the title "Django Girls Blog" in large orange font, a timestamp "published: June 30, 2014, 10:58 p.m.", and a post titled "My second post" with the text "Vestibulum id ligula porta felis euismod semper. Cum sociis natoque".

published: June 30, 2014, 10:58 p.m.

My second post

Vestibulum id ligula porta felis euismod semper. Cum sociis natoque

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

blog/static/css/blog.css

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

Agrega esto a tu CSS, guarda el archivo y ¡mira cómo funciona!

The screenshot shows the same browser window as before, but now the text "Vestibulum id ligula porta felis euismod semper. Cum sociis natoque" is indented by 15 pixels to the right, demonstrating the effect of the CSS rule added in the previous step.

Django Girls Blog

published: June 30, 2014, 10:58 p.m.

My second post

Vestibulum id ligula porta felis euismod semper. Cum sociis natoque

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

`blog/templates/blog/post_list.html`

```
<link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext" rel="stylesheet" type="text/css">
```

```
<head>
  <link href="https://fonts.googleapis.com/css?family=Lobster" type="text/css" rel="stylesheet">
```

Como antes, comprueba el orden y ubícalo antes del link al archivo `blog/static/css/blog.css`. Esta línea va a importar una tipografía llamada *Lobster* desde Google Fonts (<https://www.google.com/fonts>).

Busca la declaración `h1 a` (el código entre llaves `{ } y }`) en el archivo CSS `blog/static/css/blog.css`. Ahora agrega la línea `font-family: 'Lobster';` entre las llaves y refresca la página:

`blog/static/css/blog.css`

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



¡Genial!

Como mencionamos 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. ¡Esto puede ser muy útil!. Quizás tienes dos divs que hacen algo diferente (como el encabezado y el post). Una clase (`class`) puede ayudarte a hacer que se vean diferentes.

Adelante, nombra algunas partes del código HTML. Agrega una clase llamada `page-header` a tu `div` que contiene el encabezado, así:

`blog/templates/blog/post_list.html`

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

Y ahora agrega la clase `post` a tu `div` que contiene una entrada del blog.

blog/templates/blog/post_list.html

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

Ahora agregaremos bloques de declaración a diferentes selectores. Los selectores que comienzan con `.` hacen referencia a clases. Hay muchos tutoriales geniales y explicaciones sobre CSS en la Web para ayudarte a entender el siguiente código. Por ahora, simplemente copia y pega este bloque de código en tu archivo `blog/static/css/blog.css`:

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 {
    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 envuelve el código HTML que muestra los posts con declaraciones de clases. Reemplaza esto:

blog/templates/blog/post_list.html

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

en el archivo blog/templates/blog/post_list.html por esto:

blog/templates/blog/post_list.html

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

Guarda estos archivos y actualiza tu sitio.



¡Woohoo! Se ve genial, ¿verdad? Mira el código código que acabamos y busca los lugares dónde hemos agregado clases en el HTML y úsalas en el CSS. ¿Qué cambio deberías hacer para que la fecha se vea en turquesa?

No tengas miedo de jugar un poco con este CSS e intentar cambiar algunas cosas. Jugar con el código CSS te puede ayudar a entender qué hacen las diferentes cosas. Si rompes algo, no te preocupes, ¡siempre puedes deshacerlo!

Recomendamos encarecidamente que hagas este [Curso de HTML & CSS de Codecademy](#) gratuito y online. Te puede ayudar a aprender 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.

Las plantillas son útiles cuando quieres usar una misma información o un mismo esquema en más de un lugar. De esta forma no tienes que repetir el código en cada uno de los archivos y si quieres cambiar algo, no necesitas hacerlo en cada plantilla sino, ¡solo en una!

Crea la 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:

`blog/templates/blog/base.html`

```
{% load staticfiles %}

<html>
    <head>
        <title>Django Girls blog</title>
        <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">
        <link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'>
        <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|linebreaksbr }}</p>
    </div>
    {% endfor %}
</div>
</body>
</html>
```

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

`blog/templates/blog/base.html`

```

<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>
```

Quizás notaste que esto reemplaza todo lo de `{% for post in posts %}` por `{% endfor %}` con:

`blog/templates/blog/base.html`

```

{% block content %}
{% endblock %}
```

¿Pero porqué? Acabas de crear un `block` ! Usamos el template tag `{% block %}` para crear un área en dónde luego insertado un HTML. Ese HTML vendrá desde otra plantilla que extienda esta plantilla (`base.html`). Te mostraremos como hacer esto en un momento.

Ahora guarda `base.html` y abre tu archivo `blog/templates/blog/post_list.html` de nuevo. Elimina todo lo que está por encima de `{% for post in posts %}` y por debajo de `{% endfor %}` . Cuando lo hagas, el archivo se verá así:

`blog/templates/blog/post_list.html`

```

{% for post in posts %}
    <div class="post">
```

```

<div class="date">
    {{ post.published_date }}
</div>
<h1><a href="">{{ post.title }}</a></h1>
<p>{{ post.text|linebreaksbr }}</p>
</div>
{% endfor %}

```

Queremos utilizar esto como parte de nuestra plantilla para todos los bloques *content*. ¡Es hora de agregar una etiqueta `block` a este archivo!

Quieres que tu etiqueta `block` coincida con la etiqueta en tu archivo `base.html`. También querrás que incluya todo el código que pertenece al bloque *content*. Para hacer eso, pon todo entre `{% block content %}` y `{% endblock content %}`. Algo así:

`blog/templates/blog/post_list.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|linebreaksbr }}</p>
    </div>
{% endfor %}
{% endblock %}

```

Solo nos falta una cosa. Necesitamos conectar estas dos plantillas. ¡De eso se trata extender plantillas! Necesitamos hacer esto agregando una etiqueta al principio del archivo. Así:

`blog/templates/blog/post_list.html`

```

{% 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|linebreaksbr }}</p>
    </div>
{% endfor %}
{% endblock %}

```

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

Si obtienes un error `TemplateDoesNotExist`, significa 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`.

Extender 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 agregar nada a `models.py`.

Crear un enlace al detalle de una entrada

Vamos a empezar añadiendo un enlace dentro del archivo `blog/templates/blog/post_list.html`. Hasta el momento debería verse así:

`blog/templates/blog/post_list.html`

```
{% 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|linebreaksbr }}</p>
        </div>
    {% endfor %}
    {% endblock content %}
```

Queremos tener un enlace que vaya desde el título de la entrada en la lista de entradas hasta la página de detalle de la entrada. Vamos a cambiar `<h1>{{ post.title }}</h1>` para que enlace a la página de detalle de la entrada:

`blog/templates/blog/post_list.html`

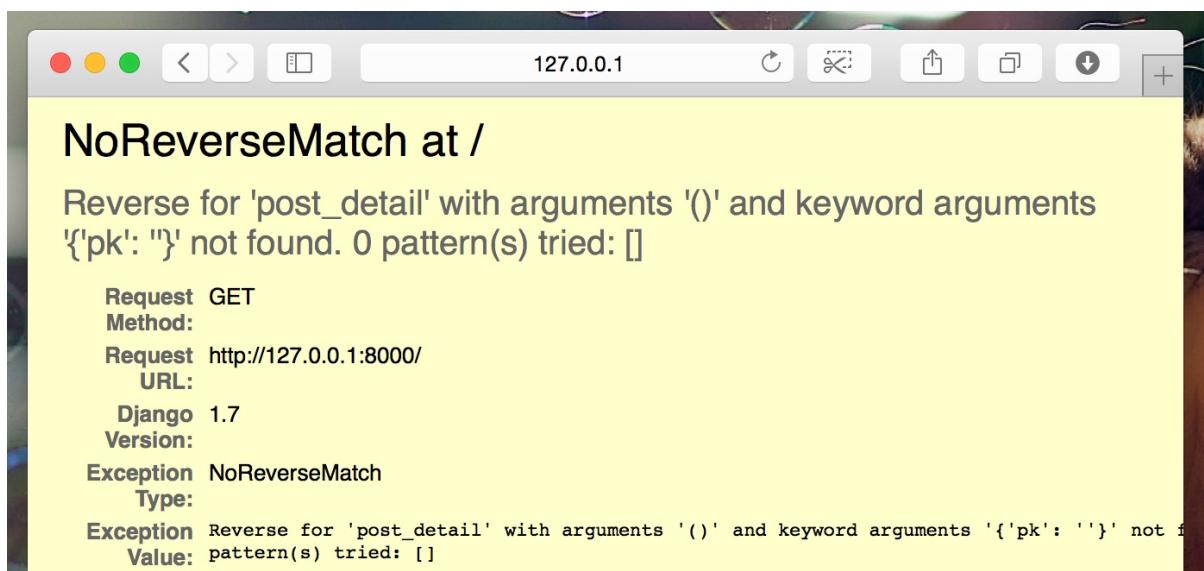
```
<h1><a href="{% url 'post_detail' pk=post.pk %}">{{ post.title }}</a></h1>
```

Es hora de explicar el misterioso `{% url 'post_detail' pk=post.pk %}`. Como probablemente sospeches, la notación `{% %}` significa que estamos utilizando Django template tags. ¡Esta vez vamos a usar una que creará una URL!

`blog.views.post_detail` es una ruta hacia la *vista* `post_detail` que queremos crear. Fíjate bien: `blog` es el nombre de nuestra aplicación (el directorio `blog`), `views` es el nombre del archivo `views.py` y la última parte, `post_detail`, es el nombre de la *vista*.

¿Y, qué hay de `pk=post.pk`? `pk` es la abreviatura de `primary key`, clave primaria, el cuál es un nombre único para cada entrada en la base de datos. Como no hemos especificado ninguno en nuestro modelo `Post`, Django crea uno por nosotros (por defecto, un número que incrementa con cada entrada, eso es 1, 2, 3) y agrega este como un campo llamado `pk` para cada uno de nuestros posts. Podemos acceder a la clave primera escribiendo `post.pk`, de la misma forma que accedemos a otros campos (`title`, `author`, etc.) en nuestro objeto `Post`.

Ahora cuando vayamos a: <http://127.0.0.1:8000/> tendremos un error (como era de esperar, ya que no tenemos una URL o una *vista* para `post_detail`). Se verá así:



Crea una URL al detalle de una entrada

Vamos a crear una URL en `urls.py` para nuestra *vista* `post_detail`!

Queremos que el detalle de la primera entrada se visualice en esta URL: <http://127.0.0.1:8000/post/1/>

Vamos a crear una URL en el archivo `blog/urls.py` que dirija a Django hacia una *vista* llamada `post_detail`, que mostrará una entrada de blog completa. Añade la línea `url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail, name='post_detail')`, al archivo `blog/urls.py`. El archivo debería parecerse a esto:

`blog/urls.py`

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

urlpatterns = [
    url(r'^$', views.post_list, name='post_list'),
    url(r'^post/(?P<pk>\d+)/$', views.post_detail, name='post_detail'),
```

]

La parte de `^post/(?P<pk>[0-9]+)/$` da un poco de miedo, pero no te preocupes, te la vamos a explicar:

- comienza con `^` otra vez, "el principio".
- `post/` quiere decir que después del comienzo, la URL debe contener la palabra `post` y `/`. Por ahora todo bien.
- `(?P<pk>[0-9]+)` - esta parte es más complicada. Significa que Django llevará todo lo que pongas aquí y lo transferirá a una vista como una variable llamada `pk`. `[0-9]` también nos dice que sólo puede ser un número, no una letra (todo lo que esté entre 0 y 9). `+` significa que tiene que haber uno o más dígitos. Así que algo como `http://127.0.0.1:8000/post//` no es válido, pero `http://127.0.0.1:8000/post/1234567890/` es perfectamente aceptable!
- `/` - necesitamos `/` de nuevo
- `$` - ¡"el final"!

Esto quiere decir que si pones `http://127.0.0.1:8000/post/5/` en tu navegador, Django entenderá que estás buscando una *vista* llamada `post_detail` y transferirá la información de que `pk` es igual a `5` a esa *vista*.

Bien, ¡hemos añadido un nuevo patrón de URL a `blog/urls.py`! Vamos a refrescar la página:
<http://127.0.0.1:8000/> ¡Zas! ¡Otro error! ¡Era de esperar!

```
return _bootstrap._gcd_import(name[level:], package, level)
File "<frozen importlib._bootstrap>", line 2231, in _gcd_import
File "<frozen importlib._bootstrap>", line 2214, in _find_and_load
File "<frozen importlib._bootstrap>", line 2203, in _find_and_load_unlocked
File "<frozen importlib._bootstrap>", line 1200, in _load_unlocked
File "<frozen importlib._bootstrap>", line 1129, in _exec
File "<frozen importlib._bootstrap>", line 1448, in _exec_module
File "<frozen importlib._bootstrap>", line 321, in _call_with_frames_removed
File "/home/hel/code/djangogirls/workthrough/blog/urls.py", line 6, in <module>
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail, name='post_detail'),
AttributeError: 'module' object has no attribute 'post_detail'
[]
```

¿Recuerdas cuál es el próximo paso? Por supuesto: ¡agregar una vista!

Agregar una vista de detalle de la entrada

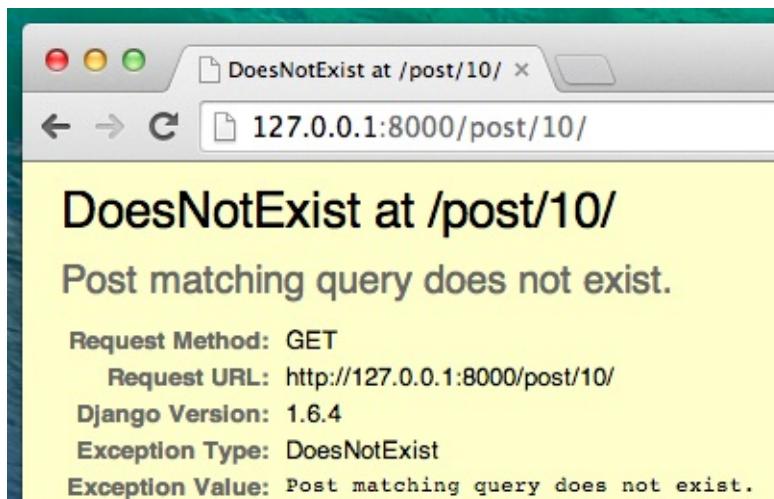
Esta vez nuestra *vista* tomará un parámetro adicional `pk`. Nuestra *vista* necesita recibirla, ¿verdad? Así que definiremos nuestra función como `def post_detail (request, pk):`. Ten en cuenta que tenemos que usar exactamente el mismo nombre que especificamos en las urls (`pk`). ¡Omitir esta variable es incorrecto y resultará en un error!

Ahora, queremos obtener una sola entrada del blog. Para ello podemos usar querysets como este:

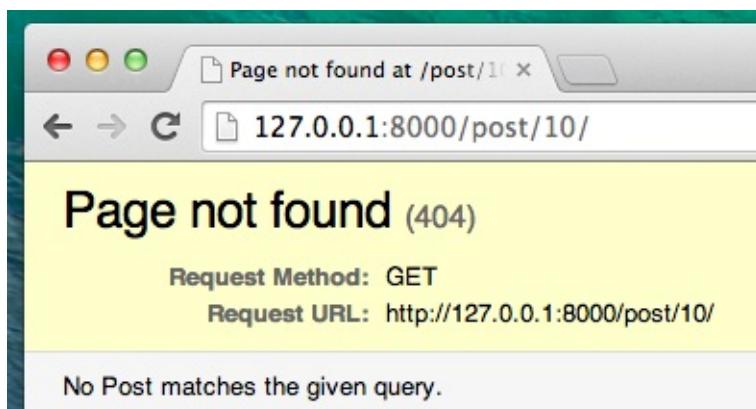
`blog/views.py`

```
Post.objects.get(pk=pk)
```

Pero este código tiene un problema. Si no hay ningún `Post` con esa clave primaria (`pk`), ¡tendremos un error muy feo!



¡No queremos eso! Pero, por supuesto, Django viene con algo que se encargará de ese problema por nosotros: `get_object_or_404`. En caso de que no haya ningún `Post` con el dato `pk` se mostrará una más agradable página (`Page Not Found 404`).



La buena noticia es que puedes crear tu propia página `Page Not Found` y diseñarla como deseas. Pero por ahora no es tan importante, así que lo omitiremos.

¡Es hora de agregar una `view` a nuestro archivo `views.py` !

Deberíamos abrir `blog/views.py` y agregar el siguiente código debajo de las otras líneas `from :`
`blog/views.py`

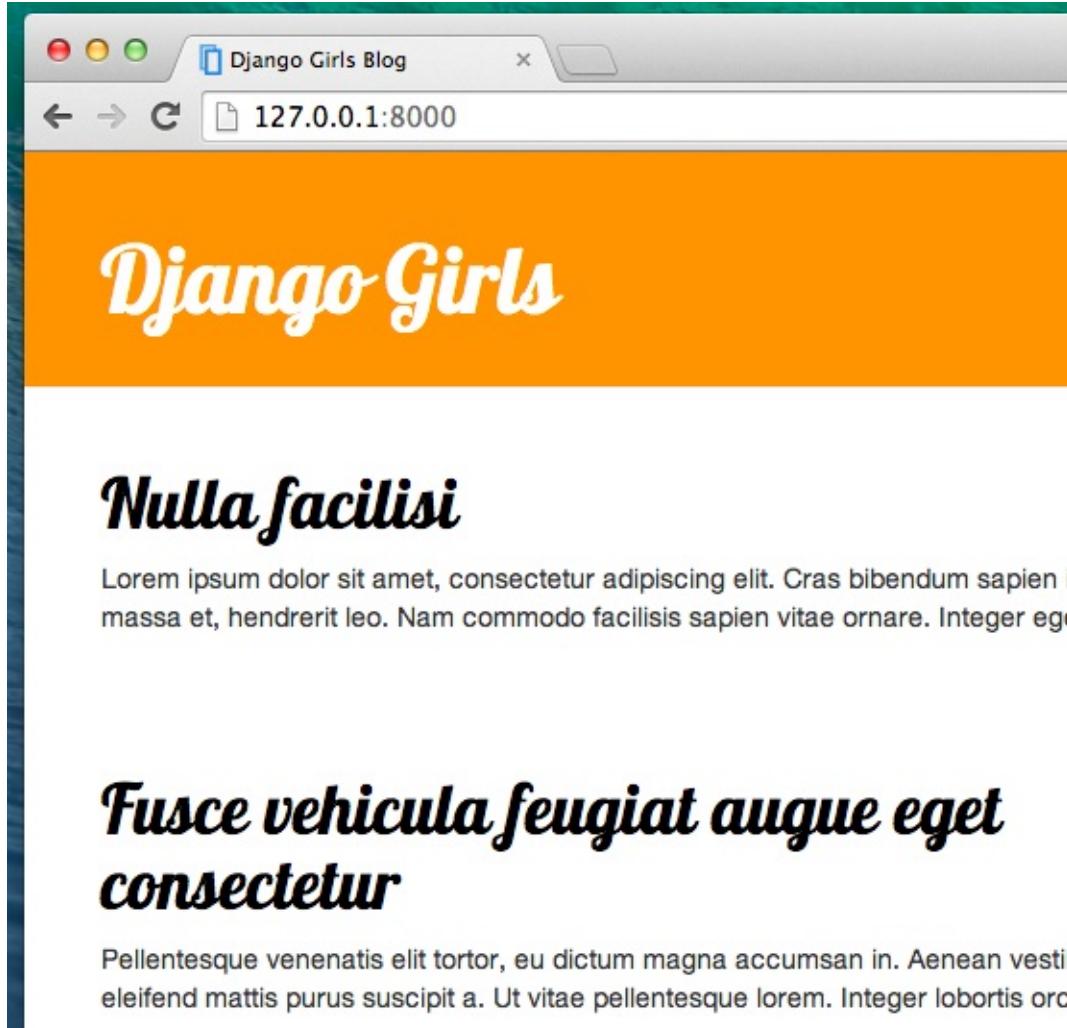
```
from django.shortcuts import render, get_object_or_404
```

Y en el final del archivo añadimos nuestra `view`:

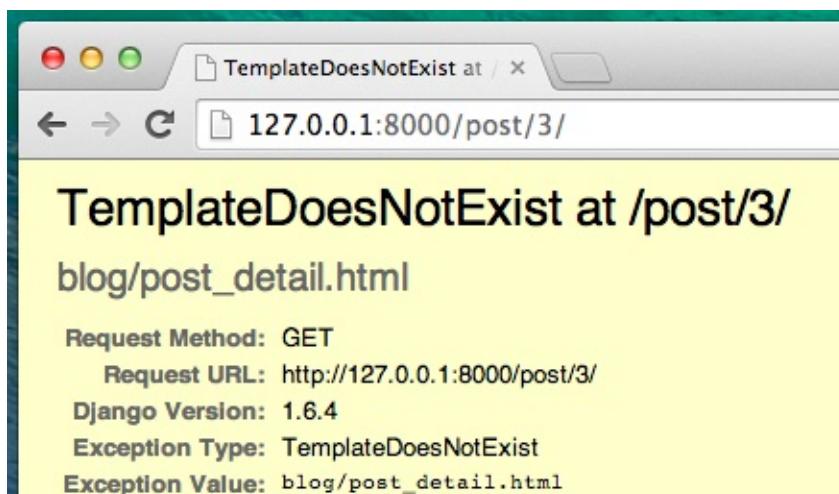
```
blog/views.py
```

```
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 agregar una plantilla!

Crear una plantilla para post detail

Crearemos un archivo en `blog/templates/blog` llamado `post_detail.html`.

Se verá así:

`blog/templates/blog/post_detail.html`

```
{% 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|linebreaksbr }}</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 querremos 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.

Terminal

```
$ git status
$ git add --all .
$ git status
$ git commit -m "Added view and template for detailed blog post as well as CSS for the site."
$ git push
```

Luego, en una [consola Bash de PythonAnywhere](#):

Terminal

```
$ 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 sitio web será crear una forma agradable de agregar y editar posts en el blog. El `admin` de Django 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 los formularios de Django es que podemos definirlos desde cero o crear un `ModelForm`, el cual 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, los formularios tienen su propio archivo: `forms.py`.

Necesitamos crear un archivo con este nombre en el directorio `blog`.

```
blog
└── forms.py
```

OK, vamos a abrirlo y a escribir el siguiente código:

`blog/forms.py`

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

    class Meta:
        model = Post
        fields = ('title', 'text',)
```

Primero necesitamos importar los formularios de Django (`from django import forms`) y, obviamente, nuestro modelo `Post` (`from .models import Post`).

`PostForm`, como probablemente sospechas, es el nombre de nuestro formulario. Necesitamos decirle a Django que este formulario es un `ModelForm` (así Django hará algo de magia por 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) deberían estar en nuestro formulario. En este escenario sólo queremos `title` y `text` para ser mostrados - `author` será la persona que se ha autenticado (¡tú!) y `created_date` se definirá automáticamente cuando creamos un post (es decir, en el código), ¿cierto?

¡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 agregar un enlace en un `div` llamado `page-header`:

`blog/templates/blog/base.html`

```
<a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
```

Ten en cuenta que queremos llamar a nuestra nueva vista `post_new`. La clase `"glyphicon glyphicon-plus"` es provista por el tema de bootstrap que estamos usando, y mostrará un signo más por nosotros.

Después de agregar la línea, tu archivo html debería tener este aspecto:

`blog/templates/blog/base.html`

```
{% load staticfiles %}
<html>
    <head>
        <title>Django Girls blog</title>
        <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">
                <link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'>
                    <link rel="stylesheet" href="{% static 'css/blog.css' %}">
    </head>
    <body>
        <div class="page-header">
            <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
            <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:

`blog/urls.py`

```
url(r'^post/new/$', views.post_new, name='post_new'),
```

Y el código final tendrá este aspecto:

`blog/urls.py`

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

urlpatterns = [
    url(r'^$', views.post_list, name='post_list'),
    url(r'^post/(?P<pk>\d+)/$', views.post_detail, name='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 agregarla 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 :
```

`blog/views.py`

```
from .forms import PostForm
```

And then our view:

`blog/views.py`

```
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 `{{ form.as_p }}`.
- 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 por último justo después de abrir la etiqueta `<form ...>` tenemos que 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 intentes guardar el formulario.



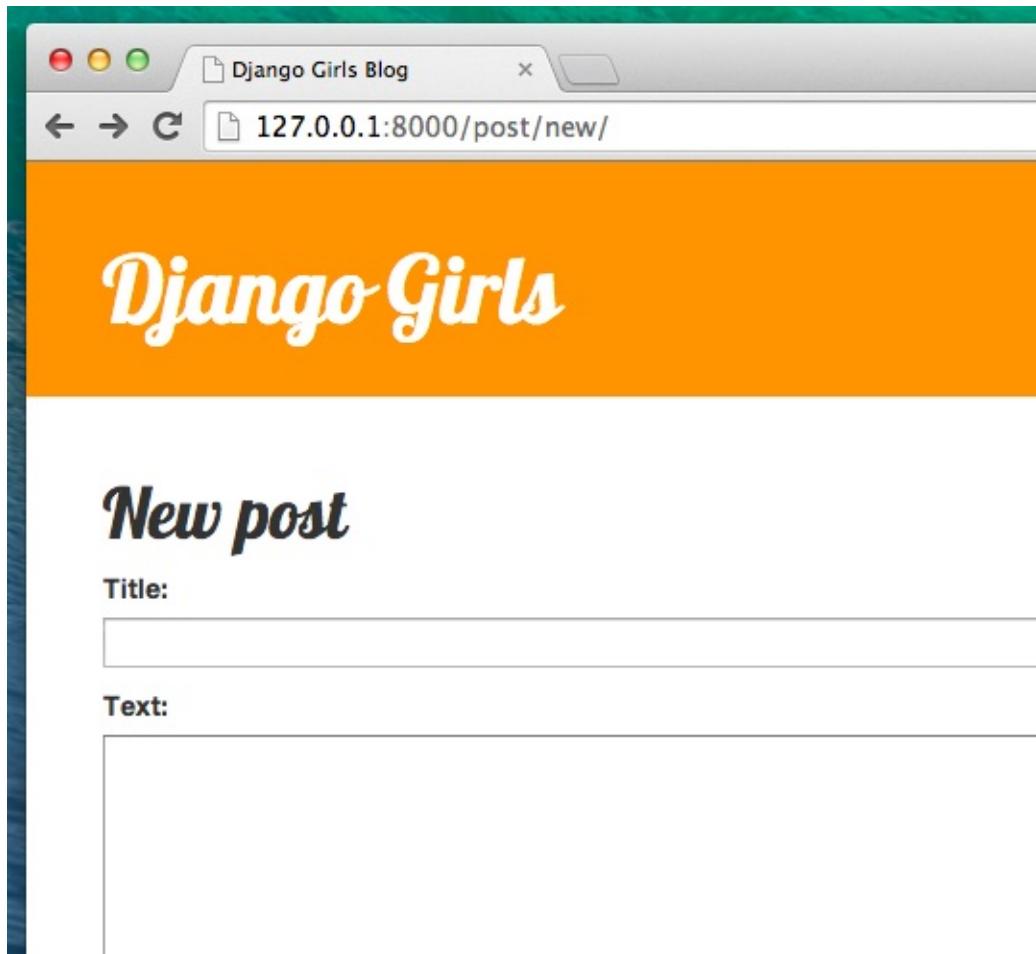
Bueno, vamos a ver cómo quedará el HTML en `post_edit.html`:

`blog/templates/blog/post_edit.html`

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

{% block content %}
    <h1>New post</h1>
    <form method="POST" class="post-form">{% csrf_token %}
        {{ 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:

`blog/views.py`

```
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 están 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 agregar una condición (utilizaremos `if` para eso).

blog/views.py

```
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:

blog/views.py

```
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!

blog/views.py

```
if form.is_valid():  
    post = form.save(commit=False)  
    post.author = request.user  
    post.published_date = timezone.now()  
    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 agregar 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 el 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:

blog/views.py

```
from django.shortcuts import redirect
```

Agrégalo al principio del archivo. Y ahora podemos decir: vé a la página `post_detail` del post recién creado.

blog/views.py

```
return redirect('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?

blog/views.py

```
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.published_date = timezone.now()
            post.save()
            return redirect('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`.

Puede que hayas notado que estamos indicando la fecha de publicación antes de guardar la entrada. Más adelante introduciremos un *botón de publicar* en el libro **Django Girls Tutorial: Extensions**.

¡Eso 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.

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!

The screenshot shows a web browser window titled "Django Girls Blog" at the URL "127.0.0.1:8000/post/new/". The page has a yellow header with the "Django Girls" logo. Below the header, the text "New post" is centered. There are two input fields: one for "Title:" and one for "Text:". Both fields have a red border and the error message "• This field is required." below them. The browser's status bar shows the URL "localhost:8000/post/new/".

Django se encarga de validar que todos los campos en el formulario estén correctos. ¿No es genial?

The screenshot shows a browser window with a yellow error box. The error message is "ValueError at /post/new/". Below it, the text "Cannot assign <SimpleLazyObject: <django.contrib.auth.models.AnonymousUser object at 0x3264b50>>: "Post.author" must be a "User" instance." is shown. At the bottom of the error box, there is some smaller text about the request method and URL.

```

Request Method: POST
Request URL: http://localhost:8000/post/new/
Django Version: 1.6.5
Exception Type: ValueError
Exception Value: Cannot assign <SimpleLazyObject: <django.contrib.auth.models.AnonymousUser object at 0x3264b50>>: "Post.author" must be a "User" instance.
Exception Location: /home/miguel/Documentos/django/django/django/contrib/auth/models.py in save, line 230

```

Editar el 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:

`blog/templates/blog/post_detail.html`

```
<a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>
```

para que la plantilla quede:

`blog/templates/blog/post_detail.html`

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

{% block content %}
<div class="post">
    {% if post.published_date %}
        <div class="date">
            {{ post.published_date }}
        </div>
    {% endif %}
    <a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>
    <h1>{{ post.title }}</h1>
    <p>{{ post.text|linebreaksbr }}</p>
</div>
{% endblock %}
```

En el archivo `blog/urls.py` añadimos esta línea:

`blog/urls.py`

```
url(r'^post/(?P<pk>\d+)/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`.

Abrimos el archivo `blog/views.py` y añadimos al final esta línea:

`blog/views.py`

```
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.published_date = timezone.now()
        post.save()
        return redirect('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:

`blog/views.py`

```
form = PostForm(request.POST, instance=post)
```

como al abrir un formulario con este post para editarlo:

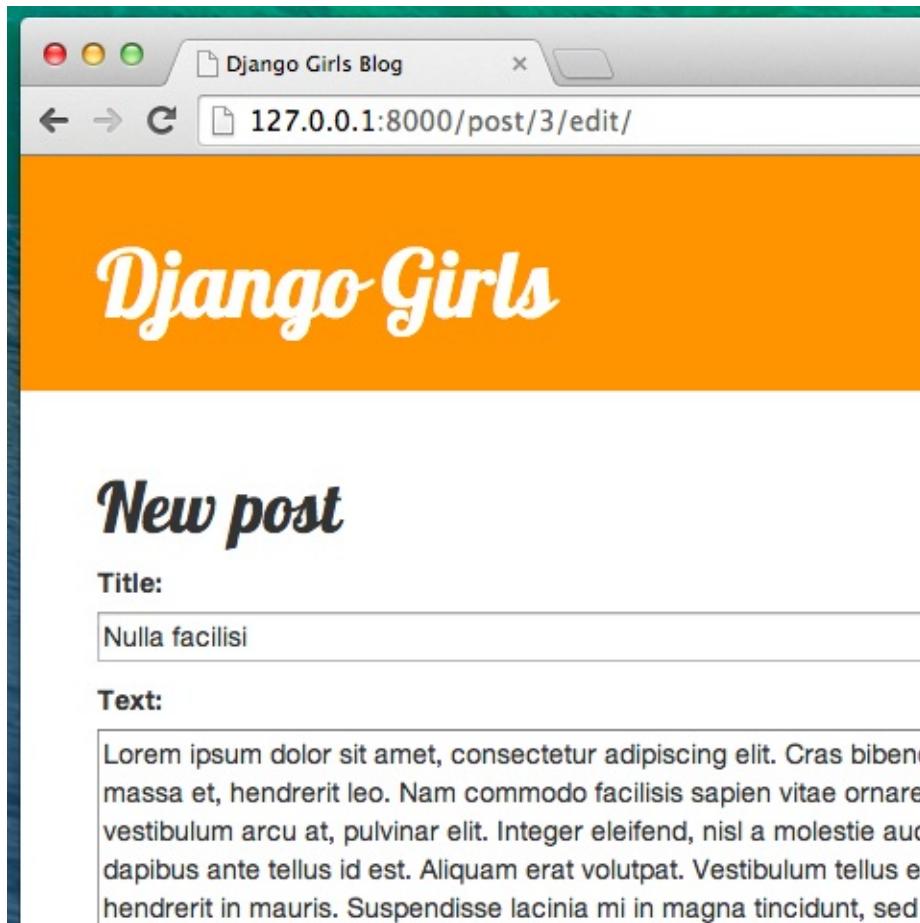
`blog/views.py`

```
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/1.10/topics/forms/>

Seguridad

¡Poder crear entradas nuevas sólo con pulsar un botón es genial! Pero, ahora mismo, cualquiera que visite tu página podría publicar una nueva entrada y seguro que eso no es lo que quieres. Vamos a hacer que el botón sea visible para ti pero no para nadie más.

En `blog/templates/blog/base.html`, buscar el `div page-header` y la etiqueta de enlace que has puesto ahí antes. Debería ser algo como:

`blog/templates/blog/base.html`

```
<a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
```

Vamos a agregar otra etiqueta `{% if %}` que hará que el enlace sólo parezca para los usuarios que hayan iniciado sesión en el admin. Ahora mismo, ¡eres sólo tú! Cambia la etiqueta `<a>` para que se parezca a esto:

blog/templates/blog/base.html

```
{% if user.is_authenticated %}
    <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus">
    </span></a>
{% endif %}
```

Este `{% if %}` hará que el enlace sólo se envíe al navegador si el usuario que solicita la página ha iniciado sesión. Esto no protege completamente la creación de nuevas entradas, pero es un buen primer paso. Veremos más sobre seguridad en el libro de extensiones.

¿Recuerdas el ícono de editar que agregamos en nuestra página de detalle? Queremos agregar ese mismo cambio aquí, así otras personas no son capaces de editar entradas existentes.

Abre `blog/templates/blog/post_detail.html` y busca esta línea:

blog/templates/blog/post_detail.html

```
<a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>
```

Cámbiala por esta:

blog/templates/blog/post_detail.html

```
{% if user.is_authenticated %}
    <a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>
{% endif %}
```

Ya que probablemente tengas iniciada la sesión, si refrescas la página no verás nada distinto. Sin embargo, abre la página en un navegador nuevo o en una ventana de incógnito y verás que el enlace no se muestra!

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

* Luego, en una [consola Bash de PythonAnywhere](https://www.pythonanywhere.com/consoles/)

```
:
```

```
{% filename %}Terminal{% endfilename %}
```

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

- Finalmente, ve a la pestaña [Web](#) y haz click en [Reload](#).

¡Y eso debería ser todo! Felicidades :)

¿Qué sigue?

¡Date muchas felicitaciones! ¡Eres increíble! ¡Estamos orgullosas! <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 otro libro llamado [Tutorial de Django Girls: Extensiones](#).

Más adelante, puedes intentar los recursos listados a continuación. ¡Son todos muy recomendables!

- [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](#)

Si estás haciendo el tutorial en casa

Si estás haciendo el tutorial en casa, y no en uno de los [eventos de Django Girls](#), puede saltarse este capítulo por completo e ir directamente al capítulo [¿cómo funciona Internet?](#).

Esto es porque cubrimos estas cosas en todo el tutorial de todos modos, y esto es sólo una página adicional que recoge todas las instrucciones de instalación en un solo lugar. El evento de chicas Django incluye una "noche de la instalación" cuando instalamos todo, así que no tenemos que molestar con ella durante el taller, así que esto es útil para nosotros.

Si lo encuentra útil, puede seguir este capítulo también. Pero si quieras empezar a aprender antes de instalar un montón de cosas en tu computadora, sáltate este capítulo y te explicaremos la parte de instalación más adelante.

¡Buena suerte!

Instalación

En el taller construiremos un blog, y hay algunas tareas de configuración en el tutorial que sería bueno hacer de antemano para que estés listo para comenzar a codificar en el día.

Configuración de Chromebook (si estás usando una)

Puedes [ir directamente a esta sección](#) si no utilizas una Chromebook. Si lo haces, el proceso de instalación será algo diferente. Puedes ignorar el resto de las instrucciones de instalación.

Cloud 9

Cloud 9 es una herramienta que te da un editor de código y acceso a una computadora ejecutándose en Internet donde puedes instalar, escribir y ejecutar software. Durante el tutorial Cloud 9 será como tu *máquina local*. Vas a estar ejecutando comandos en una interfaz de terminal al igual que tus compañeros de clase en OS X, Ubuntu o Windows pero tu terminal estará conectada a una computadora ejecutándose en algún otro lugar que Cloud 9 configure para vos.

1. Instala Cloud 9 desde la [tienda web de Chrome](#)
2. Dirígete a c9.io
3. Regístrate para obtener una cuenta
4. Clickea en *Create a New Workspace*
5. Nombralo *djangogirls*
6. Selecciona *Blank* (el segundo desde la derecha en la fila inferior con logo naranja)

Ahora deberías ver una interfaz con una barra lateral, una ventana principal grande con algo de texto y, una ventana pequeña en la parte inferior que luce como esta:

Ahora deberías ver una interfaz con una barra lateral, una ventana principal grande con algo de texto y, una ventana pequeña en la parte inferior que luce como esta:

Cloud 9

```
yourusername:~/workspace $
```

El área inferior es tu *terminal*, donde obtendrás la computadora que Cloud 9 ha preparado para tus instrucciones. Puedes redimensionar esa ventana para hacerla un poco mas grande.

Entorno Virtual

Un entorno virtual (también llamado virtualenv) es como una caja personal donde guardamos el código útil de un proyecto en el que estamos trabajando. Lo usamos para mantener separados fragmentos de código de distintos proyectos, de modo que no se mezclen entre si.

En la terminal que se encuentra en la parte inferior de la interfaz de Cloud 9 ejecuta lo siguiente:

Cloud 9

```
sudo apt install python3.5-venv
```

Si esto no funciona, pide ayuda a tu guía.

Luego, ejecuta:

Cloud 9

```
mkdir djangogirls
cd djangogirls
python3.5 -mvenv myvenv
source myvenv/bin/activate
pip install django~=1.10.0
```

(nota que en la última línea usamos un tilde seguido por un signo igual: ~=).

Github

Crea una cuenta en [Github](#).

PythonAnywhere

El tutorial de Django Girls incluye una sección denominada Despliega, que consiste en el proceso de tomar el código de tu nueva aplicación web y llevártalo a una computadora accesible al público (llamada servidor) para que otras personas puedan ver tu trabajo.

Este proceso es algo particular al hacer el tutorial en una Chromebook debido a que estamos utilizando una computadora que está en Internet (en contraposición, digamos, a una portátil). Sin embargo aún así es útil, ya que podemos pensar a Cloud 9 como un lugar para nuestro trabajo "en progreso" y a Python Anywhere como el lugar para mostrar el trabajo a medida que se va completando.

Por tanto, registra una nueva cuenta en Python Anywhere www.pythonanywhere.com.

Instalar Python

Para lectores en casa: esta parte está cubierta en el video [Installing Python & Code Editor](#).

Esta sección está basada en un tutorial por Geek Girls Carrots
(<https://github.com/ggcarrots/django-carrots>)

Django está escrito en Python. Necesitamos Python para hacer cualquier cosa en Django. ¡Vamos a empezar con la instalación! Queremos que instales Python 3.5, así que si tienes alguna versión anterior, deberás actualizarla.

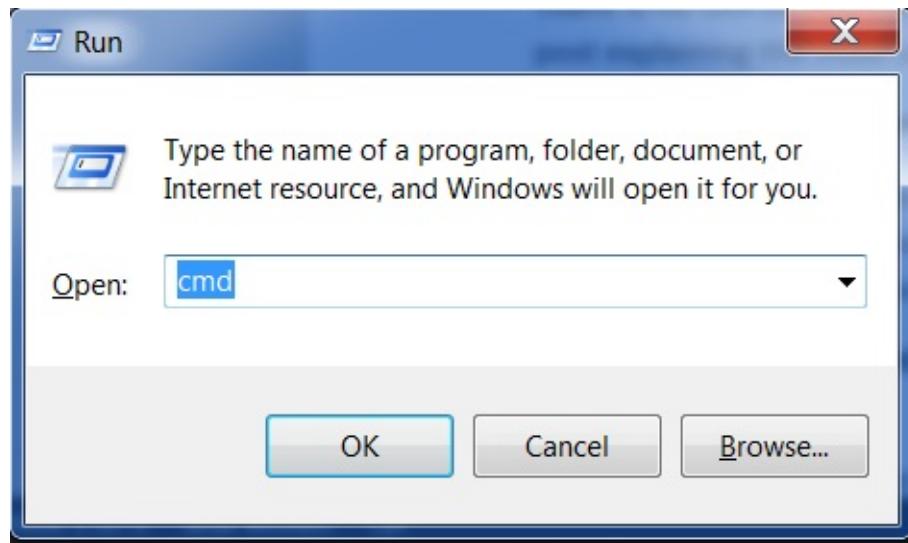
Windows

Primero comprueba si tu computadora está corriendo una versión de 32-bit o una de 64-bit de Windows en <https://support.microsoft.com/en-au/kb/827218>. Puedes descargar Python para Windows desde el sitio web <https://www.python.org/downloads/windows/>. Haz click en link "Latest Python 3 Release - Python x.x.x". Si tu computadora está corriendo una versión de **64-bit** de Windows, descarga el **Windows x86-64 executable installer**. De lo contrario, descarga el **Windows x86 executable installer**. Luego de descargar el instalador, debes ejecutarlo (doble click en él) y seguir las instrucciones.

Algo para tener en cuenta: Durante la instalación notarás una ventana llamada "Setup". Asegúrate de tildar la opción "Add Python 3.5 to PATH" y luego hacer click en "Install Now", como se muestra aquí:



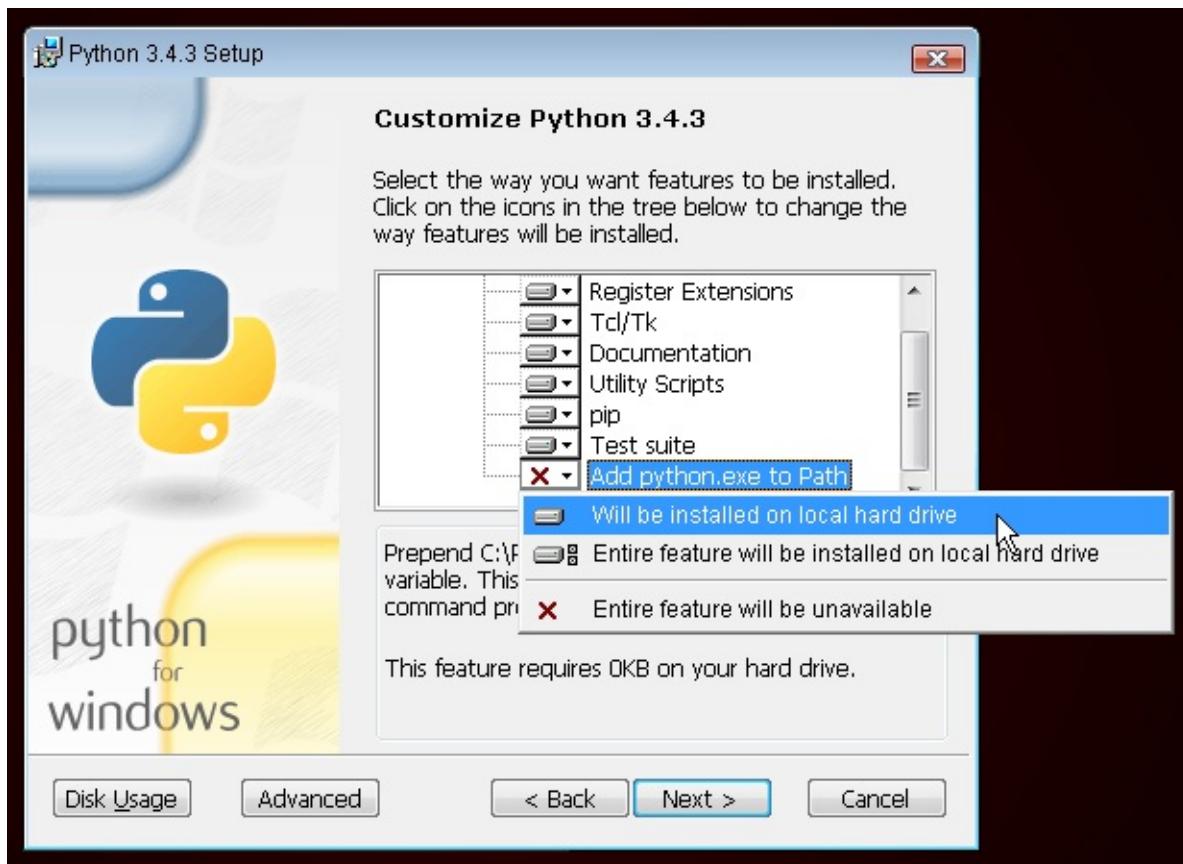
En los siguientes pasos, vamos a usar la Línea de Comandos de Windows (ya te vamos a contar sobre ella). Por ahora, si necesitas escribir algún comando, ve a Menu → Todos los programas → Accesorios → Símbolo del sistema. También puedes mantener presionada la tecla de Windows y presionar la tecla "R" hasta que aparezca la ventanita "Run". Para abrir la Línea de Comandos de Windows, escribe "cmd" y presiona `intro` la ventana "Run" (en las nuevas versiones de Windows, quizás tengas que buscar "Línea de Comandos" ya que a veces está escondido)



Nota: si estás usando una versión más vieja de Windows (7, Vista o cualquiera más vieja) y el instalador de Python 3.5.x falla con un error, puedes probar lo siguiente:

1. instala todos las Actualizaciones de Windows y prueba volver a instalar Python 3.5; o
2. instala una [versión más vieja de Python](#), por ejemplo la [3.4.4](#).

Si instalas una versión más vieja de Python, es posible que la pantalla del instalador se vea un poquito diferente que la mostrada anteriormente. Asegúrate de ir hacia el final y ver "Add python.exe to Path", luego hacer click en el botón y seleccionar "Will be installed on local hard drive":



OS X

Nota Antes de que instales Python en OS X, debes asegurarte de que las configuraciones de tu Mac te permitan instalar paquetes que no son del App Store. Ve a Preferencias del Sistema (está en la carpeta Aplicaciones), haz click en "Seguridad y Privacidad", y luego en la pestaña "General". Si yo "Permitir aplicaciones descargadas desde:" está configurado como "Mac App Store", cámbialo a "Mac App Store and identified developers"

Necesitas ir a la página web <https://www.python.org/downloads/release/python-351/> y descargar el instalador de Python:

- Descarga el archivo *Mac OS X 64-bit/32-bit installer*,
- Haz doble click en *python-3.5.1-macosx10.6.pkg* para ejecutar el instalador.

Linux

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

Terminal

```
$ python3 --version  
Python 3.5.1
```

Si tienes una 'micro versión' diferente de Python instalado, por ejemplo 3.5.0, no necesitas actualizar. Si no tienes instalado Python o si deseas una versión diferente, puedes instalarla de la siguiente manera:

Debian o Ubuntu

Escribe este comando en tu consola:

Terminal

```
$ sudo apt-get install python3.5
```

Fedora (hasta 21)

Usa este comando en tu consola:

Terminal

```
$ sudo yum install python3
```

Fedora (22+)

Usa este comando en tu consola:

Terminal

```
$ sudo dnf install python3
```

openSUSE

Use this command in your console:

Terminal

```
$ sudo zypper install python3
```

Verifica que la instalación fue correcta abriendo la aplicación de *Terminal* y ejecutando el comando

```
python3 :
```

Terminal

```
$ python3 --version  
Python 3.5.1
```

NOTA: Si estás en Windows y obtienes un mensaje de error diciendo `python3` no se reconoce como comando, intenta con `python` (sin el `3`) y comprueba si todavía es una versión de Python 3.5.

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

Configurar `virtualenv` e instalar Django

Parte de esta sección está basada en tutoriales de Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Parte de esta sección se basa en el [django-marcador tutorial](#) bajo licencia 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 tu computadora. Es posible saltarse este paso, pero es altamente recomendable. ¡Empezar con la mejor configuración posible te ahorrará muchos problemas en el futuro!

Así que, vamos a crear un **entorno virtual** (también llamado un *virtualenv*). Virtualenv aísla tu configuración de Python/Django por cada proyecto. Esto quiere decir que cualquier cambio que hagas en un sitio web no afectará a ningún otro que estés desarrollando. Genial, ¿no?

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

NOTA: En Windows, asegúrate que este directorio no contenga acentos ni caracteres especiales; si tu nombre contiene acentos, utiliza un directorio diferente, por ejemplo `C:\djangogirls`.

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

Terminal

```
$ mkdir djangogirls
$ cd djangogirls
```

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

Terminal

```
$ 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:\Python35\python -m venv myvenv`. Se verá así:

Terminal

```
C:\Users\Name\djangogirls> C:\Python35\python -m venv myvenv
```

en donde `C:\Python35\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 usar 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:

Terminal

```
$ python3 -m venv myvenv
```

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

NOTA: En algunas versiones de Debian/Ubuntu quizás recibas el siguiente error:

Terminal

```
The virtual environment was not created successfully because ensurepip is not available. On Debian/Ubuntu systems, you need to install the python3-venv package using the following command.
```

```
apt-get install python3-venv  
You may need to use sudo with that command. After installing the python3-venv package, recreate your virtual environment.
```

En este caso, sigue las instrucciones mencionadas arriba e instala el paquete `python3-venv` así:

Terminal

```
$ sudo apt-get install python3-venv
```

NOTA: En algunas versiones de Debian/Ubuntu inicializar el entorno virtual puede producir el siguiente error:

Terminal

```
Error: Command '['/home/eddie/Slask/tmp/venv/bin/python3', '-Im', 'ensurepip', '--upgrade', '--default-pip']}' returned non-zero exit status 1
```

Para solucionar esto, usa el comando `virtualenv` en cambio.

Terminal

```
$ sudo apt-get install python-virtualenv  
$ virtualenv --python=python3.5 myvenv
```

NOTA: Si obtienes un error como este

Terminal

```
E: Unable to locate package python3-venv
```

intenta ejecutar el siguiente comando en cambio:

Terminal

```
sudo apt install python3.5-venv
```

Trabajar con `virtualenv`

El comando anterior creará un directorio llamado `myvenv` (o cualquier nombre que hayas elegido) que contiene nuestro entorno virtual (básicamente un montón de archivos y carpetas).

Windows

Inicia el entorno virtual ejecutando:

Terminal

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

NOTA: en Windows 10 quizás obtengas un error en la consola Window PowerShell diciendo `execution of scripts is disabled on this system`. En ese caso, abre otra ventana Windows PowerShell con la opción "Run as Administrator". Luego intenta escribir lo siguiente antes de iniciar tu entorno virtual:

Terminal

```
C:\WINDOWS\system32> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned  
Execution Policy Change  
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the about_Execution_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy? [Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): A
```

Linux y OS X

Inicia el entorno virtual ejecutando:

Terminal

```
$ source myvenv/bin/activate
```

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

NOTA: a veces `source` podría no estar disponible. En ese caso trata hacerlo de esta forma:

Terminal

```
$ . myvenv/bin/activate
```

Sabrás que tienes `virtualenv` iniciado cuando veas que la línea de comando tiene este prefijo `(myvenv)`.

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

OK, tenemos todas las dependencias importantes en su lugar. ¡Finalmente podemos instalar Django!

Instalar Django

Ahora que tienes tu `virtualenv` iniciado, puedes instalar Django.

Antes de hacer eso, deberíamos asegurarnos de que tenemos la última versión de `pip`, el programa que usamos para instalar Django:

Terminal

```
(myvenv) ~$ pip install --upgrade pip
```

Luego ejecuta `pip install django~=1.10.0` (nota que usamos la tilde seguido de un signo igual: `~=`) para instalar Django.

Terminal

```
(myvenv) ~$ pip install django~=1.10.0
Collecting django~=1.10.0
  Downloading Django-1.10.4-py2.py3-none-any.whl (6.8MB)
Installing collected packages: django
Successfully installed django-1.10.4
```

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\djangoirls`). Si lo tiene, por favor considera moverla a otro lugar sin espacios, acentos o caracteres especiales (sugerencia: `C:\djangogirls`). Crea un nuevo entorno virtual en ese directorio, luego elimina el viejo y vuelve a probar el comando anterior (mover el directorio del `virtualenv` no funcionará debido a que `virtualenv` usa rutas absolutas)

Windows 8 y Windows 10

Tu línea de comandos quizás se congele luego de que intentas instalar Django. Si esto sucede, intenta este comando en cambio:

Terminal

```
C:\Users\Name\djangogirls> python -m pip install django~=1.10.0
```

Linux

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

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

Instalar un editor de código

Existe una gran cantidad de editores diferentes y la elección queda reducida en gran medida a las preferencias personales. La mayoría de programadoras de Python usan IDEs (Entornos de Desarrollo Integrados) complejos pero muy poderosos, como PyCharm. Sin embargo, como principiante, probablemente no es muy adecuado; nuestras recomendaciones son igualmente poderosas pero mucho más simples.

Nuestras sugerencias están listadas abajo, pero siéntete libre de preguntarle a tu tutora o 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 3

Sublime Text es un editor muy popular con un periodo de prueba gratis. Es fácil de instalar y de usar, 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 no producen texto plano. Lo que generan es texto enriquecido (con fuentes y formato), usando formatos propios como [RTF \(Rich Text Format\)](#) o en español, "Formato de Texto Enriquecido".

La segunda razón es que los editores de código son herramientas especiales para editar código, porque proveen características útiles como resaltar el 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 :)

Instalar Git

Git es un "sistema de control de versiones" que utilizan muchas programadoras y programadores. Este software puede rastrear los cambios realizados en archivos a lo largo del tiempo de forma que puedas recuperar una versión específica más tarde. Es un poco parecido a la opción de "control de cambios" de Microsoft Word, pero mucho más potente.

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", elige "Run Git and associated Unix tools from the Windows command-line" (la última opción). Aparte de eso, los valores por defecto están bien. "Checkout Windows-style, commit Unix-style line endings" también está bien.

OS X

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

Nota Si estás en OS X 10.6, 10.7 o 10.8, necesitarás instalar la versión de Git desde aquí: [instalador Git para OS X Snow Leopard](#)

Debian o Ubuntu

Terminal

```
$ sudo apt-get install git
```

Fedora (hasta 21)

Terminal

```
$ sudo yum install git
```

Fedora 22+

Terminal

```
$ sudo dnf install git
```

openSUSE

Terminal

```
$ sudo zypper install git
```

Crear una cuenta de GitHub

Visita [GitHub.com](#) y regístrate para una nueva cuenta de usuario, gratuita.

Crear una cuenta de 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 elige tu propio apodo o bien un nombre que describa sobre qué trata tu blog.

Comienza a leer

Felicitaciones, ya tienes todo configurado y listo para seguir! Si aún tienes tiempo antes del taller, sería útil comenzar a leer algunos de los capítulos iniciales:

- [¿Cómo funciona internet?](#)
- [Introducción a la línea de comandos](#)
- [Introducción a Python](#)
- [¿Qué es Django?](#)

¡Disfruta el taller!

Cuando comiences el taller podrás ir directamente a la [¡Tu primer project Django!](#) porque ya cubriste el contenido de los capítulos anteriores.