Tutorial

django girls

Django Girls 教學

介紹

你曾經感覺到這個世界越來越高科技而你似乎被遺棄了嗎？你曾經好奇如何創建一個網站但卻一直沒有足夠的動機開始動手嗎？你有沒有曾經認為，在你想要做一些屬於自己的作品時，這個軟體的世界對你來說好像是太過於複雜了呢？

我們為你帶來佳音了！學寫程式沒有你想得困難，我們現在就可以秀給你看，寫程式能多有趣！

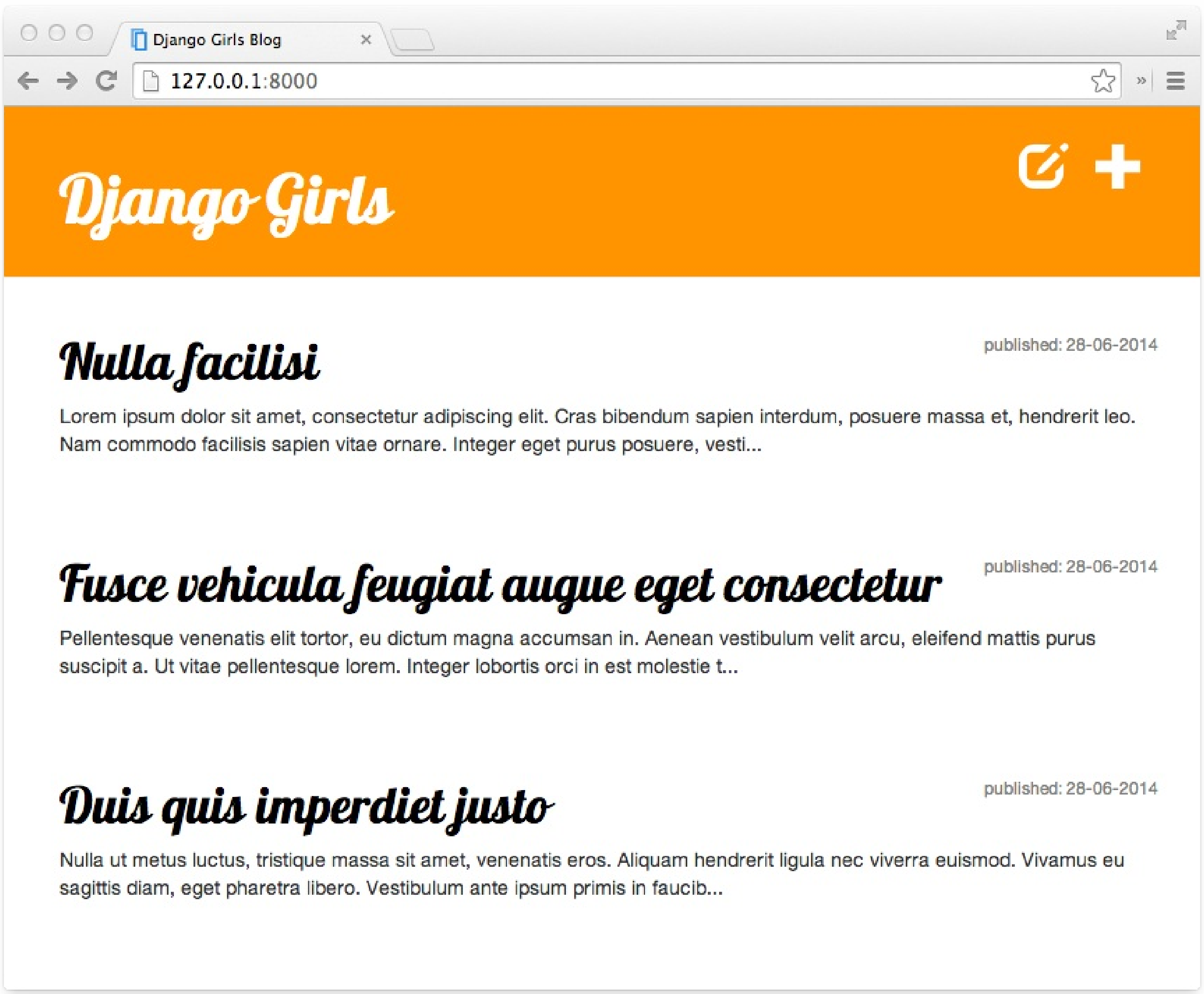
這個教程不會像變魔術一樣讓你一夜之間成為一個程序員。如果你想要成為高手，你需要經年累月的學習與實戰。但我們想要展現給你看看寫程式與創建一個網站並沒有想像中困難。我們會盡其所能的解釋每一個你需要知道的基礎知識，讓你不再對科技感到卻步。

我們期待，我們能夠讓你愛上這些技術，如同我們一樣！

在這本教程中你能夠學到什麼？

一旦你完成本教程，你將會有一個自己的作品，一個可運作的網站程式：你自己的部落格。我們將為你示範如何將它放上線，讓大家都可以看見你的成果！

這個作品大概會看起來像這樣：



Nulla facilisi

published: 28-06-2014

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras bibendum sapien interdum, posuere massa et, hendrerit leo. Nam commodo facilisis sapien vitae ornare. Integer eget purus posuere, vesti...

Fusce vehicula feugiat augue eget consectetur

published: 28-06-2014

Pellentesque venenatis elit tortor, eu dictum magna accumsan in. Aenean vestibulum velit arcu, eleifend mattis purus suscipit a. Ut vitae pellentesque lorem. Integer lobortis orci in est molestie t...

Duis quis imperdiet justo

published: 28-06-2014

Nulla ut metus luctus, tristique massa sit amet, venenatis eros. Aliquam hendrerit ligula nec viverra euismod. Vivamus eu sagittis diam, eget pharetra libero. Vestibulum ante ipsum primis in faucib...

好了，讓我們細說從頭吧...

About and contributing

這是 Django Girls 官方教材中譯版。如果你發現任何錯誤並且想要更新它歡迎 Fork 這個專案，本書全冊可於 GitBook Django Girls' Tutorial 中文版教材 上閱讀。

Django Girls Taipei 有編寫一份Django Girls 學習指南 於 Django Girls Taipei 使用，需要的人也可以參考 :)

網際網路(Internet)是如何運作的

> 本章節是受 Jessica McKellar (http://web.mit.edu/jesstess/www/) 的一場小講題 "How the Internet works" 所啓發

我們敢打賭你應該每天都有使用網際網路。但是你真的知道，當你輸入一個網址時，像是 http://djangogirls.org 到你的瀏覽器並按下「Enter」的時候發生了哪些事情嗎？

你需要明白的第一件事就是，其實一個網站就是一堆存在硬碟的檔案啦。就像是你電腦裡的眾多影片、音樂和照片。即便如此，網站的組成中還是有一部分是獨一無二的：他們都含有一種叫做「HTML」的程式碼。

如果你還不熟悉編程，你一開始可能會有點難理解什麼叫做 HTML，但是你的網站瀏覽器（像是 Chrome, Safari, Firefox 等等）超愛它。網站瀏覽器就是設計出來讀懂 HTML 的，遵循它的架構及切切實實地展示你希望瀏覽器展示的所有檔案。

我們需要將 HTML 檔案們，及其所包含的所有檔案存在硬碟的某處。為了放上網際網路，我們使用了特別且強悍的電腦 -- 稱為 伺服器 (server)。這些伺服器不需要螢幕、滑鼠甚至是鍵盤，因為這些電腦的主要目的就是儲存這些資料並且提供所有人使用(serve it)。這是為什麼它們被稱為 伺服器*(server)* -- 因為他們 提供*(serve)* 你的資料。

好了，但你還是想要知道網際網路到底是啥，對嗎？

我們為你畫了張示意圖，它看起來像是這樣：

看起來真是一團亂啊 XD

事實上它就是聯繫機器（上面提到的伺服器）間的網絡。成千上萬台的機器啊！無法想像幾公里長的電纜存在在這個世界上！你可以訪問一個叫做 Submarine 纜線地圖 (http://submarinecablemap.com/) 的網站，去看看這個連結有多複雜。這是網站中的一個截圖：

這真是無法想像，對吧？但是很明顯地，要在任意兩台機器間都有電纜連結這件事本身是不可能的。所以呢，為了可以連結到某台機器（例如說儲存了本站檔案的 http://djangogirls.org）我們需要傳遞我們的要求（Request），經由很多很多台不同的機器。

這件事看起來就像是這樣：

想像一下當你輸入 http://djangogirls.org ，你等同於像是寄了一封信裡面寫著：「親愛的 Django Girls, 我想要看看 djangogirls.org 這個網站，請把裡面的資訊寄給我，謝謝！」

你的信會先送到最近的郵局。然後再送到下一個離你的收件地址更近一點的，然後再下一個，直到這封信被傳遞到目的地為止。唯一一件比較特別的事是如果你送了頻繁的信 (封包) 到同一個地址，每一封信都會經由完全不同的郵局路徑 (路由)，這是依據他們如何被分散到各個郵局。

沒錯以上就是個小範例。你送出訊息並且期望會收到某個回應，當然，除了紙筆你也可以使用資料位元，這是同樣的概念！

除了有國家、郵遞區號、城市與街道名稱所組成的地址，我們也使用 IP 作為地址。你的電腦會先要求網域系統 (DNS - Domain Name System) 將 djangogirls.org 這個網域轉換為 IP。這個運作方式就有點像是古早以前的黃頁電話簿，你可以先找某個人的名字然後用他的電話號碼和地址與他聯繫。

當你要寄一封信，就需要一些功能讓它可以被正確的遞送：一個地址，郵戳等等。你也需要使用收件者可以理解的語言，對吧？這與你依順序送出的 封包 要求觀看一個網站其實是同一件事：你使用一個被稱為 HTTP（超文本轉換） 的協定（protocol）

所以呢，基本上，當你有了一個網站程式你也需要有一個 伺服器*(server)* (機器) 讓你的網站活起來。這個 伺服器 一直在等待來自各方的 要求*(request)* (這個詞意思是要求伺服器傳回你的網站資料)，然後伺服器會就回應你你要求的網站 *(response)* 。

從這個 Django tutorial 教材中你可以了解 Django 實際上做了什麼，當你需要回傳一個回應，你並不想總是對不同的人回傳同一個回應，你的信是更加個人化的、更加特別針對某個人只為你量身打造的，豈不是更好嗎？Django 便會幫助你去創造這些更私人，更有趣的回信 :)

說夠了，動手做吧！

介紹命令行(command-line)介面

嗯，實在是好興奮啊，對嗎？XD

待會兒你就要寫第一行程式碼了 :)

容我們為你介紹你的第一個新朋友：命令行！！

接下來的步驟將會為你示範如何使用這個駭客們都一定要使用的黑畫面。這或許第一次看起來有點可怕，但其實不過就是一個閃爍的小游標在等著你下命令給它啦～

什麼是命令行(command-line)

這個視窗，就是通常被稱為 命令行（**command line**） 或 命令行介面 的東西，這是一個以文字為主的應用程式，可以查看、處理並且控制你電腦裡的檔案們（更像是 Windows 系統中的檔案總管或是 Mac 裡的 Finder，但是少了圖形化界面）。還有其他稱呼像是 *cmd*, *CLT*, *prompt*, *console* 或是 *terminal(*終端機*)*。

打開命令行介面

開始試驗看看吧，首先我們需要打開命令行介面。

Windows

按下 開始 → 所有程式 → 附屬應用程式 → 命令提示字元

Mac OS X

應用程式 → 工具程式 → 終端機

Linux

他有可能在 應用程式 → 附屬應用程式 → 終端機 這個路徑下，但主要視你的版本而定。如果不是這裡所提到的開啟方式，就麻煩 Google 它一下 :)

提示字元(Prompt)

你應該看到一個白的（或黑的）視窗正在等候你的命令。

如果你是用 Mac 或 Linux, 你大概看到一個 `$` 符號，像這樣：

```
$
```

在 Windows 下，它是 `>` 符號，像這樣

```
>
```

每個命令都將會以這個提示字元與一個空白字元來準備運行，但是你不需要鍵入這兩個字元，你的電腦會很貼心的幫你打好 :)

> 小提示：在你的情況下可能會像是 `C:\Users\ola>` 或 `Olas-MacBook-Air:~ ola$` 。在這本教程我們會僅可能讓事情敘述的單純點。

你的第一個 **command** （命令）**! \o/**

我們來點簡單的開頭的，輸入以下指令：

```
$ whoami
```

或

```
> whoami
```

然後按下「Enter」。這是我們的結果

```
$ whoami
olasitarska
```

如同你所看到的，電腦就會秀出你的使用者名稱，很簡潔對吧？

> 試著鍵入每一個指令，不要複製貼上。用這個方法你可以記住更多！

基礎

每一個作業系統的命令行都有可觀的不同的指令，所以確定遵循你的自有設備，現在就容我們來試試看吧！

目前路徑

知道我們現在身在路徑的哪裡是很棒的事情，讓我們試試看，鍵入以下指令並按下 Enter：

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

如果你使用 Windows 作業系統：

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

你將可能在你的電腦上看到這些相似的結果。通常你打開命令行，都會以你的使用者根目錄為起點。

列出所有檔案和路徑

有哪些東西在這個資料夾下呢？試著找找吧，讓我們看看：

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

Windows:

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

改變現在的路徑

或許我們現在可以到「桌面」這個路徑去看看？

```
$ cd Desktop
```

Windows:

```
> cd Desktop
```

檢查看看是不是路徑已經變更了？

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

Windows:

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

就是這麼簡單！

> 進階技巧：如果你輸入 `cd D` 然後按一下鍵盤上的 `tab` 鍵，這個命令就會自動完成剩下未打完的部分。如果有超過一個開頭是 "D" 的資料夾，那就重複按 `tab` 鍵來選擇你真的要進入的目錄。

創建路徑

新建一個 Djangogirls 的資料夾在你的桌面上怎麼樣？你可以這麼做：

```
$ mkdir djangogirls
```

Windows:

```
> mkdir djangogirls
```

這些小指令可以建立一個叫做 `djangogirls` 的資料夾在你的桌面。你可以用 `ls` / `dir` 指令檢查看看是不是真的有在你的桌面上，試試看吧 :)

> 進階技巧：如果你不想重複的輸入一樣的指令，試著使用上下方向鍵來選擇你使用過的指令。

實戰！

給你的一個小挑戰： 在你新創建的 `djangogirls` 資料夾下創造一個叫做 `test` 的資料夾，使用 `cd` 及 `mkdir` 指令。

解答：

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

Windows:

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

恭喜! :)

清除

我們不想弄得一團亂，所以來移除上一個練習我們所創建的全部東西。

首先，我們需要回到桌面：

```
$ cd ..
```

Windows:

```
> cd ..
```

回到上層目錄使用 `cd` 與 `..` 將會改變你目前的資料夾令你回到上一層（父目錄）

檢查一下目前路徑（你在哪）

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

Windows:

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

是時候刪除 `djangogirls` 資料夾了:

```
$ rm -r djangogirls
```

Windows:

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

完成！確保這個資料夾真的被刪除了，我們可以輸入：

```
$ ls
```

Windows:

```
> dir
```

> 注意: 用 `del` , `rmdir` 或 `rm` 刪除檔案是無法回復的，代表 永遠刪除檔案! 所以請小心服用。

離開

是時候了！你現在可以安全地關閉命令行。我們用更 hacker 的方法來做這件事，試試看吧？ :)

```
$ exit
```

Windows:

```
> exit
```

酷吧？:)

總結

這裡是一些常用的指令：

指令 **(Windows)**	指令 **(Mac OS / Linux)**	描述	例子
exit	exit	關閉視窗	**exit**
cd	cd	修改資料夾	**cd test**
dir	ls	條列資料檔案路徑	**dir**
copy	cp	複製檔案	**copy c:\test\test.txt c:\windows\test.txt**
move	mv	移動檔案	**move c:\test\test.txt c:\windows\test.txt**
mkdir	mkdir	新建目錄	**mkdir testdirectory**
del	rm	刪除目錄/檔案	**del c:\test\test.txt**

這裡只是一些最基礎的命令行，你可以跑跑看。

如果你還好奇，ss64.com 包含了非常完整所有作業系統的的指令。

好了嗎**?**

讓我們迎接 Python 吧！

讓我們開始 **Python** 吧！

終於到了這一步！

但首先，讓我們告訴你什麼是 Python。Python 是一個非常流行的程式語言，可以用來創建網站、遊戲、科學研究用的軟體、圖表，還有更多說不完的東西。

Python 始於 1980 年代末，它主要的目的是希望更貼近「人類」（不僅是機器！）所使用的自然語言，這是為什麼它看起來比其他程式語言更加簡潔。這使 Python 更易學，但別擔心， Python 同時也非常強悍！

安裝 Python

> 本次節是源於怪咖女孩 Carrots (http://django.carrots.pl/)

Django 是用 Python 寫成的。我們需要使用 Python 在 Django 中做任何事。讓我們開始安裝吧！我們希望你安裝 Python 3.4，所以如果你的電腦裡有其他更早期的版本，你將需要升級它。

Windows

你可以從這個網站 https://www.python.org/downloads/release/python-341/ 下載給 Windows 使用的 Python。在下載了一個 ***.msi** 檔案之後，你就直接雙擊左鍵運行他，然後跟著提示的步驟做。還有一件很重要的事情，一定要記住你安裝 Python 的路徑，我們稍後會需要這個路徑！

Linux

通常你已經內建 Python 了。檢查看看你是不是已安裝（以及它的版本），打開終端機然後輸入下面這個指令：

```
$ python3 --version
Python 3.4.1
```

如果你沒有已安裝的 Python 或是你的版本不一樣，你可以如下步驟來安裝：

Ubuntu

在你的終端機輸入以下指令：

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

Fedora

在你的終端機輸入以下指令：

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

OS X

你需要到這個網站 https://www.python.org/downloads/release/python-341/ 下載 Python 安裝檔:

- 下載這個 *Mac OS X 64-bit/32-bit installer DMG* 檔案,
- 雙擊點開它,
- 雙擊 *Python.mpkg* 運行這個安裝檔.

打開你的 終端機*(terminal)* 驗證安裝過程是否有成功，像這樣子運行 `python3` 指令：

```
$ python3 --version
Python 3.4.1
```

如果你有什麼問題或是有什麼奇怪的錯誤讓你不知道該如何繼續下去 -- 就問你的教練吧！請教一個經驗豐富的人有時可以讓事情更順利更完美！

初試 Python

> 部分章節是來自怪咖女孩 Carrots (http://django.carrots.pl/)

我們來寫些程式碼吧！

Python 的命令提示字元

開始玩 Python 之前，我們需要在電腦上打開命令行視窗（就是終端機）。你應該已經知道那是什麼了 -- 你已經學過了 命令行簡介 一節。

一旦你準備好了，就按著下面的步驟來吧。

現在，我們想打開 Python 自己的主控台(console)，讓我們輸入 `python3` 然後按下 Enter。

```
$ python3
Python 3.4.1 (...)
Type "copyright", "credits" or "license" for more information.
>>>
```

你的第一個 **Python** 指令!

在運行了 Python 指令後，命令提示字元就改變成 `>>>` 了。對我們來說這是代表現在我們只能在這裡使用 Python 語言的指令。你不需要輸入 `>>>` - Python 本身已經幫你輸入好了。

如果你隨時想要離開 Python 主控台的話，就輸入 `exit()` 或是在 Windows 環境下按下熱鍵 `Ctrl + Z`，在 Mac/Linux 環境下按下熱鍵 `Ctrl + D` ，然後你就不會再看見 `>>>` 了。

但現在，我們還不想離開 Python 主控台，我們想要學習更多關於它的事。讓我們從最簡單的開始，舉例來說，試著輸入一點數學，像是 `2 + 3` 然後按下 Enter。

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

幹得好！現在看看視窗跳出什麼來了？Python 懂數學耶！你可以嘗試其他指令像是：

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

放輕鬆隨意玩會兒，待會再回來學習新進展 :)

如你所見，Python 是很棒的計算機。如果你還好奇你還可以拿它來做些什麼...

字串

那輸入你的名字呢？你可以像下面這樣試試看：

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

你現在已經創造了你的第一個字串！這是一個可以被電腦運用的字元集合。字串通常會存在以一個同樣的字元來起頭和結束，或許是一個單引號 (`'`) 或者雙引號 (`"`) - 這會讓 Python 知道引號中的東西是一個字串。

字串們也可以被串在一起，試試看這招：

```
>>> "Hi there " + "Ola"
'Hi there Ola'
```

你也可以用一個數字來把字串乘起來：

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

還有，如果你想要放一個單引號在你的字串裡面，你有兩種方式可以達成這個目標。

使用雙引號來宣告字串：

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

或者，在想要加入的單引號前面給他一個反斜線 (`\`):

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

不賴吧？去看看如何把你的名字全部換成大寫，簡單的輸入這個：

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

你只需要用 `upper` 這個 函數**(function)** 在你的字串後面就可以了！一個函數（像是 `upper()` ）就是一個指令集，讓 Python 可以在這個 (`"Ola"`) 物件(object)中展現你呼叫這個函數的效果。

如果你想要知道你的名字總共有幾個字元，也有相對應的函數可以叫用！

```
>>> len("Ola")
3
```

你應該很好奇，為什麼有時候你會在字串結尾用 `.` 來叫用一個函數（例如 `"Ola".upper()` ），但有時你會先呼叫函數，然後把字串放在括號中？嗯，在某些狀況下，函數屬於某個物件，像是 `upper()` ，這個函數就只能表現在字串上。在這個狀況下，我們可以稱呼這類型函數為 方法**(method)**。其他時候，函數不屬於任何特定的類型或物件，可為各種不同類型或物件所用，像是 `len()` 。這是為什麼我們可以在 `len()` 這個函數中放入一個 `"Ola"` 字串作為參數。

總結

好了，說夠了字串。目前為止你學到了：

- 命令提示字元**(prompt)** - 在 Python 的命令提示字元下輸入指令（程式碼）來得到結果
- 數字**(number)**及字串**(string)** - 在 Python 中數字可計算，而字串表示一個文字物件
- 運算子**(operator)** - 像是 + 與 *, 可結合兩個數值進行運算產生一個新的值
- 函數**(function)** - 像是 upper() 與 len(), 讓物件展現出某種行為

你剛剛學到的是所有程式語言的基礎。準備好接受更多挑戰了嗎？來吧！

錯誤訊息

我們來試試新的東西。看看我們能不能使用找到我們名字長度的那個函數，來得到某個數字的長度呢？輸入 `len(304023)` 按下 Enter:

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

恭喜！我們得到了我們的第一個錯誤訊息！它抱怨說這個物件的類型是 "int" (integer, 整數) 所以根本沒有長度這件事。所以我們現在應該怎麼辦呢？或許我們可以把我們這個數字寫成一個字串看看如何？字串就有長度啦，對吧？

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

看起來成功了！我們在 `len` 函數中使用了 `str` 這個函數去把任何物件轉成字串。

- `str` 函數可將物件轉為 字串
- `int` 函數可將物件轉為 整數

> 重要：我們可以轉數字為字串，但我們不必然能把文字轉為數字 - 想想看要是 `int('hello')` 會有怎樣的結果？

變數

一個重要的程式觀念叫做變數(variables)。一個變數其實沒什麼，就是一個你稍後可以叫用的一個名字。程序員使用這些變數去儲存資料，讓他們的程式碼有更高的可讀性，這樣他們就不必一直記得那些資料是什麼。

所以我們就說好吧，現在我們做了一個新的變數叫做 `name`:

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

你看吧？就這麼簡單！其實只代表一件事情：這個叫做 name 的變數等於 "Ola" 這個字串。

如同你所注意到的，你的程式沒有回傳任何你宣告的東西。所以我們如何得知我們所宣告的變數確實存在呢？簡單的輸入 `name` 然後按下 Enter:

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

呀比！你的第一個變數啊啊啊 :)！你可以隨時改變它的值像這樣：

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

你也可以在函數中使用它：

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

太完美了，對吧？當然，變數可以是任何東西，數字亦然！試試看這個：

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

但是萬一我們叫錯變數了呢？你可以猜到會發生什麼事情嗎？讓我們試試看吧！

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

又是一個錯誤訊息！如同你所見，Python 有各式各樣的錯誤訊息，這個就叫做一個 **NameError**。Python 會在你試圖使用一個從未宣告的變數時給你這個錯誤訊息。如果你得到像這樣的錯誤，就回頭看看你的程式碼裡面是不是有什麼變數的名字是打錯的。

玩一會兒吧，看看你還能做什麼！

Print（印出） 函數

試試這招：

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

當你只是輸入 `name`， Python 直譯器會馬上回應你變數 `name` 所代表的字串，就是個用單引號包起來的 M-a-r-i-a 字母們。當你說 `print(name)` ， Python 就會在螢幕上「印出」這個變數的乾淨內容，沒有引號。

就像是我們稍後會看到的， 當我們想要印出某些東西時，或是想要印出一個有很多行的東西時， `print()` 就是個超有用的函數。

串列(List)

除了字串和整數，Python 還有一堆不同形態的物件。現在我們要介紹的稱為 串列**(list)**。List 就是你想的那樣：它是一個列出一堆物件的物件 XD

開始建立一個 List 吧：

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

對沒錯，這就是一個空的 List。看起來沒什麼用啊... 我們來建立一組包含樂透彩數字的 List 吧。我們不想每次都重覆輸入這個長的 List，所以我們就把它存進一個變數裡吧：

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

好了，我們有一個 List 了！我們可以拿它來幹嘛？來看看這個 List 的長度吧。你想到可以用什麼函數了嗎？沒錯就是它！

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

賓果！ `len()` 可以告訴你這個 List 有多長。真是太神奇了傑克！或許我們還可以幫這個 List 排序：

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

這不會回傳任何值，只是默默的的改變了 List 中數字們的順序。我們把它印出來看看發生了什麼事吧：

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

如你所見，這串 List 數字已經從小到大排序了。恭喜囉！

那如果我們想要反排序呢？試試看吧！

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

很簡單對吧？如果你還想替 List 新加一點東西，可以輸入這個指令：

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

如果你只想要秀某一個數字，你可以使用 索引**(index)** 來指定。索引就是一個指出 List 某個物件位置的數字，電腦宅們喜歡讓索引從 0 開始，所以你的 List 中的第一個物件的索引就是 0，下一個物件則是 1，以此類推，試試看這些：

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

如你所見，你可以呼叫 List 變數的名字並指定索引值，並將索引值放在中括號 `[]` 中，來取得 List 裡不同的物件。

還有更多好玩的，試試看這些索引值： 6, 7, 1000, -1, -6 or -1000。看看你能不能在輸入指令前就預測到結果，這些結果合理嗎？

你可以在這個 Python 文件章節中，找到所有支援 List 物件的方法
(method)：https://docs.python.org/3/tutorial/datastructures.html

字典(Dictionaries)

Dictionary 和 List 有點像，差別在於，你可以用鍵值(key)而非索引值(index)來取得在 Dictionary 中的值(value)。一個 key 可以是字串或數字。你可以使用兩個大括號來宣告一個 dictionary：

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

這樣就創建了一個空字典，喲齁！

現在，試著輸入下面指令（你也可以試著輸入你自己的資訊）：

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

使用這個指令，你就等於是宣告了一個叫做 `participant` 的 key-value 結對變數：

- `name` 這個 key 對應到 `'Ola'` 這個 value (一個 `字串(string)` 物件),
- `country` 對應到 `'Poland'` (另一個 `字串(string)`),
- 還有 `favorite_numbers` 對應到 `[7, 42, 92]` (一個包含了三個數字在內的 `串列(list)`).

你可以指定特殊的 key 來檢查內容，語法如下：

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

你看，和 List 有點像。不過你就不需要再記住索引值(index)了 - 就變成一個好記的名稱。

如果你向 Python 要一個不存在的 key 所對應的 value 的話會發生什麼事呢？我們試試看吧！

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

看到了吧，又是一個錯誤訊息，這次是 **KeyError** 。Python 會讓你知道這個 `'age'` key 事實上是不存在目前的 dictionary 中的。

什麼時候要用 List，什麼時候又要用 Dictionary 呢？嗯，這個問題值得深思。再看看下面內容前先把這個問題放在心裡吧。

- 你是不是只是需要一個有序的序列呢？就使用 List 吧。
- 你需要有 key 值的 value，好讓你可以更有效率地找到特定值嗎？那就使用 Dictionary 吧。

Dictionary 也像 "List" 一樣，可以默默地把它裡面的東西做點改變，你可以加入一對新的 key/value 值像這樣：

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

如同 List，Dictionary 也可以使用 `len()` ，回傳 Dictionary 總共有幾對 key-value 值吧，輸入下面指令：

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

我希望目前為止對你來說都還可以了解。 :) 準備好玩玩更多有趣的 Dictionary 了嗎？跳到下一行看看一些很酷的事情吧。

你可以使用 `del` 指令來刪除 Dictionary 中的項目。使用它，如果你想要刪掉一個叫做 `'favorite_numbers'` 的 key，就輸入下面指令：

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

輸出結果如你所見，符合 'favorite_numbers' 的 key-value 數對已經被刪除了。

除此之外，你也可以改變已經建立的 Dictionary 中的特定值，輸入：

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

一樣如同你所看見的，一個 key 值為 `'country'` 的值從 `'Poland'` 變成了 `'Germany'`。 :) 興奮嗎？喲齁！你剛剛學到更多超棒的知識了！

總結

太完美了！你現在了解更多程式設計的知識了。在上面你學到了：

- 錯誤訊息 - 當 Python 看不懂你給的指令時，你現在知道怎麼去閱讀並且了解錯誤訊息的意思了
- 變數 - 這是物件的名字，讓你可以更輕鬆的寫程式，也讓你的程式更具可讀性
- 串列 - 可以用特定順序來儲存多個物件的清單

人生要進階了，興奮嗎？ XDDDD

比對(compare)

程式中有很大一部份都在比對東西。什麼是最容易被比對的東西呢？當然是數字。我們來看看這裡怎麼做的：

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

我們給 Python 幾個數字去互相比對。如同你看見的，Python 不僅可以比對數字，甚至可以比對不同方法產生的結果，酷斃了對吧？

你說不定也會好奇為什麼我們寫了兩個等號 `==` 就可以比對兩個數字是不是相等，我們用單一個等號 `=` 去給變數賦值。如果你想要知道兩個東西是不是相等，你通常， 通常 需要放兩個 `==` 在兩個東西之間。

給 Python 更多工作：

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

`>` 及 `<` 很直觀，不過 `>=` 和 `<=` 是什麼意思? 其實事情是這樣的：

- x `>` y 表示: x 大於 y
- x `<` y 表示: x 小於 y
- x `<=` y 表示: x 小於等於 y
- x `>=` y 表示: x 大於等於 y

很棒吧！想知道更多嗎？試試這些：

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

你儘可以給 Python 所有你想比對的數字，它都會給你答案！非常聰明對吧？

- **and** - 如果你使用 `and` 運算子，會回傳給你一個交集（所有結果都必須為 `true` 才是 `true）
- **or** - 如果你使用 `or` 運算子，會回傳給你一個聯集（只要其中一個結果都必須為 `true` 就是 `true）

不知道你有沒有聽過一個諺語「牛頭不對馬嘴」（按：原文為 "comparing apples to oranges"）呢? 讓我們來試試下面的等式吧！：

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

在此你看到在這個表達式中，Python 無法用數字 (`int`) 來與字串 (`str`) 做比較。 此外，它給了一個 **TypeError** 並告訴我們這兩種型態無法放在一起比較。

布林值(Boolean)

很快的，你學到了一個新的 Python 物件型別。這稱為 布林值**(Boolean)** -- 這或許是一種最簡單的型別。

這裡是兩個 Boolean 物件

- True
- False

為了讓 Python 知道那是布林值，你必須把它寫成像這樣 True（字首大寫，其餘小寫）。**true, TRUE, tRUE** 都不會有作用 **--** 只有 **True** 是正確的。 （當然囉，False 亦同）

Boolean 也值可以存成變數！像這樣：

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

你也可以這麼做：

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

玩一玩練習一下 Boolean 值，運行下面的指令：

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

恭喜！你現在要進入到程式設計裡面最主要的一個段落了：

If...elif...else

在程式運行時，很多時候就只是在判斷一些設定條件是不是符合而已。這就是為什麼 Python 中有個東西叫做 **if** 判斷式 。

試一下：

```
>>> if 3 > 2:
...
```

目前為止，除了本來應該是 `>>>` 的地方出現了點點點 `...` 以外，Python 毫無反應。Python 預期我們會給出更多的說明來讓它執行一個 if 的 `3 > 2` 成立的條件式。我們來讓 Python 印出 "It works!"：

```
>>> if 3 > 2:
... print('It works!')
  File "<stdin>", line 2
    print('It works')
        ^
IndentationError: expected an indented block
```

呃... 這裡出了一些錯！Python 需要知道這些說明是不是接續在 `if` 敘述之後，或者這是ㄌ外一個與條件式無關的段落。我們必須縮排這行 code 讓它可以運行：

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

然後你又有一行在 `...` 後面的空白了。為了避免雜亂無章，多數的 Python 程序員習慣使用 4 個空白作為縮排。

所有在 `if` 判斷式後面的程式碼都縮排了，在條件符合的狀況下都會被執行。看吧：

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

如果我們不想要這樣呢？

在前一個範例中，這些程式碼只會在條件是成立的時候被執行。但 Python 還有 `elif` 與 `else` 判斷式：

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

如果 2 大於 5，那第二段指令就會被執行。很簡單吧？我們看看 `elif` 如何運作：

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

看看會發生什麼事吧？

總結

在剛剛的三個練習中，你已經學到：

- 比對 - 在 Python 中你可以用 `>`, `>=`, `==`, `<=`, `<` 和 `and`, `or` 來比對東西
- **Boolean** - 一種只有兩種值的物件型別： `True` 或 `False`
- **if...elif...else** - 判斷式讓你可以在條件符合的狀況下執行相應的程式碼

現在是時候進入到最後一章了！

你自己的函數!

還記得你先前在 Python 中執行過像是 `len()` 這樣的函數嗎？嗯，好消息，你將可以學習如何寫一個屬於你自己的函數囉！

一個函數就是一堆 Python 應該執行哪些行為的集合。在 Python 中每個函數都用一個關鍵字 `def` 來宣告，可以被賦予一個名稱與一些參數。我們來做一個簡單的：

```
>>> def hi():
...
```

如你所見，這裡又出現點點們了！這代表你還可以繼續說些話... 以及沒錯，在做更多描述錢，我們需要按幾次空白鍵：

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

好了，我們的第一個函數完成了！按下 Enter 再次回到 Python 命令提示字元。現在我們來執行我們的函數：

```
>>> hi()
Hi there!
How are you?
```

棒極了！你現在是個程序員了，恭喜一下自己吧 :)！

這真是簡單！讓我們來建立自己的第一個有參數的函數吧。我們將會使用前一個範例 - 一個會對某個人說 'hi' 的函數，並且執行它 - 並帶有一個名字：

```
>>> def hi(name):
...
```

如你所見，我們現在讓這個函數帶有一個叫做 `name` 的參數：

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

這裡你可以看到，我們需要放兩個縮排在 `print` 函數前面了，因為 `if` 需要知道接下來的條件符合的話會發生什麼事。我們來看看現在他運作得如何：

```
>>> hi()
    Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: hi() missing 1 required positional argument: 'name'
```

喔喔，一個錯誤訊息。很幸運的，Python 給我們一個非常有用的錯誤訊息。它告訴我們這個叫做 `hi()` 的函數（就我們剛剛定義的那個）要求必須傳入一個參數（就是 `name`），只是我們叫用這個函數的時候忘記忘記傳入了。

我們把它修好吧：

```
>>> hi("Ola")
Hi Ola!
>>> hi("Sonja")
Hi Sonja!
>>> hi("Anja")
Hi anonymous!
```

太完美了！這下子當你需要換個人名說 hi 的時候，你就不需要一直重複做一樣的事情。這就是為什麼我們如此需要函數 - 你永遠不會想要重複你的程式碼！

我們再來做點更聰明的事 -- 如果我們有兩個以上的人名，寫兩次差不多的句子真的很累對吧？

```
>>> def hi(name):
...     print('Hi ' + name + '!')
...
```

現在我們可以叫用這個函數了：

```
>>> hi("Rachel")
Hi Rachel!
```

恭喜！你剛剛學到如何寫一個函數了 :) ！

迴圈(Loop)

現在就是最後一步了，進展滿快的呢 :)

就像我們先前所提到的，程序員很懶，他們很討厭重複自己。程式設計應該是一件完全自動化的事情，所以我們不希望手動去對每個不同的人名說嗨，沒錯吧？所以迴圈的出現非常好用。

還記得 List 嗎？我們來做一個女孩 List:

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

我們想要對每個人的名字說嗨。我們已經有一個 `hi` 函數可以用了，我們讓我們把這個函數放在迴圈之中：

```
>>> for name in girls:
...
```

噢又是點點們！記得點點出現時要做什麼事情嗎？沒錯，縮排 :)

```
>>> for name in girls:
...     hi(name)
...     print('Next girl')
...
Hi Rachel!
Next girl
Hi Monica!
Next girl
Hi Phoebe!
Next girl
Hi Ola!
Next girl
Hi You!
Next girl
```

如你所見，你利用縮排將一些程式碼放在 for 判斷式之中都會重複執行，而且會逐次帶入 List 中的女孩名字。

你也可以用 for 來帶入一個數字範圍：

```
>>> for i in range(1, 6):
...     print(i)
...
1
2
3
4
5
```

`range` 是個可以建立兩個數字範圍中所有數字 List 的函數（這兩個數字來自由你提供的參數）

要注意的是，在 Python 中，range 會輸出第二個參數的前一個整數（就是說如果有一個函數 `range(1, 6)` 就會輸出 1 到 5，但不會包含數字 6）。

離開 **Python** 主控台

你可以用指令 `exit()` 來離開 Python 主控台。

總結

這就是全部了。 你真是酷斃了！ 這真的很不簡單，所以你可以為你自己鼓鼓掌。我們也超級為你可以走到這裡感到驕傲！

進行到下一章之前抓個杯子蛋糕來吃吧 :)

什麼是 **Django?**

Django (*/ˈdʒæŋɡoʊ/ jang-goh*) 是個用 Python 寫成的，免費而且開放原始碼的 Web 應用程式框架。他是個 Web 框架 - 就是一堆零件的組成，可以幫助你輕鬆快速的開發網站。

這麼說吧，當你蓋一個網站的時候，你總是需要一些很類似的零件：使用者登入（註冊、登入、登出），網站後台，表單，檔案上傳等等。

幸運的世界上有很多人很久以前就幫你注意到這件事，Web 開發者蓋一個新的網站的時候總是面對著一樣的問題，所以他們合作開發了框架使你可以直接擁有你會用到的零件（Django 就是其中之一）。

各種框架的存在就是拯救你免於重造輪子，當你新建一個網站的時候可以減少重工。

為什麼你需要一個框架？

要真正了解 Django 事實上到底在做什麼，我們需要更了解伺服器。首先就是伺服器需要知道你想要如何來提供你的網頁。

想像一個信箱（埠 port）會偵測寄來的信（請求 request）。Web 伺服器就是在做這件事。Web 伺服器讀這些信，然後寄出相對的網頁作為回應。但是當你想要寄出某個東西，你需要一些內容，而 Django 就是再幫你產生相對應的內容。

當某個人向你的伺服器請求的時候發生了什麼？

當伺服器收到某個請求，這個請求會通過 Django，Django 則負責判斷這個請求是什麼。Django 會先收到網址然後來判斷應該要給出什麼回應。這部分是由 Django 的 **urlresolver** 來處理（網址其實就是所謂的 URL - Uniform Resource Locator，所以這就是為什麼這裡取名為 *urlresolver* ）。它不聰明 - 他有一堆範例去判斷這個 URL 符合哪一個範例。Django 則查看範本，如果 URL 符合某一個，Django 就送出這個請求相對應的函數們（在這裡稱為 *view* ）

想像一個帶著信的郵差。她在街上遊走，確認每家的地址把信送給他們，如果地址對了，她就把信放進去，這就是 urlresolver 在做的事情。

在 *view* 函數中會做一些有趣的事情：我們會去資料庫中找某些資料。萬一使用者要求要更改某些資料呢？例如某封請求「拜託把我的工作敘述改一下吧」的信 - *view* 就會檢查你是不是允許他可以做這件事，然後你會在更新了他的工作敘述以後回傳給他一個「完成囉！」的訊息。之後 *view* 就會產生一個回應，Django 就會將回應送到使用者的瀏覽器上。

當然了，以上的敘述已經簡化了很多，但你也不需要知道所有技術上的細節。有一個簡單的概念即可。

所以不糾結在太多細節敘述上，我們打算就簡單的開始用 Django 做點事，就可以從中學習到更多了。

安裝 Django

虛擬環境

在我們安裝 Django 之前，我們會讓妳安裝一個超有用的工具，這可以讓你電腦中的 coding 環境保持乾淨。也可以跳過這個步驟，但是我們高度建議你不要 - 用最好的安裝方式起手可以節省你未來碰到許多麻煩的時間！

所以我們來創建一個 虛擬環境**(virtual environment)** 吧（也拼做 *virtualenv*）。這會將你的 Python/Django 獨立為一個專案形態，表示你在一個網站專案所做的任何改變都不會影響到你同時在進行的其他網站專案，超乾淨的對吧！

你所需要做的事就是找到一個目錄去創建 這個 `virtualenv`；舉例而言，你的家目錄 (`/home`)，在 Windows 環境下會是 `C:\Users\Name\` （ `Name` 會是你目前登入的使用者名稱）。

在這個教程中我們會在你的家目錄下使用一個新目錄 `djangogirls` ：

```
mkdir djangogirls
cd djangogirls
```

我們會將會創建一個虛擬環境叫做 `myvenv`。這個指令基本的格式如下：

```
python -m venv myvenv
```

Windows

創建一個新的 `virtualenv`，你需要打開終端機（我們已經在前面一些章節中告訴過你了 - 記得嗎？），並且執行 `C:\Python\python -m venv venv`。這會看起來像這樣

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

`C:\Python34\python` 是你之前安裝 Python 的路徑，而 `myvenv` 則是你的 `virtualenv` 的名稱。你可以取自己喜歡的名稱，不過必須要是小寫並且沒有其他空白。保持名稱簡短是好主意 - 因為你將會常常提起它！

Linux and OS X

在 Linux 與 Mac 系統創建一個 `virtualenv` 就是很簡單的執行一下 `python3 -m venv myvenv`。看起來像這樣：

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

`myvenv` 則你的 `virtualenv` 名稱。你可以取自己喜歡的名稱，不過必須要是小寫並且沒有其他空白。保持名稱簡短是好主意 - 因為你將會常常提起它！

> 小提示**:** 在 Ubuntu 14.04 下初始化虛擬環境目前會給你以下的錯誤訊息：
>
> ```
> Error: Command '['/home/eddie/Slask/tmp/venv/bin/python3', '-Im', 'ensurepip', '--upgrade', '--default-pip']' returned non-zero exi
> ```
>
> 將指令改為 `virtualenv` 可以避免掉這個問題
>
> ```
> ~/djangogirls$ sudo apt-get install python-virtualenv
> ~/djangogirls$ virtualenv myvenv
> ```

在虛擬環境下工作

以上的指令會創建一個叫做 `myvenv` 的目錄（或是你自己決定的任何名稱），裡面就是我們的虛擬環境。我們現在想要做的就是把這個虛擬環境執行起來：

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

在 Windows 上, 或者:

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

在 OS X 與 Linux 上.

記得，如果你取了自己喜歡的名稱，就把指令中的 `myvenv` 代換掉！

> 小提示: 有時候你系統中或許不支援 `source` 指令，在這個狀況下你可以使用以下方式：
>
> ```
> ~/djangogirls$ . myvenv/bin/activate
> ```

當你看到你的終端機的命令提示字元看起來像是下面這樣的時候，你就知道你現在是在 `virtualenv` 中：

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

或這樣:

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

`(myvenv)` 總是會出現！

當你在虛擬環境下工作時， `python` 會自動切換到目前所使用的版本，所以你就可以用 `python` 而不需要再輸入 `python3`。

好了，我們已經有了所有相關的套件了，我們終於可以安裝 Django 了！

安裝 Django

現在你應該已經啟動了你的 `virtualenv`，你可以用 `pip` 來安裝 Django。在終端機底下，執行 `pip install django==1.6.6` （注意噢，我們是使用雙等號： `==` ）。

```
(myvenv) ~$ pip install django==1.6.6
Downloading/unpacking django==1.6.6
Installing collected packages: django
Successfully installed django
Cleaning up...
```

> 如果當你在 Ubuntu 12.04 下呼叫 `pip` 時出現錯誤訊息，請執行 `python -m pip install -U --force-reinstall pip` 去修復這個 pip 在 virtualenv 下的安裝問題。

這就是全部了！你現在（終於）準備好去創建一個 Django 應用程式！但在這之前，你需要一個好的軟體讓你可以好好寫程式...

程式碼編輯器

你就快要寫你的第一行程式碼了，所以現在正是時候來下載一個程式碼編輯器！

其實世界上有很多不同的編輯器，而且很大一部份都是取決於個人喜好。大部份的 Python 程序員習慣使用複雜但超級強大的 IDE (ntegrated Development Environment 集成開發環境)，像是 PyCharm。身為新手，還是不要好了 - 那大概不是很適合你；我們的推薦是選擇足夠強，但更簡單的。

我們的建議如下，你隨時都可以問問你身邊的教練他喜歡哪一種 - 這對你要下決定會更有幫助。

Gedit

Gedit 是一款開放原始碼、免費的編輯器，適用於所有的作業系統。

這裡下載

Sublime Text 2

Subline Text 是一款非常流行的編輯器，有免費試用期。它易用、易安裝，而且也同樣適用於所有的作業系統。

這裡下載

Atom

Atom 是一款由 GitHub 出品的，非常年輕的程式碼編輯器。它免費並且也開放原始碼，同樣易用易安裝，但是目前僅有 beta 版本，只適用於 Windows 與 OS X。不久的未來也會在 Linux 上釋出 beta 版。

這裡下載

你的第一個 Django 專案！

> 部分章節是來自怪咖女孩 Carrots (http://django.carrots.pl/)
>
> 部分章節是來自 django-marcador tutorial licensed under Creative Commons Attribution-ShareAlike 4.0 International License. The django-marcador tutorial is copyrighted by Markus Zapke-Gründemann et al.

我們要開始來做一個簡單的 blog 了！

要新增一個 Django 專案的第一步。基本上，這代表我們將會運行由 Django 提供的某些腳本，它會幫我們建立一個 Django 專案的基本架構：一堆我們待會會用上的檔案夾和檔案。

這些檔案夾和檔案們的命名對 Django 來說是非常重要的。你不應該重新命名這些我們剛剛弄出來的檔案。將這些檔案移動到別的地方也不是好主意。為了可以容易找到重要的事物，Django 必須保持這些檔案結構。

在終端機下你應該執行（應該記得吧，你不需要再輸入 `(myvenv) ~/djangogirls$` 囉）：

> 記得要在虛擬環境裡執行所有動作。如果你的終端機命令列沒有看到前綴 `(myvenv)` 時，你就需要重新喚起你的虛擬環境。我們已經在 安裝 *Django* 這章的 在虛擬環境下工作_ 中解釋過如何做囉。

在 Windows 下執行：

```
(myvenv) ~/djangogirls$ python myvenv\Scripts\django-admin.py startproject mysite .
```

或者在 Linux 或 Mac OS 下執行：

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

`django-admin.py` 是一個可以替你創建資料夾與檔案的腳本(script)。你現在應該有像是下面這樣的資料目錄了：

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

`manage.py` 是個協助管理這個網站的腳本。有了它我們將可以在我們的電腦上叫起一個 web server，不需安裝其他的東西。

至於 `settings.py` 則包含了你的網站中的各種配置。

記得當我們談過一個郵差如何檢查信要送到哪裡嗎？ `urls.py` 就是一個有很多範例的清單，被 `urlresolver` 所用。

現在讓我們先忽略其他檔案 - 我們不會改變那些。唯一需要記得的一件事就是不要不小心刪了它們 XD

改變設定

我們來在 `mysite/settings.py` 中做點改變。用你早先安裝的程式碼編輯器打開這個檔案。

如果我們網站中有正確的時間是件好事。到 http://en.wikipedia.org/wiki/List_of_tz_database_time_zones 這個維基頁面去複製你所在時區吧（TZ - time zone）。（例如這裡是 `Asia/Taipei` ）

你應該可以找到幾行包含了 `USE_TZ` 與 `TIME_ZONE`， 像這樣修改它，代入你的時區的資料來源 `Asia/Taipei` ：

```
USE_TZ = False
TIME_ZONE = 'Asia/Taipei'
```

設定資料庫

這裡有好幾種不同的資料庫可以為你的網站儲存資料。我們將使用內定的那個， `sqlite3` 。

這已經在你的 `mysite/settings.py` 檔案中設定好了：

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

要為我們的部落格創建一個資料庫，我們要在終端機下執行這個： `python manage.py syncdb` (我們需要在已經建好了 `manage.py` 檔的 `djangogirls` 目錄下)。如果一切順利，你應該可以看到像這樣的東西：

```
(myvenv) ~/djangogirls$ python manage.py syncdb
Creating tables ...
Creating table django_admin_log
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_groups
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session

You just installed Django's auth system, which means you don't have any superusers defined.
Would you like to create one now? (yes/no): yes
Username (leave blank to use 'Name'):
Email address: admin@example.com
Password:
Password (again):
Superuser created successfully.
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

它會問你想不想創建一個 超級使用者*(superuser)* - 就是一個可以控制這個網站上所有功能的使用者。輸入 `yes` ， 按下 Enter 並且輸入你的使用者名稱（小寫無空白）， email 帳號以及密碼（必填）。記住這組帳密！我們稍後會用到。

這樣我們就完成了！現在是時候叫起 web server 然後看看我們的網站怎麼樣！

你一樣需要在這個包含了 `manage.py` 檔案的 `djangogirls` 目錄下。在終端機下，我們可以開始用 `python manage.py runserver` 來叫起我們的 web server 了：

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

至此，你需要做的就是檢查你的網站是不是有運作起來 - 打開你的瀏覽器（火狐，Crhome, Safari 甚至是 IE 或是任何你有在用的）然後輸入這個網址：

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

你可以再次停掉這個 web server 了（就是說在命令提示字元下輸入其他指令）， 藉由按下 CTRL+C - Control 和 C 鍵一起按下去。

恭喜恭喜！你剛剛就建好了你的第一個網站，並且用 web server 運作起來了！好 Django 不學嗎？

準備好進入下一階段了嗎？是時候加入一些內容了！

Django models

我們現在想要新建立的東西，將可以儲存我們部落格中的所有發文。為了這個目的我們需要講一點點小東西，這個東西被稱為 `物件(object)` 。

物件 (Objects)

有一個被程序員們稱為 `物件導向程式設計 (Object-oriented programming)` 的概念。這個想法是說，無論是什麼東西，即便只是一個程式敘述無趣至極的序列，我們都可以把它模型化，並且定義它如何與其他東西互動。

所以什麼是物件？就是一個把性質與行為量化的東西。這聽起來很詭異，我們用一個例子來解釋。

如果我們要模擬一隻貓，我們就會建立一個叫做 `Cat` 的物件，並且這個 `Cat` 會有一些屬性。例如說， `毛色` 、 `貓齡` 、 `心情` (像是「好心情」、「壞心情」、「昏昏欲睡」等等)， `飼主` (這或許就是一個 `Person` 物件，如果是一隻走失的貓咪，這個屬性應該就是空的)。

然後這隻貓會有一些行為： `發出呼嚕聲` 、 `磨蹭` 或者 `被餵食` (這時我們會給貓咪一些 `貓食` ， 當然這個貓食也應該要被獨立成另外一個擁有自己的屬性的物件，像是 `口味`)。

```
Cat（貓咪）
--------
color （毛色）
age （貓齡）
mood （心情）
owner （飼主）
purr() （發出呼嚕聲()）
scratch() （磨蹭()）
feed(cat_food) （被餵食(貓食)）


CatFood （貓食）
--------
taste （口味）
```

所以說基本上這個想法可以在程式碼中描述真實的物品，藉由一些屬性（稱為 `物件屬性 object properties` ） 與行為（稱為 `方法 methods` ）。

那我們如何去把部落格中的文章模型化？我們想要建立一個部落格對吧？

我們需要先思索一個問題：什麼是網誌？它擁有哪些屬性？

嗯，可以確定的是我們的網誌一定會有一些文字，像是內容與標題，對吧？能知道這篇網誌是誰寫的也是一個好主意 - 所以我們還需要作者。最後，我們想要知道何時新增與發表了這篇網誌。

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

那網誌又可以做什麼樣的事情呢？如果可以有一些 `method` 來發佈網誌就太好了對吧！

所以我們也需要 `publish (發佈)` 方法。

既然我們已經知道有哪些需要做的事情，我們就可以開始用 Django 模擬它了！

Django model

了解了何謂物件，我們就來做一個我們的網誌的 Django model 吧。

Model 在 Django 中是一種特別類型的物件 - 它被儲存在 `資料庫 database` 中。一個資料庫就是一堆資料的集合。這裡你可以儲存使用者的資訊、你的網誌等等。我們將使用 SQLite 資料庫來儲存我們的資料。這是 Django 內建的資料庫連接器 -- 這夠我們用了。

你可以將一個 model 想像成在資料庫裡面的一個有行(rows - 一筆資料)與列(columns - 欄位)的電子表格。

創建一個應用程序

為了保持一切整潔乾淨，我們會將一個獨立的應用程序創建在我們的專案中。如果可以從一開始就讓一切井然有序最好了。我們可以在終端機中執行以下指令來新建一個應用程式（在有 `manage.py` 檔的 `djangogirls` 資料夾下）：

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

你會發現多了一個新建的 `blog` 資料夾，裡面有一些檔案。我們在這個專案下的的整個資料目錄就變成這樣子了：

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

在創建了一個應用程序後我們也需要告訴 Django 說這個應用程序要用上它。我們在 `mysite/settings.py` 這個檔案中設定。我們需要找到 `INSTALLED_APPS` 然後在 `(` 內加上一行 `'blog',` 。所以最終產物就會長得像這樣。

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

建立一個網誌 model

我們定義所有叫做 `Model` 的物件在 `blog/models.py` 檔中 - 這禮我們可以定義我們的網誌。

讓我們打開 `blog/models.py` ，移除裡面所有的東西，並寫下這樣的程式碼：

```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User

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

看起來豪可怕啊對嗎？但別擔心，我們會一行一行的解釋給你聽！

所有行首是 `from` 或 `import` 的程式碼就是引進其他檔案的程式。所以除了複製貼上一堆一樣的程式碼以外，我們也可以用 `from... import...` 來把檔案內容載入進來。

`class Post(models.Model):` - 這行定義了我們的 model (它就是一個 `object`)

- `class` 是由我們定義的一個物件，是一個保留字。
- `Post` 是這個 model 的名稱，我們可以給各種不同的名字（但必須避免特殊自元與空白）。這裡的字首永遠大寫。
- `models.Model` 這意指 Post 是一個 Django Model, 這樣 Django 就會知道要這個東西要存進資料庫。

現在我們定義我們剛才提到的各種性質： `title` 、 `text` 、 `created_date` 、 `published_date` 與 `author` 。為了做這件事我們必須要定義資料形態（它是文字嗎？還是數字？是個日期嗎？是個關聯性物件嗎，像是 User 這類的？）

- `models.CharField` - 這裏你可以定義這個資料是一個有上限的字元。
- `models.TextField` - 這裏你可以定義這個資料是一個有無上限的字串。這適合被部落格網誌的內容所用，對吧？
- `models.DateTimeField` - 這是是日期與時間。
- `models.ForeignKey` - 這是與其他 model 的關聯。

我們不解釋所有的程式碼，因為會花上太多時間。你應該看一下 Django 的文件 (https://docs.djangoproject.com/en/1.6/ref/models/fields/#field-types)，如果你想知道更多關於 Model fields 的事情，或是如何定義其他在此處提到的東西。

那 `def publish(self):` 又是啥？它事實上就是我們之前所提到的 `publish` 方法。 `def` 表示這是一個 函數/方法 `publish` 就是這個方法的名稱。如果你喜歡，你可以改變它。在此處我們用小寫與底線（取代空白）來作為它的命名規則（就是說如果你想要有一個可以計算平均價錢的方法你可以將名稱取為 `calculate_average_price` ）。

方法通常會 `return(回傳)` 一些東西。這裡有個在 `__str__` 方法中的範例。在這個情境下，當我們呼叫 `__str__()` 我們將得到關於網誌標題的一組文字 (**string**)。

如果對於 model 還有什麼不清楚的，馬上問你的教練！我們知道這滿複雜的，特別是你同時學了物件與函數。但希望這個東西現在開始可以讓你擁有神奇魔法！

在資料庫中創建與 model 相關的 table

最後一步是把我們的新 model 加進我們的資料庫 (database)。這很輕鬆，就輸入 `python manage.py syncdb` 即可，看起來會像這樣：

```
(myvenv) ~/djangogirls$ python manage.py syncdb
Creating tables ...
Creating table blog_post
Installing custom SQL ...
Installing indexes ...
Installed 0 object(s) from 0 fixture(s)
```

看到這個 Post model 的感覺真好啊對吧？進入下一個章節看看你的網誌長怎樣吧！

Django ORM 與 QuerySets

在這個章節你將了解 Django 如何連接到資料庫，並且把資料儲存進入。來吧！

什麼是 **QuerySet?**

一個 QuerySet，大體上來說，是一個充滿了 Model 物件的清單。QuerySet 允許你從資料庫中讀取資料，也可以對資料做篩選或排序。

從做中學最簡單明瞭了。現在就來試試看好嗎 :)

Django shell

打開你的終端機輸入以下指令：

```
> $ python manage.py shell
```

結果會像下面這樣：

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

你現在就在 Django 的互動式介面裡了。它就像一般單純的 Python 命令提示字元，但是多了 Django 魔法 :) 當然囉，你可以使用所有的 Python 指令。

所有物件

首先我們來試試展示所有我們的網誌。你可以輸入下面指令：

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

噢喔！出現一個錯誤了。它抱怨說沒有 Post 啊。沒錯 -- 我們忘記先把 Post 載入進來了。

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

這是一個範例：我們從 `blog.models` 載入了 model `Post` 。我們重新來試試秀出所有的網誌吧：

```
>>> Post.objects.all()
[]
```

看起來是個空的 list。這沒錯吧？我們確實還沒有新增任何的網誌！讓我們增加一下。

創造物件

下面示範了你可以怎麼做來增加一個 Post 物件在資料庫裡：

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

但我們好像少了一個要件： `user`。我們需要傳入一個 `User` Model 實體來作為文章作者。怎麼做呢？

先來載入一個 User model：

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

我們資料庫裡面有哪些 users 呢？試試這個吧：

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

酷斃了！我們獲得一個 user 實體了：

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

如你所見，我們知道 `get` 一個 `User` 的 `username` 會是 `ola`。乾淨俐落！當然了，你必須將它改為你的使用者名稱。

現在我們終於可以創一個我們的第一個 post 了：

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

萬歲！想要看看有沒有成功嗎？

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

新增更多 posts

你了解到，新增一些 post 然後看看它有沒有成功就可以從中獲得一點小確幸。立馬新增兩三個或更多個吧！

篩選 objects

QuerySet 很大一部份的能力是篩選資料。這麼說吧，我們想要找到所有由使用者 ola 新增的 post。我們將會使用 `filter` 而不是在 `Post.objects.all()` 的 `all`。在括號中我們可以放入很多條件，而找到的資料必須符合我們的查詢條件。在這個例子中 `author` 等於 `user`。用 Django 寫起來就是 `author=user`。現在我們這段 code 就會像這樣：

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

或是也許我們想要看看所有 `title` 有 `title` 這個字的網誌呢？

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

你也可以獲得所有已發佈的網誌。我們會用 `published_date` 來篩選出來：

```
>>> Post.objects.filter(published_date__isnull=False)
[]
```

不幸的是，目前沒有半篇網誌是已經發佈的。我們可以改變這個情形！首先取得所有我們想要發佈的網誌：

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

然後用我們的 `publish` 方法發佈它！

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

現在試試得到一個已發佈網誌清單吧（按 3 次上方向鍵然後按下 Enter）：

```
>>> Post.objects.filter(published_date__isnull=False)
[<Post: Sample title>]
```

對 **objects** 排序

QuerySets 也允許你對物件清單排序。我們來試試用 `created_date` 這個欄位去排序它：

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

我們也可以在開頭加上 `-` 反排：

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

酷斃了！你現在已經對下一個階段蓄勢待發了！輸入下面指令關掉終端機：

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

Django admin

To add, edit and delete posts we've just modeled, we will use Django admin.

Let's open the `blog/admin.py` file and replace its content with this:

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

admin.site.register(Post)
```

As you can see, we import (include) the Post model defined in the previous chapter. To make our model visible on the admin page, we need to register the model with `admin.site.register(Post)`.

OK, time to look at our Post model. Remember to run `python manage.py runserver` in the console to run the web server. Go to the browser and type the address:

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

You will see a login page like this:

You should use the username and password you chose when you were creating a database (in the **Starting Django project** chapter). After logging in, you should see the Django admin dashboard.

Go to Posts and experiment a little bit with it. Add five or six blog posts. Don't worry about the content - you can simply copy-paste some text from this tutorial as your posts' content to save time :).

Make sure that at least two or three posts (but not all) have the publish date set. It will be helpful later.

If you want to know more about Django admin, you should check Django's documentation: https://docs.djangoproject.com/en/1.6/ref/contrib/admin/

It is probably a good moment to grab a coffee (or tea) and eat something sweet. You created your first Django model - you deserve a little treat!

Deploy!

Until now your website was only available on your computer, now you will learn how to deploy it! Deploying is the process of publishing your application on the Internet so people can finally go and see your app :).

As you learned, a website has to be located on a server. There are a lot of providers, but we will use the one with the simplest deployment process: Heroku. Heroku is free for small applications that don't have too many visitors, it'll definitely be enough for you now.

We will be following this tutorial: https://devcenter.heroku.com/articles/getting-started-with-django, but we pasted it here so it's easier for you.

The `requirements.txt` file

We need to create a `requirements.txt` file to tell Heroku what Python packages need to be installed on our server.

But first, Heroku needs us to install a few packages. Go to your console with `virtualenv` activated and type this:

```
(myvenv) $ pip install dj-database-url gunicorn whitenoise
```

After the installation is finished, go to the `djangogirls` directory and run this command:

```
(myvenv) $ pip freeze > requirements.txt
```

This will create a file called `requirements.txt` with a list of your installed packages (i.e. Python libraries that you are using, for example Django :)).

Open this file and add the following line at the bottom:

```
psycopg2==2.5.3
```

This line is needed for your application to work on Heroku.

Procfile

Another thing we need to create is a Procfile. This will let Heroku know which commands to run in order to start our website. Open up your code editor, create a file called `Procfile` in `djangogirls` directory and add this line:

```
web: gunicorn mysite.wsgi
```

This line means that we're going to be deploying a `web` application, and we'll do that by running the command `gunicorn mysite.wsgi` (`gunicorn` is a program that's like a more powerful version of Django's `runserver` command).

Then save it. Done!

The `runtime.txt` file

We need to tell Heroku which Python version we want to use. This is simply done by creating a `runtime.txt` and putting the following text inside:

```
python-3.4.1
```

mysite/local_settings.py

There is a difference between settings we are using locally (on our computer) and settings for our server. Heroku is using one database, and your computer is using a different database. That's why we need to create a seperate file for settings that will only be available for our local enviroment.

Go ahead and create `mysite/local_settings.py` file. It should contain your `DATABASE` setup from your `mysite/settings.py` file. Just like that:

```
import os
BASE_DIR = os.path.dirname(os.path.dirname(__file__))

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

DEBUG = True
```

Then just save it! :)

mysite/settings.py

Another thing we need to do is modify our website's `settings.py` file. Open `mysite/settings.py` in your editor and add the following lines at the end of the file:

```
import dj_database_url
DATABASES['default'] =  dj_database_url.config()

SECURE_PROXY_SSL_HEADER = ('HTTP_X_FORWARDED_PROTO', 'https')

ALLOWED_HOSTS = ['*']

STATIC_ROOT = 'staticfiles'

DEBUG = False
```

At the end of the `mysite/settings.py`, copy and paste this:

```
try:
    from .local_settings import *
except ImportError:
    pass
```

It'll import all of your local settings if the file exists.

Then save the file.

mysite/wsgi.py

Open the `mysite/wsgi.py` file and add these lines at the end:

```
from whitenoise.django import DjangoWhiteNoise
application = DjangoWhiteNoise(application)
```

All right!

Heroku account

You need to install your Heroku toolbelt which you can find here (you can skip the installation if you've already installed it during setup): https://toolbelt.heroku.com/

> When running the Heroku toolbelt installation program on Windows make sure to choose "Custom Installation" when being asked which components to install. In the list of components that shows up after that please additionally check the checkbox in front of "Git and SSH".
>
> On Windows you also must run the following command to add Git and SSH to your command prompt's `PATH`: `setx PATH "%PATH%;C:\Program Files\Git\bin"`. Restart the command prompt program afterwards to enable the change.

Please also create a free Heroku account here: https://id.heroku.com/signup/www-home-top

Then authenticate your Heroku account on your computer by running this command:

```
$ heroku login
```

In case you don't have an SSH key this command will automatically create one. SSH keys are required to push code to the Heroku.

Ignore some files

We don't want to send all of our project files to Heroku. Some of them, like `local_settings.py` or our database, need to stay only on our local computer. In order to tell Heroku which files should it ignore, we need to create a `.gitignore` file in our project main directory.

Create `.gitignore` file with the following content:

```
myvenv
__pycache__
staticfiles
local_settings.py
db.sqlite3
```

and save it. The dot on the beginning of the file name is important!

> **Note:** Remember to replace `myvenv` with the name you gave your `virtualenv`!

Deploy to Heroku!

It was a lot of configuration and installing, right? But you only need to do that once! Now you can deploy:

It's as simple as running this command, replacing `djangogirlsblog` with your own application name:

```
$ heroku create djangogirlsblog
```

One more thing: let's install heroku-push plugin by running this command:

```
$ heroku plugins:install https://github.com/ddollar/heroku-push
```

Now we can do a simple push to deploy our application:

```
$ heroku push --app djangogirlsblog
```

> : Remember to replace `djangogirlsblog` with the name of your application on Heroku.

Visit your application

You’ve deployed your code to Heroku, and specified the process types in a `Procfile` (we chose a `web` process type earlier). We can now tell Heroku to start this `web process`.

To do that, run the following command:

```
$ heroku ps:scale web=1 --app djangogirlsblog
```

This tells Heroku to run just one instance of our `web` process. Since our blog application is quite simple, we don't need too much power and so it's fine to run just one process. It's possible to ask Heroku to run more processes (by the way, Heroku calls these processes "Dynos" so don't be surprised if you see this term) but it will no longer be free.

We can now visit the app in our browser with `heroku open`.

```
$ heroku open --app djangogirlsblog
```

As you can see, there is an error. Heroku created a new database for us but it's empty. We also need to sync it:

```
$ heroku run python manage.py syncdb --app djangogirlsblog
```

You should now be able to see your website in a browser! Congrats :)!

Django urls

We're about to build our first webpage -- a homepage for your blog! But first, let's learn a little bit about Django urls.

What is a URL?

A URL is simply a web address, you can see a URL every time you visit any website - it is visible in your browser's address bar (yes! `127.0.0.1:8000` is a URL! And http://djangogirls.com is also a URL):

Every page on the Internet needs its own URL. This way your application knows what it should show to a user who opens a URL. In Django we use something called `URLconf` (URL configuration), which is a set of patterns that Django will try to match with the received URL to find the correct view.

How do URLs work in Django?

Let's open up the `mysite/urls.py` file and see what it looks like:

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

from django.contrib import admin
admin.autodiscover()

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

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

As you can see, Django already put something here for us.

Lines that start with `#` are comments - it means that those lines won't be run by Python. Pretty handy, right?

The admin URL, which you visited in previous chapter is already here:

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

It means that for every URL that starts with `admin/` Django will find a corresponding *view*. In this case we're including a lot of admin URLs so it isn't all packed into this small file -- it's more readable and cleaner.

Regex

Do you wonder how Django matches URLs to views? Well, this part is tricky. Django uses `regex` -- regular expressions. Regex has a lot (a lot!) of rules that form a search pattern. It is not so easy to understand so we won't worry about it today and you'll definitely get to know them in the future. Today we will only use the ones we need.

Your first Django url!

Time to create our first URL! We want http://127.0.0.1:8000/ to be a homepage of our blog and display a list of posts.

We also want to keep the `mysite/urls.py` file clean, so we will import urls from our `blog` application to the main `mysite/urls.py` file.

Go ahead, delete the commented lines (lines starting with `#`) and add a line that will import `blog.urls` into the main url (`''`).

Your `mysite/urls.py` file should now look like this:

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

from django.contrib import admin
admin.autodiscover()

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

Django will now redirect everything that comes into `http://127.0.0.1:8000/` to `blog.urls` and look for further instructions there.

blog.urls

Create a new `blog/urls.py` empty file. All right! Add these two first lines:

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

Here we're just importing Django's methods and all of our `views` from `blog` application (we don't have any yet, but we will get to that in a minute!)

After that, we can add our first URL pattern:

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

As you can see, we're now assigning a `view` called `post_list` to `^$` URL. But what does `^$` mean? It's a regex magic :) Let's break it down:

- `^` in regex means "the beginning"; from this sign we can start looking for our pattern
- `$` matches only "the end" of the string, which means that we will finish looking for our pattern here

If you put these two signs together, it looks like we're looking for an empty string! And that's correct, because in Django url resolvers, `http://127.0.0.1:8000/` is not a part of URL. This pattern will show Django that `views.post_list` is the right place to go if someone enters your website at the `http://127.0.0.1:8000/` address.

Everything all right? Open http://127.0.0.1:8000/ in your browser to see the result.

There is no "It works" anymore, huh? Don't worry, it's just an error page, nothing to be scared of! They're actually pretty useful:

You can read that there is **no attribute 'post_list'**. Is *post_list* reminding you of anything? This is how we called our view! This means that everything is in place, we just didn't create our *view* yet. No worries, we will get there.

> If you want to know more about Django URLconfs, look at the official documentation: https://docs.djangoproject.com/en/1.6/topics/http/urls/

Django views - time to create!

Time to get rid of the bug we created in the last chapter :)

A *view* is a place where we put the "logic" of our application. It will request information from the `model` you created before and pass it to a `template` that you will create in the next chapter. Views are just Python methods that are a little bit more complicated than the thing we did in the **Introduction to Python** chapter.

Views are placed in the `views.py` file. We will add our *views* to the `blog/views.py` file.

blog/views.py

OK, let's open up this file and see what's in there:

```
from django.shortcuts import render

# Create your views here.
```

Not too much stuff here yet. The simplest *view* can look like this.

```
def post_list(request):

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

As you can see, we created a method (`def`) called `post_list` that takes `request` and `return` a method `render` that will render (put together) our template `blog/post_list.html` .

Save the file, go to http://127.0.0.1:8000/ and see what we have got now.

Another error! Read what's going on now:

This one is easy: *TemplateDoesNotExist*. Let's fix this bug and create a template in the next chapter!

> Learn more about Django views by reading the official documentation:
> https://docs.djangoproject.com/en/1.6/topics/http/views/

Introduction to HTML

What's a template, you may ask?

A template is a file that we can re-use to present different information in a consistent format - for example, you could use a template to help you write a letter, because although each letter might contain a different message and be addressed to a different person, they will share the same format.

A Django template's format is described in a language called HTML (that's the HTML we mentioned in the first chapter **How the Internet works**).

What is HTML?

HTML is a simple code that is interpreted by your web browser - such as Chrome, Firefox or Safari - to display a webpage for user.

HTML stands for "HyperText Markup Language." **HyperText** means it's a type of text that supports hyperlinks between pages. **Markup** means we have taken a document and marked it up with code to tell something (in this case, a browser) how to interpret the page. HTML code is built with **tags**, each one starting with `<` and ending with `>`. These tags markup **elements**.

Your first template!

Creating a template means creating a template file. Everything is a file, right? You have probably noticed this already.

Templates are saved in `blog/templates/blog` directory. So first create a directory called `templates` inside your blog directory. Then create another directory called `blog` inside your templates directory:

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

(You might wonder why we need two directories both called `blog` - as you will discover later, this is simply a useful naming convention that makes life easier when things start to get more complicated.)

And now create a `post_list.html` file (just leave it blank for now) inside the `blog/templates/blog` directory.

See how your website looks now: http://127.0.0.1:8000/

> If you still have an error `TemplateDoesNotExists`, try to restart your server. Go into command line, stop the reserver by pressing Ctrl+C (Control and C buttons together) and start it again by running a `python manage.py runserver` command.

No error anymore! Congratulations :) However, your website isn't actually publishing anything except an empty page, because your template is empty too. We need to fix that.

Add the following to your template file:

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

So how does your website look now? Click to find out: http://127.0.0.1:8000/

It worked! Nice work there :)

- The most basic tag, `<html>`, is always the beginning of any webpage and `</html>` is always the end. As you can see, the whole content of the website goes between the beginning tag `<html>` and closing tag `</html>`
- `<p>` is a tag for paragraph elements; `</p>` closes each paragraph

Head & body

Each HTML page is also divided into two elements: **head** and **body**.

- **head** is an element that contains information about the document that is not displayed on the screen.
- **body** is an element that contains everything else that is displayed as part of the web page.

We use `<head>` to tell the browser about the configuration of the page, and `<body>` to tell it what's actually on the page.

For example, you can put a webpage title element inside the `<head>`, like this:

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

Save the file and refresh your page.

Notice how the browser has understood that "Ola's blog" is the title of your page? It has interpreted `<title>Ola's blog</title>` and placed the text in the title bar of your browser (it will also be used for bookmarks and so on).

Probably you have also noticed that each opening tag is matched by a *closing tag*, with a `/`, and that elements are *nested* (i.e. you can't close a particular tag until all the ones that were inside it have been closed too).

It's like putting things into boxes. You have one big box, `<html></html>`; inside it there is `<body></body>`, and that contains still smaller boxes: `<p></p>`.

You need to follow these rules of *closing* tags, and of *nesting* elements - if you don't, the browser may not be able to interpret them properly and your page will display incorrectly.

Customize your template

You can now have a little fun and try to customize your template! Here are a few useful tags for that:

- `<h1>A heading</h1>` - for your most important heading
- `<h2>A sub-heading</h2>` for a heading at the next level
- `<h3>A sub-sub-heading</h3>` ... and so on, up to `<h6>`
- `<em>text</em>` emphasizes your text
- `<strong>text</strong>` strongly emphasizes your text
- `<br />` goes to another line (you can't put anything inside br)
- `<a href="http://google.com/">link</a>` creates a link
- `<ul><li>first item</li><li>second item</li></ul>` makes a list, just like this one!
- `<div></div>` defines a section of the page

Here's an example of a full template:

```
<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
        </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
        </div>
    </body>
</html>
```

We've created three `div` sections here.

- The first `div` element contains the title of our blogpost - it's a heading and a link
- Another two `div` elements contain our blogposts with a published date, `h2` with a post title that is clickable and two `p` s (paragraph) of text, one for the date and one for our blogpost.

It gives us this effect:

Yaaay! But so far, our template only ever displays exactly **the same information** - whereas earlier we were talking about templates as allowing us to display **different** information in the **same format**.

What we want really want to do is display real posts added in our Django admin - and that's where we're going next.

One more thing

It'd be good to see if your website will be still working on Heroku, right? Let's try deploying again. Open up your console and type this:

```
heroku push --app djangogirlsblog
```

> : Remember to replace `djangogirlsblog` with the name of your application on Heroku.

And that should be it! Once Heroku is finished, you can go ahead and refresh your website in the browser. Changes should be visible!

Django Querysets

We have different pieces in place: the `Post` model is defined in `models.py`, we have `post_list` in `views.py` and the template added. But how will we actually make our posts appear in our HTML template? Because that is what we want: take some content (models saved in the database) and display it nicely in our template, right?

This is exactly what *views* are supposed to do: connect models and templates. In our `post_list` *view* we will need to take models we want to display and pass them to the template. So basically in a *view* we decide what (model) will be displayed in a template.

OK, so how will we achieve it?

We need to open our `blog/views.py`. So far `post_list` *view* looks like this:

```
from django.shortcuts import render

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

Remember when we talked about including code written in different files? Now it is the moment when we have to include the model we have written in `models.py`. We will add this line `from .models import Post` like this:

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

Dot after `from` means *current directory* or *current application*. Since `views.py` and `models.py` are in the same directory we can simply use `.` and the name of the file (without `.py`). Then we import the name of the model (`Post`).

But what's next? To take actual blog posts from `Post` model we need something called `QuerySet`.

QuerySet

You should already be familiar with how QuerySets work. We talked about it in Django ORM (QuerySets) chapter.

So now we are interested in a list of blog posts that are published and sorted by `published_date`, right? We already did that in QuerySets chapter!

```
Post.objects.filter(published_date__isnull=False).order_by('published_date')
```

Now we put this piece of code inside the `post_list` file, by adding it to the function `def post_list(request)`:

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

def post_list(request):
    posts = Post.objects.filter(published_date__isnull=False).order_by('published_date')
    return render(request, 'blog/post_list.html', {})
```

Please note that we create a *variable* for our QuerySet: `posts`. Treat this as the name of our QuerySet. From now on we can refer to it by this name.

The last missing part is passing the `posts` QuerySet to the template (we will cover how to display it in a next chapter).

In the `render` function we already have parameter with `request` (so everything we receive from the user via the Internet) and a template file `'blog/post_list.html'`. The last parameter, which looks like this: `{}` is a place in which we can add some things for the template to use. We need to give them names (we will stick to `'posts'` right now :)). It should look like this: `{'posts': posts}`. Please note that the part before `:` is wrapped with quotes `''`.

So finally our `blog/views.py` file should look like this:

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

def post_list(request):
    posts = Post.objects.filter(published_date__isnull=False).order_by('published_date')
    return render(request, 'blog/post_list.html', {'posts': posts})
```

That's it! Time to go back to our template and display this QuerySet!

If you want to read a little bit more about QuerySets in Django you should look here:
https://docs.djangoproject.com/en/1.6/ref/models/querysets/

Django templates

Time to display some data! Django gives us some helpful, built-in **template tags** for that.

What are template tags?

You see, in HTML, you can't really put Python code, because browsers don't understand it. They only know HTML. We know that HTML is rather static, while Python is much more dynamic.

Django template tags allow us to transfer Python-like things into HTML, so you can build dynamic websites faster and easier. Yikes!

Display post list template

In the previous chapter we gave our template a list of posts in the `posts` variable. Now we will display it in HTML.

To print a variable in Django template, we use double curly brackets with the variable's name inside, like this:

```
{{ posts }}
```

Try this in your `blog/templates/blog/post_list.html` template (replace everything between the second `<div></div>` tags with `{{ posts }}` line), save the file and refresh the page to see the results:

As you can see, all we've got is this:

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

This means that Django understands it as a list of objects. Remember from **Introduction to Python** how we can display lists? Yes, with the for loops! In a Django template, you do them this way:

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

Try this in your template.

It works! But we want them to be displayed like the static posts we created earlier in the **Introduction to HTML** chapter. You can mix HTML and template tags. Our `body` will look like this:

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

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

Everything you put between `{% for %}` and `{% endfor %}` will be repeated for each object in the list. Refresh your page:

Have you noticed that we used a slightly different notation this time `{{ post.title }}` or `{{ post.text }}`. We are accessing data in each of the fields defined in our `Post` model. Also the `|linebreaks` is piping the posts text through a filter to convert line-breaks into paragraphs.

One more thing

It'd be good to see if your website will be still working on Heroku, right? Let's try deploying again. Open up your console and type this:

```
heroku push --app djangogirlsblog
```

> : Remember to replace `djangogirlsblog` with the name of your application on Heroku.

Congrats! Now go ahead and try adding a new post in your Django admin (remember to add published_date!), then refresh your page to see if the post appears there.

Works like a charm? We're proud! Treat yourself something sweet, you have earned it :)

CSS - make it pretty!

Our blog still looks pretty ugly, right? Time to make it nice! We will use CSS for that.

What is CSS?

Cascading Style Sheets (CSS) is a language used for describing the look and formatting of a website written in markup language (like HTML). Treat it as make-up for our webpage ;).

But we don't want to start from scratch again, right? We will, once more, use something that has already been done by programmers and released on the Internet for free. You know, reinventing the wheel is no fun.

Let's use Bootstrap!

Bootstrap is one of the most popular HTML and CSS frameworks for developing beautiful websites:
http://getbootstrap.com/

It was written by programmers who worked for Twitter and is now developed by volunteers from all over the world.

Install Boostrap

To install Bootstrap, you need to add this to your `<head>` in your `.html` file (`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">
<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
```

This doesn't add any files to your project. It just points to files that exist on the internet. Just go ahead, open your website and refresh the page. Here it is!

Looking nicer already!

Static files in Django

Another thing you will learn about today is called **static files**. Static files are all your CSS and images -- files that are not dynamic, so their content doesn't depend on request context and will be the same for every user.

CSS is a static file, so in order to customize CSS, we need to first configure static files in Django. You'll only need to do it once. Let's start:

Configure static files in Django

First, we need to create a directory to store our static files in. Go ahead and create a directory called `static` inside your `djangogirls` directory.

```
djangogirls
├─── static
└─── manage.py
```

Open up the `mysite/settings.py` file, scroll to the bottom of it and add the following lines:

```
STATICFILES_DIRS = (
    os.path.join(BASE_DIR, "static"),
)
```

This way Django will know where to find your static files.

Your first CSS file!

Let's create a CSS file now, to add your own style to your web-page. Create a new directory called `css` inside your `static` directory. Then create a new file called `blog.css` inside this `css` directory. Ready?

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

Time to write some CSS! Open up the `static/css/blog.css` file in your code editor.

We won't be going too deep into customizing and learning about CSS here, because it's pretty easy and you can learn it on your own after this workshop. We really recommend doing this Codeacademy HTML & CSS course to learn everything you need to know about making your websites more pretty with CSS.

But let's do at least a little. Maybe we could change the color of our header? To understand colors, computers use special codes. They start with `#` and are followed by 6 letters (A-F) and numbers (0-9). You can find color codes for example here: http://www.colorpicker.com/. You may also use predefined colors, such as `red` and `green`.

In your `static/css/blog.css` file you should add following code:

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

`h1 a` is a CSS Selector. Any `a` element inside of an `h1` element (i.e. when we have in code something like: `<h1><a href="">link</a></h1>`) is the tag we're applying our styles to, and we're telling it to change its color to `#FCA205`, which is orange. Of course, you can put your own color here!

In a CSS file we determine styles for elements in the HTML file. The elements are identified by the element name (i.e. `a`, `h1`, `body`), the attribute `class` or the attribute `id`. Class and id are names you give the element by yourself. Classes define groups of elements, and ids point to specific elements. For example, the following tag may be identified by CSS using the tag name `a`, the class `external_link`, or the id `link_to_wiki_page`:

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

Read about CSS Selectors in w3schools.

Then, we need to also tell our HTML template that we added some CSS. Open the `blog/templates/blog/post_list.html` file and add this line at the very beginning of it:

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

We're just loading static files here :). Then, between the `<head>` and `</head>`, after the links to the Bootstrap CSS files (the browser reads the files in the order they're given, so code in our file may override code in Bootstrap files), add this line:

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

We just told our template where our CSS file is located.

Your file should now look like this:

```
{% 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">
        <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
        <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 }}</p>
            </div>
        {% endfor %}
    </body>
</html>
```

OK, save the file and refresh the site!

Nice work! Maybe we would also like to give our website a little air and increase the margin on the left side? Let's try this!

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

Add this to your CSS, save the file and see how it works!

Maybe we can customize the font in our header? Paste this into your `<head>` in `blog/templates/blog/post_list.html` file:

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

This line will import a font called *Lobster* from Google Fonts (https://www.google.com/fonts).

Now add the line `font-family: 'Lobster';` in the CSS file `static/css/blog.css` inside the `h1 a` declaration block (the code between the braces `{` and `}`) and refresh the page:

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

Great!

As mentioned above, CSS has a concept of classes, which basically allows you to name a part of the HTML code and apply styles only to this part, not affecting others. It's super helpful if you have two divs, but they're doing something very different (like your header and your post), so you don't want them to look the same.

Go ahead and name some parts of the HTML code. Add a class called `page-header` to your `div` that contains your header, like this:

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

And now add a class `post` to your `div` containing a blog post.

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

We will now add declaration blocks to different selectors. Selectors starting with `.` relate to classes. There are many great tutorials and explanations about CSS on the Web to help you understand the following code. For now, just copy and paste it into your `mysite/static/css/blog.css` file:

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

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

.content {
    margin-left: 40px;
}

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

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

.save {
    float: right;
}

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

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

.post {
    margin-bottom: 70px;
}

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

Then surround the HTML code which displays the posts with declarations of classes. Replace this:

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

in the `blog/templates/blog/post_list.html` with this:

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

Save those files and refresh your website.

Wohoo! Looks awesome, right? The code we just pasted is not really so hard to understand and you should be able to understand most of it just by reading it.

Don't be afraid to tinker with this CSS a little bit and try to change some things. If you break something, don't worry, you can always undo it!

Anyway, we really recommend taking this free online Codeacademy HTML & CSS course as some post-workshop homework to learn everything you need to know about making your websites prettier with CSS.

Ready for the next chapter?! :)

Template extending

Another nice thing Django has for you is **template extending**. What does this mean? It means that you can use the same parts of your HTML for different pages of your website.

This way you don't have to repeat yourself in every file, when you want to use the same information/layout and, if you want to change something, you don't have to do it in every template, just once!

Create base template

A base template is the most basic template that you extend on every page of your website.

Let's create a `base.html` file in `blog/templates/blog/`:

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

Then open it up and copy everything from `post_list.html` to `base.html` file, like this:

```
{% 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">
        <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
        <link href='http://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">
            <div class="row">
                <div class="col-md-8">
                {% for post in posts %}
                    <div class="post">
                        <p>published: {{ post.published_date }}</p>
                        <h1><a href="">{{ post.title }}</a></h1>
                        <p>{{ post.text|linebreaks }}</p>
                    </div>
                {% endfor %}
                </div>
            </div>
        </div>
    </body>
</html>
```

Then in `base.html`, replace your whole `<body>` (everything between `<body>` and `</body>`) with this:

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

We basically replaced everything between `{% for post in posts %}{% endfor %}` with:

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

What does it mean? You just created a `block`, which is a template tag that allows you to insert HTML in this block in other templates that extend `base.html`. We will show you how to do this in a moment.

Now save it, and open your `blog/templates/blog/post_list.html` again. Delete everything else other than what's inside the body and then also delete `<div class="page-header"></div>`, so the file will look like this:

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

And now add this line to the beginning of the file:

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

It means that we're now extending `base.html` template in `post_list.html`. Only one thing left: put everything (except the line we just added) between `{% block content %}` and `{% endblock content %}`. Like this:

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

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

That's it! Check if your website is still working properly :)

> If you have an error `TemplateDoesNotExists` that says that there is no `blog/base.html` file and you have `runserver` running in the console, try to stop it (by pressing Ctrl+C - Control and C buttons together) and restart it by running a `python manage.py runserver` command.

Extend your application

We've already completed all the different steps necessary for the creation of our website: we know how to write a model, url, view and template. We also know how to make our website pretty.

Time to practice!

The first thing we need in our blog is, obviously, a page to display one post, right?

We already have a `Post` model, so we don't need to add anything to `models.py`.

Create a link in the template

We will start with adding a link inside `blog/templates/blog/post_list.html` file. So far it should look like:

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

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

We want to have a link to a post detail page on the post's title. Let's change `<h1><a href="">{{ post.title }}</a></h1>` into a link:

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

Time to explain the mysterious `{% url 'blog.views.post_detail' pk=post.pk %}`. As you suspect `{% %}` notation means that we are using Django template tags. This time we will use one that will create a URL for us!

`blog.views.post_detail` is a path to a `post_detail` *view* we want to create. Please note: `blog` is the name of our application (the directory `blog`), `views` is from the name of the `views.py` file and the last bit: `post_detail` is the name of the *view*.

Now when we go to:

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

we will have an error (as suspected, since we don't have a URL or a *view* for `post_detail`). It will look like this:

Let's create a URL in `urls.py` for our `post_detail` *view*!

URL: http://127.0.0.1:8000/post/1/

We want to create a URL to point Django to a *view* called `post_detail`, that will show an entire blog post. Add the line `url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail),` to the `blog/urls.py` file. It should look like this:

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

urlpatterns = patterns('',
    url(r'^$', views.post_list),
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail),
)
```

That one looks scary, but no worries - we will explain it for you:

- it's starts with `^` again -- "the beginning"
- `post/` only means that after the beginning, the URL should contain the word **post** and **/**. So far so good.
- `(?P<pk>[0-9]+)` - this part is trickier. It means that Django will take everything that you'll place here and transfer it to a view as a variable called `pk`. `[0-9]` also tells us that it can only be a number, not a letter (so everything between 0 and 9). `+` means that there needs to be one or more digits there. So something like `http://127.0.0.1:8000/post//` is not

valid, but `http://127.0.0.1:8000/post/1234567890/` is perfectly ok!
- `/` - then we need ***/*** again
- `$` - "the end"!

That means if you enter `http://127.0.0.1:8000/post/5/` into your browser, Django will understand that you are looking for a *view* called `post_detail` and transfer the information that `pk` equals `5` to that *view*.

`pk` is shortcut for `primary key`. This name is often used in many Django projects. But you can name your variable as you like (remember: lowercase and `_` instead of whitespaces!). For example instead of `(?P<pk>[0-9]+)` we could have variable `post_id`, so this bit would look like: `(?P<post_id>[0-9]+)`.

Ok! Let's refresh the page:

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

Boom! Yet another error! As expected!

Do you remember what the next step is? Of course: adding a view!

post_detail view

This time our *view* is given an extra parameter `pk`. Our *view* needs to catch it, right? So we will define our function as `def post_detail(request, pk):`. Note that we need to use exactly the same name as the one we specified in urls (`pk`). Omitting this variable is incorrect and will result in an error!

Now, we want to get one and only one blog post. To do this we can use querysets like this:

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

But this code has a problem. If there is no `Post` with given `primary key` (`pk`) we will have a super ugly error!

We don't want that! But, of course, Django comes with something that will handle that for us: `get_object_or_404`. In case there is no `Post` with the given `pk` it will display much nicer page (called `Page Not Found 404` page).

The good news is that you can actually create your own `Page not found` page and make it as pretty as you want. But it's not super important right now, so we will skip it.

Ok, time to add a *view* to our `views.py` file!

We should open `blog/views.py` and add the following code:

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

Near other `from` lines. And at the end of the file we will add our *view*:

```
def post_detail(request, pk):
    post = get_object_or_404(Post, pk=pk)
    return render(request, 'blog/post_detail.html', {'post': post})
```

Yes. It is time to refresh the page:

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

It worked! But what happens when you click a link in blog post title?

Oh no! Another error! But we already know how to deal with it, right? We need to add a template!

We will create a file in `blog/templates/blog` called `post_detail.html`.

It will look like this:

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

{% block content %}
    <div class="date">
        {% if post.published_date %}
            {{ post.published_date }}
        {% endif %}
    </div>
    <h1>{{ post.title }}</h1>
    <p>{{ post.text }}</p>
{% endblock %}
```

Once again we are extending `base.html`. In the `content` block we want to display a post's published_date (if it exists), title and text. But we should discuss some important things, right?

`{% if ... %} ... {% endif %}` is a template tag we can use when we want to check something (remember `if ... else ..` from **Introduction to Python** chapter?). In this scenario we want to check if a post's `published_date` is not empty.

Ok, we can refresh our page and see if `Page not found` is gone now.

Yay! It works!

One more thing: deploy time!

It'd be good to see if your website will be still working on Heroku, right? Let's try deploying again. Open up your console and type this:

```
heroku push --app djangogirlsblog
```

> : Remember to replace `djangogirlsblog` with the name of your application on Heroku.

And that should be it! Congrats :)

Django Forms

The final thing we want to do on our website is create a nice way to add and edit blog posts. Django's `admin` is cool, but it is rather hard to customize and make pretty. With `forms` we will have absolute power over our interface - we can do almost anything we can imagine!

The nice thing about Django forms is that we can either define one from scratch or create a `ModelForm` which will save the result of the form to the model.

This is exactly what we want to do: we will create a form for our `Post` model.

Like every important part of Django, forms have their own file: `forms.py`.

We need to create a file with this name in the `blog` directory.

```
blog
   └── forms.py
```

Ok, let's open it and type the following code:

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

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

We need to import Django forms first (`from django import forms`) and, obviously, our `Post` model (`from .models import Post`).

`PostForm` , as you probably suspect is the name of our form. We need to tell Django, that this form is a `ModelForm` (so Django will do some magic for us) - `forms.ModelForm` is responsible for that.

Next, we have `class Meta` , where we tell Django which model should be used to create this form (`model = Post`).

Finally, we can say which field(s) should end up in our form. In this scenario we want only `title` and `text` to be exposed - `author` should be the person who is currently logged in (you!) and `created_date` should be automatically set when we create a post (i.e in the code), right?

And that's it! All we need to do now is use the form in a *view* and display it in a template.

So once again we will create: a link to the page, a URL, a view and a template.

Link to a page with the form

It's time to open `blog/templates/blog/base.html`. We will add a link in `div` named `page-header`:

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

Note that we want to call our new view `post_new`.

After adding the line, your html file should now look like this:

```
{% 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">
        <script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>
        <link href='http://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 'blog.views.post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
            <h1><a href="/">Django Girls Blog</a></h1>
        </div>
        <div class="content">
            <div class="row">
                <div class="col-md-8">
                    {% block content %}
                    {% endblock %}
                </div>
            </div>
        </div>
    </body>
</html>
```

After saving and refreshing the page `http://127.0.0.1:8000` you will obviously see a familiar `NoReverseMatch` error, right?

URL

We open `blog/urls.py` and add a line:

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

And the final code will look like this:

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

urlpatterns = patterns('',
    url(r'^$', views.post_list),
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail),
    url(r'^post/new/$', views.post_new, name='post_new'),
)
```

After refreshing the site, we see an `AttributeError`, since we don't have `post_new` view implemented. Let's add it right now.

post_new view

Time to open the `blog/views.py` file and add the following lines with the rest of the `from` rows:

```
from .forms import PostForm
```

and our *view*:

```
def post_new(request):
    form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

To create a new `Post` form, we need to call `PostForm()` and pass it to the template. We will go back to this *view*, but for now, let's create quickly a template for the form.

Template

We need to create a file `post_edit.html` in the `blog/templates/blog` directory. To make a form work we need several things:

- we have to display the form. We can do that for example with a simple `{{ form.as_p }}`.
- the line above needs to be wrapped with an HTML form tag: <`form method="POST">...</form>`
- we need a `Save` button. We do that with an HTML button: `<button type="submit">Save</button>`
- and finally just after the opening `<form ...>` tag we need to add `{% csrf_token %}`. This is very important, since it makes your forms secure! Django will complain if you forget about this bit if you try to save the form:

Ok, so let's see how the HTML in `post_edit.html` should look:

```
{% 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 %}
```

Time to refresh! Yay! Your form is displayed!

But, wait a minute! When you type something in `title` and `text` fields and try to save it - what will happen?

Nothing! We are once again on the same page and our text is gone... and no new post is added. So what went wrong?

The answer is: nothing. We need to do a little bit more work in our *view*.

Saving the form

Open `blog/views.py` once again. Currently all we have in `post_new` view is:

```
def post_new(request):
    form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

When we submit the form, we are back in the same view, but this time we have some more data in `request`, more specifically in `request.POST`. Remember that in HTML file our `<form>` definition had variable `method="POST"`? All the fields from the form are now in `request.POST`. You should not rename `POST` to anything else (the only other valid value for `method` is `GET`, but we have no time to explain what the difference is).

So in our *view* we have two separate situations to handle. First one: when we access the page for the first time and we want a blank form. Second one: when we go back to the *view* with all form's data we just typed. So we need to add a condition (we will use `if` for that).

```
if request.method == "POST":
    [...]
else:
    form = PostForm()
```

It's time to fill in the dots `[...]`. If `method` is `POST` then we want to construct the `PostForm` with data from the form, right? We will do that with:

```
form = PostForm(request.POST)
```

Easy! Next thing is to check if the form is correct (all required fields are set and no incorrect values will be saved). We do that with `form.is_valid()`.

We check if the form is valid and if so, we can save it!

```
if form.is_valid():
    post = form.save(commit=False)
    post.author = request.user
    post.save()
```

Basically, we have two things here: we save the form with `form.save` and we add an author (since there was no `author` field in the `PostForm` and this field is required!). `commit=False` means that we don't want to save `Post` model yet - we want to add author first. Most of the time you will use `form.save()`, without `commit=False`, but in this case, we need to do that. `post.save()` will preserve changes (adding author) and a new blog post is created!

Finally, it would be awesome if we can immediatelly go to `post_detail` page for newly created blog post, right? To do that we need more imports:

```
from django.core.urlresolvers import reverse
from django.shortcuts import redirect
```

Add them at the very beginning of your file. And now we can say: go to `post_detail` page for a newly created post.

```
return redirect('blog.views.post_detail', pk=post.pk)
```

`blog.views.post_detail` is the name of the view we want to go to. Remember that this *view* requires a `pk` variable? To pass it to the views we use `pk=post.pk`, where `post` is the newly created blog post!

Ok, we talked a lot, but we probably want to see what the whole *view* looks like now, right?

```
def post_new(request):
    if request.method == "POST":
        form = PostForm(request.POST)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.save()
            return redirect('blog.views.post_detail', pk=post.pk)
    else:
        form = PostForm()
    return render(request, 'blog/post_edit.html', {'form': form})
```

Let's see if it works. Go to the page `http://127.0.0.1:8000/post/new/`, add a `title` and `text`, save it... and voilà! The new blog post is added and we are redirected to `post_detail` page!

You problably have noticed that we are not setting publish date at all. We will introduce a *publish button* in **Django Girls Tutorial: Extensions**.

That is awesome!

Form validation

Now, we will show you how cool Django forms are. Blog post needs to have `title` and `text` fields. In our `Post` model we did not say (as opposed to `published_date`) that these fields are not required, so Django, by default, expects them to be set.

Try to save the form without `title` and `text`. Guess, what will happen!

Django is taking care of validating that all the fields in our form are correct. Isn't it awesome?

> As we have recently used the Django admin interface the system currently thinks we are logged in. There are a few situations that could lead to us being logged out (closing the browser, restarting the DB etc.). If you find that you are getting errors creating a post referring to a lack of a logged in user. Head to the admin page `http://127.0.0.1:8000/admin` and login again. This will fix the issue temporarily. There is a permanent fix awaiting you in the **Homework: add security to your website!** chapter after the main tutorial.

Edit form

Now we know how to add a new form. But what if we want to edit an existing one? It is very similar to what we just did. Let's create some important things quickly (if you don't understand something - you should ask your coach or look at the previous chapters, since we covered all the steps already).

Open `blog/post_detail.html` and add this line:

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

so that the template will look like:

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

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

In `blog/urls.py` we add this line:

```
url(r'^post/(?P<pk>[0-9]+)/edit/$', views.post_edit, name='post_edit'),
```

We will reuse the template `blog/templates/blog/post_edit.html`, so the last missing thing is a *view*.

Let's open a `blog/views.py` and add at the very end of the file:

```
def post_edit(request, pk):
    post = get_object_or_404(Post, pk=pk)
    if request.method == "POST":
        form = PostForm(request.POST, instance=post)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.save()
            return redirect('blog.views.post_detail', pk=post.pk)
    else:
        form = PostForm(instance=post)
    return render(request, 'blog/post_edit.html', {'form': form})
```

This looks almost exactly the same as our `post_new` view, right? But not entirely. First thing: we pass an extra `pk` parameter from urls. Next - we get the `Post` model we want to edit with `get_object_or_404(Post, pk=pk)` and then, when we create a form we pass this post as an `instance` both when we save the form:

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

and when we just opened a form with this post to edit:

```
form = PostForm(instance=post)
```

Ok, let's test if it works! Let's go to `post_detail` page. There should be an edit button in the top-right corner::

When you click it you will see the form with our blog post:

Feel free to change the title or the text and save changes!

Congratulations! Your application is more and more complete!

If you need more information about Django forms you should read the documentation:
https://docs.djangoproject.com/en/1.6/topics/forms/

One more thing: deploy time!

It'd be good to see if your website will be still working on Heroku, right? Let's try deploying again. Open up your console and type this:

```
heroku push --app djangogirlsblog
```

> : Remember to replace `djangogirlsblog` with the name of your application on Heroku.

And that should be it! Congrats :)

Domain

Heroku gave you a domain, but it's long, hard to remember, and ugly. It'd be awesome to have a domain name that is short and easy to remember, right?

In this chapter we will teach you how to buy a domain and direct it to Heroku!

Where to register a domain?

A typical domain costs around $15 a year. There are cheaper and more expensive options, depending on the provider. There are a lot of companies that you can buy a domain from: a simple google search give hundreds of options.

Our favourite one is I want my name. They advertise as "painless domain management" and it really is painless.

How to register domain in IWantMyName?

Go to iwantmyname and type a domain you want to have in the search box.

You should now see a list of all available domains with the term you put in the search box. As you can see, a smiley face indicate that the domain is available for you to buy, and a sad face that it is already taken.

We've decided to buy `djangogirls.in`:

Go to checkout. You should now sign up for iwantmyname if you don't have an account yet. After that, provide your credit card info and buy a domain!

After that, click `Domains` in the menu and choose your newly purchased domain. Then locate and click on the `manage DNS records` link:

Now you need to locate this form:

And fill it in with the following details:

- Hostname: www
- Type: CNAME
- Value: your domain from Heroku (for example djangogirls.herokuapp.com)
- TTL: 3600

Click the Add button and Save changes at the bottom.

It can take up to a couple of hours for your domain to start working, so be patient!

Configure domain in Heroku

You also need to tell Heroku that you want to use your custom domain.

Go to the Heroku Dashboard, login to your Heroku account and choose your app. Then go into app Settings and add your domain in the `Domains` section and save your changes.

That's it!

What's next?

Congratulate yourself! **You're totally awesome**. We're proud! <3

What to do now?

Take a break and relax. You have just done something really huge.

After that make sure to:

- Follow Django Girls on Facebook or Twitter to stay up to date
- Join our community: ask questions, share your progress, get to know us better :)

Can you recommend any further resources?

Yes! First, go ahead and try our another book, called Django Girls Tutorial: Extensions.

Later on, you can try recources listed below. They're all very recommended!

- 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

Table of Contents