

Tutorial

djangogirls

Sommario

Introduzione	1.1
Installazione	1.2
Come funziona Internet	1.3
Introduzione al command line	1.4
Installazione di Python	1.5
Code Editor	1.6
Introduzione a Python	1.7
Che cos'è Django?	1.8
Installazione di Django	1.9
Il tuo primo progetto Django!	1.10
I modelli di Django	1.11
L'admin di Django	1.12
Deploy!	1.13
Django URL	1.14
Le views di Django - è arrivata l'ora di creare!	1.15
Introduzione all'HTML	1.16
ORM di Django (Querysets)	1.17
I dati dinamici in templates	1.18
I templates di Django	1.19
CSS - dagli un bell'aspetto	1.20
Estendere il template	1.21
Estendi la tua applicazione	1.22
Form Django	1.23
Quali sono le prospettive?	1.24

Il tutorial di Django Girls

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

Questo lavoro è sotto la licenza internazionale di Creative Commons Attribution-ShareAlike 4.0. Per vedere una copia di questa licenza, visita <https://creativecommons.org/licenses/by-sa/4.0/>

Introduzione

Hai mai sentito che il mondo sta diventando sempre più tecnologico e sei in qualche modo rimasto indietro? ti sei mai chiesta come creare un sito web ma non hai mai avuto abbastanza motivazione per iniziare? Hai mai pensato che il mondo del software è troppo complicato per te persino per provare a fare qualcosa per conto tuo?

Beh, abbiamo buone notizie per te! La programmazione non è così complicata come sembra e vogliamo dimostrarti quanto può essere divertente.

Il tutorial non ti trasformerà magicamente in un programmatore. Se vuoi diventare bravo/a, ci vorranno mesi o addirittura anni di apprendimento e pratica. Ma ti vogliamo dimostrare che la programmazione o creare siti web non è complicato come sembra. Proveremo a spiegarti diversi argomenti come meglio possiamo, in modo che non ti senta più intimidito/a dalla tecnologia.

Speriamo di essere in grado di farti amare la tecnologia come lo facciamo noi!

Cosa imparerai durante questo tutorial?

Una volta che avrai terminato il tutorial, avrai a disposizione una semplice applicazione web: il tuo blog personale. Ti mostreremo come metterlo in linea, in modo che gli altri possano vedere il tuo lavoro!

Assomiglierà (più o meno) a questo:

The screenshot shows a web browser window titled "Django Girls Blog" at "127.0.0.1:8000". The page has a bright orange header with the "Django Girls" logo and a navigation bar with a pencil icon and a plus sign. Below the header, there are three article cards:

- 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...

Se lavori da solo/a al tutorial e non hai un'insegnante che ti possa aiutare nel caso avessi qualche problema, abbiamo una chat per te: [gitter](#) [join chat](#). Abbiamo chiesto ai nostri insegnanti e partecipanti precedenti di esserci di volta in volta ed aiutare gli altri con il tutorial! Non aver paura di fare domande!

OK, [cominciamo dall'inizio...](#)

Informazioni e contribuzioni

Questo tutorial è mantenuto da [DjangoGirls](#). Se trovi errori o se vuoi aggiornare questo tutorial, [segui le linee guida per i collaboratori](#).

Vorresti aiutarci a tradurre il tutorial in un'altra lingua?

Attualmente, le traduzioni vengono mantenute sulla piattaforma [crowdin.com](#) in:

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

Se la tua lingua non è elencata su crowding, per favore [apri una nuova istanza](#) con la tua lingua in modo che possiamo aggiungerla.

Se stai facendo il tutorial a casa

Se stai facendo il tutorial a casa e non durante uno degli [eventi Django Girls](#), puoi saltare questo capitolo e andare direttamente a [Come funziona Internet?](#).

Copriamo questi argomenti nel tutorial completo, e questa è una pagina aggiuntiva contenente tutte le istruzioni di installazione. L'evento Django Girls include una "serata dedicata all'installazione" in cui installiamo tutto, in modo da non preoccuparcene durante il workshop, quindi questo capitolo è utile per chi partecipa ad uno dei nostri workshop.

Se lo trovi utile puoi seguire anche questo capitolo. Ma se vuoi iniziare a imparare prima di installare le cose sul tuo computer, salta questo capitolo. Ti spiegheremo l'installazione più avanti.

Buona fortuna!

Installazione

Nel workshop costruirai un blog, e ci sono alcuni task dedicati all'impostazione che sarebbe bello completare in anticipo, in modo che tu sia pronta a scrivere codice in giornata.

Installare Python

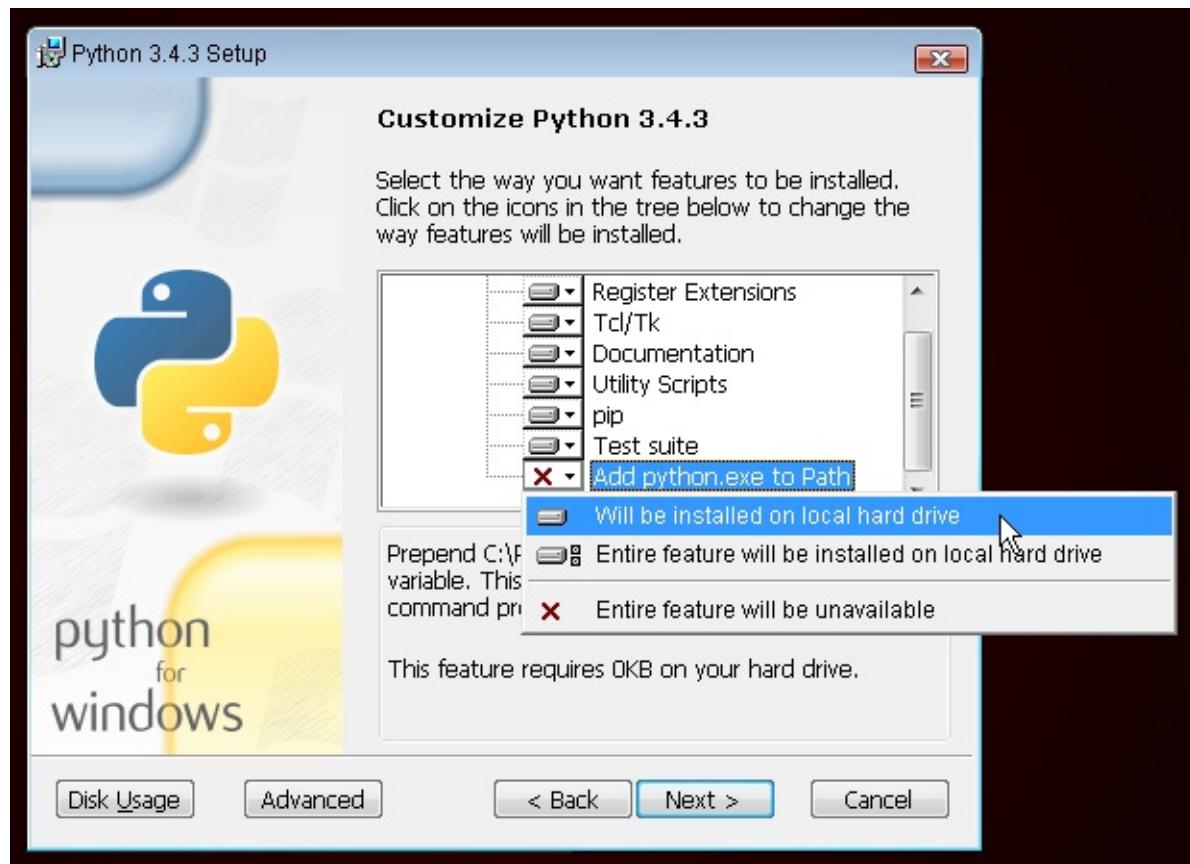
Questa sezione si basa su un tutorial fatto da Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>)

Django è scritto in Python. Abbiamo bisogno di Python per fare qualsiasi cosa in Django. Iniziamo con l'installazione! Vogliamo che sul tuo pc sia installato Python 3.4 quindi se hai una versione precedente, dovrà aggiornarlo.

Windows

Puoi scaricare Python per Windows dal sito web <https://www.python.org/downloads/release/python-343/>. Dopo aver scaricato il file ***.msi**, lo dovresti eseguire (cliccaci sopra due volte) e segui le istruzioni. È importante ricordare il percorso (la directory) dove ha installato Python. Più tardi sarà necessario!

Una cosa a cui fare attenzione: sulla seconda schermata dell'installazione guidata, contrassegnata "Customize", assicurati di scorrere verso il basso e di scegliere l'opzione "Add python.exe to the Path", come illustrato qui:



Linux

È molto probabile che tu abbia Python già installato di default. Per controllare se ce l'hai già installato (e quale versione è), apri una console e digita il seguente comando:

```
$ python3 --version  
Python 3.4.3
```

Se non hai Python installato o se vuoi una versione diversa, puoi installarla come segue:

Debian o Ubuntu

Digita questo comando nella tua console:

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

Fedora (fino a 21)

Usa questo comando nella tua console:

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

Fedora (22+)

Usa questo comando nella tua console:

```
$ sudo dnf install python3.4
```

openSUSE

Usa questo comando nella tua console:

```
$ sudo zypper install python3
```

OS X

Devi andare sul sito <https://www.python.org/downloads/release/python-343/> e scarica il programma d'installazione di Python:

- Scarica il file *Mac OS X 64-bit/32-bit installer*
- Fai doppio click su *python-3.4.3-macosx10.6.pkg* per eseguire il programma d'installazione.

Verifica che l'installazione si sia conclusa correttamente aprendo l'applicazione *Terminal* ed eseguendo il comando

```
python3 :
```

```
$ python3 --version
Python 3.4.3
```

Se hai dubbi o se qualcosa è andato storto e non hai idea di cosa fare dopo - chiedi al tuo insegnante! A volte le cose non vanno come dovrebbero ed è meglio chiedere aiuto a qualcuno con più esperienza.

Preparare virtualenv e installare Django

Una parte di questo capitolo si basa sui tutorial delle Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Una parte di questo capitolo di base sul [django-marcador tutorial](#) sotto licenza Creative Commons Attribution-ShareAlike 4.0 International License. Il tutorial di django-marcador è protetto da copyright di Markus Zapke-Gründemann et al.

Ambiente virtuale

Prima di installare Django, ti vogliamo far installare uno strumento estremamente utile per tenere in ordine l'ambiente in cui programmerai sul tuo computer. Potresti saltare questo passaggio, ma è caldamente consigliato soffermarsi. Se inizi con la migliore configurazione possibile, ti risparmierai un sacco di problemi per il futuro!

Per cui, creiamo ora un **ambiente virtuale** (chiamato anche un *virtualenv*). Virtualenv isolerà la tua configurazione di Python/Django in base ai diversi progetti. Questo significa che qualunque modifica farai su un sito non avrà alcun effetto su tutti gli altri che stai sviluppando. Chiaro ora?

Tutto quello che devi fare è trovare una cartella in cui vuoi creare il `virtualenv`; la tua home directory, ad esempio. Su Windows potrebbe essere `C:\Users\Name` (dove `Name` è il nome del tuo login).

Per questo tutorial useremo una nuova directory `djangogirls` dalla tua home directory:

```
mkdir djangogirls
cd djangogirls
```

Ora creeremo un virtualenv dal nome `myvenv`. Questo è il formato del comando generale:

```
python3 -m venv myvenv
```

Windows

Per creare un nuovo `virtualenv` è necessario aprire la console (ti abbiamo spiegato come fare nei capitoli precedenti, ricordi?) ed esegui `C:\Python34\python -m venv myvenv`. Il risultato somiglierà a questo:

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

dove `C:\Python34\python` è la directory in cui precedentemente hai installato Python e `myvenv` è il nome del tuo `virtualenv`. Puoi utilizzare qualsiasi altro nome, ma attieniti a utilizzare le minuscole, a non usare spazi, accenti o caratteri speciali. È meglio usare un nome breve dal momento che dovrà digitarlo spesso!

Linux e OS X

Creare un `virtualenv` su Linux e OS X è semplice, basta digitare: `python3 -m venv myvenv`. Il risultato sarà simile a questo:

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

`myvenv` è il nome del tuo `virtualenv`. Puoi usare qualsiasi altro nome, ma utilizza solo minuscole e niente spazi. Sarebbe meglio se il nome fosse corto perché lo dovrà digitare molte volte!

NOTA: Avviare il `virtualenv` su Ubuntu 14.04 in questa maniera al momento darà il seguente errore:

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

Per aggirare il problema utilizza, invece del precedente, il comando `virtualenv`.

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

Lavorare con `virtualenv`

Il comando sopra specificato, creerà una cartella dal nome `myvenv` (o col nome che hai scelto) che contiene il tuo virtual environment (ovvero un mucchio di files e cartelle).

Windows

Avvia il tuo `virtualenv` digitando:

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

Linux e OS X

Avvia il tuo `virtualenv` digitando:

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

Ricordati di sostituire `myvenv` con il nome `virtualenv` che hai scelto!

Nota: a volte il comando `source` potrebbe non essere disponibile. In quel caso prova ad usare questo:

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

Saprai con certezza che hai avviato `virtualenv` quando vedrai che il prompt dei comandi nella console si presenta così:

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

oppure:

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

Fai caso al prefisso `(myvenv)`!

Quando si lavora all'interno di un ambiente virtuale, `python` farà automaticamente riferimento alla versione corretta da utilizzare. Per cui puoi digitare `python` invece `python3`.

OK, abbiamo tutte le dipendenze importanti pronte. Finalmente possiamo installare Django!

Installare Django

Ora che hai iniziato ad utilizzare il tuo `virtualenv`, puoi installare Django usando `pip`. Nella console, esegui `pip install django==1.8` (nota che usiamo un doppio simbolo di uguale: `==`).

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

Su Windows

Se ottieni un errore quando chiami `pip` sulla piattaforma Windows controlla se il pathname del tuo progetto contiene spazi, accenti o caratteri speciali (i.e. `C:\Users\User Name\djangogirls`). Se è così ti conviene spostarlo in un altro path senza spazi, accenti o caratteri speciali (il suggerimento è: `c:\djangogirls`). Dopo averlo spostato, prova ad eseguire di nuovo il comando di cui sopra.

Su Linux

Se ottieni un errore quando esegui il comando `pip` su Ubuntu 12.04, prova ad eseguire `python -m pip install -U --force-reinstall pip` per risolvere il problema.

Questo è tutto! Sei (finalmente) pronto/a a creare un'applicazione Django!

Installare un editor di codice

Sono disponibili diversi editor e la scelta di uno piuttosto che un altro dipende principalmente dal gusto personale. La maggior parte dei programmatori Python usa complessi ma estremamente potenti IDE (ambienti di sviluppo integrati), come PyCharm. Tuttavia, dal momento che sei ancora agli inizi non è l'editor più appropriato; quelli che ti suggeriremo noi sono ugualmente potenti ma molto più semplici da utilizzare.

I nostri suggerimenti sono riportati qui di seguito, ma sentiti libero/a di chiedere al tuo coach quali sono le sue preferenze in materia di editor, in questo modo sarà più semplice per il tuo coach aiutarti.

Gedit

Gedit è un editor open-source e gratuito, disponibile per tutti i sistemi operativi.

[Scaricalo qui](#)

Sublime Text 3

Sublime Text è uno tra gli editor più utilizzati. Ha un periodo di prova gratuito. È molto facile da installare e da utilizzare ed è disponibile per tutti i sistemi operativi.

[Scaricalo qui](#)

Atom

Atom è un nuovo editor di codice creato da [GitHub](#). È gratuito, open-source, facile da installare e da usare. È disponibile per Windows, OSX e Linux.

[Scaricalo qui](#)

Perché installiamo un editor di codice?

Forse ti stai chiedendo per quale motivo installiamo questo editor di codice invece di usare un applicazione come Word o Blocco Note.

Il primo motivo è che il codice deve essere **testo semplice**, e il problema con programmi come Word e Textedit è che in realtà non producono testo semplice. Producono testo RTF (con caratteri e formattazione), utilizzando formati personalizzati come [RTF \(Rich Text Format\)](#).

La seconda ragione è che i code editor sono specializzati per programmare, perciò hanno molte funzionalità utili, ad esempio diversi colori per evidenziare frammenti di codice con diversi significati, o l'inserimento automatico del secondo paio di virgolette.

Vedremo tutto ciò più tardi. Il tuo fidato code editor sarà presto uno dei tuoi strumenti preferiti :)

Installare Git

Windows

È possibile scaricare Git da [git-scm.com](#). Puoi saltare tutti i passaggi tranne uno. Nel quinto passaggio, dal titolo "Regolazione della variabile PATH di sistema", scegli "Esegui Git e gli strumenti Unix associati dalla riga di comando di Windows" (l'opzione in basso). A parte questo, i valori predefiniti vanno bene. 'Checkout Windows-style' e 'commit Unix-style line endings' vanno bene.

MacOS

Scarica Git da [git-scm.com](#) e segui le istruzioni.

Linux

Se non è già installato, git dovrebbe essere disponibile tramite il gestore di pacchetti, prova a cercare:

```
sudo apt-get install git
# oppure
sudo yum install git
# oppure
sudo zypper install git
```

Creare un account GitHub

Vai su [GitHub.com](#) e iscriviti per un nuovo account gratuito.

Creare un account PythonAnywhere

Ora è il momento di registrarsi per un account gratuito "Beginner" su PythonAnywhere.

- www.pythonanywhere.com

Nota Quando scegli il tuo nome utente, tieni conto che l'URL del tuo blog diventerà `iltuounusername.pythonanywhere.com`, quindi scegli il tuo nickname oppure un nome ispirato a ciò su cui si basa il tuo blog.

Inizia la lettura

Complimenti, ora sei pronta! Se hai un po' di tempo prima del workshop, potrebbe essere utile cominciare a leggere i capitoli iniziali:

- [Come funziona Internet](#)
- [Introduzione alla riga di comando](#)
- [Introduzione a Python](#)
- [Che cos'è Django?](#)

Come funziona Internet

Questo capitolo è ispirato ad un discorso di Jessica McKellar "How the Internet Works" (<http://web.mit.edu/jessstess/www/>).

Scommettiamo che usi Internet tutti i giorni. Ma sai davvero che cosa succede quando digitri un indirizzo come [https://django.org](https://.djangoproject.org) nel tuo browser e premi ?

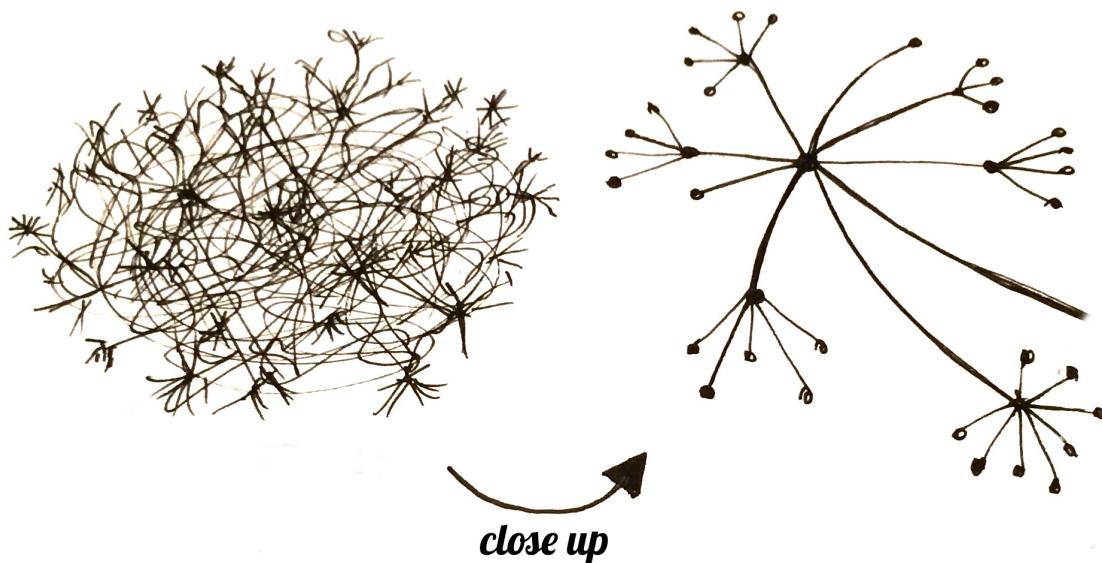
La prima cosa da capire è che un sito è solo un gruppo di file salvati su un hard disk. Proprio come i tuoi film, la tua musica e le tue immagini. Tuttavia, c'è una caratteristica tipica dei siti web: includono un codice chiamato HTML.

Se non hai familiarità con la programmazione, può essere difficile da capire l'HTML all'inizio, ma i tuoi web browser (come Chrome, Safari, Firefox, ecc) lo adorano. I browser sono progettati per capire questo codice, seguire le sue istruzioni e presentare questi file che costituiscono il tuo sito web esattamente nel modo desiderato.

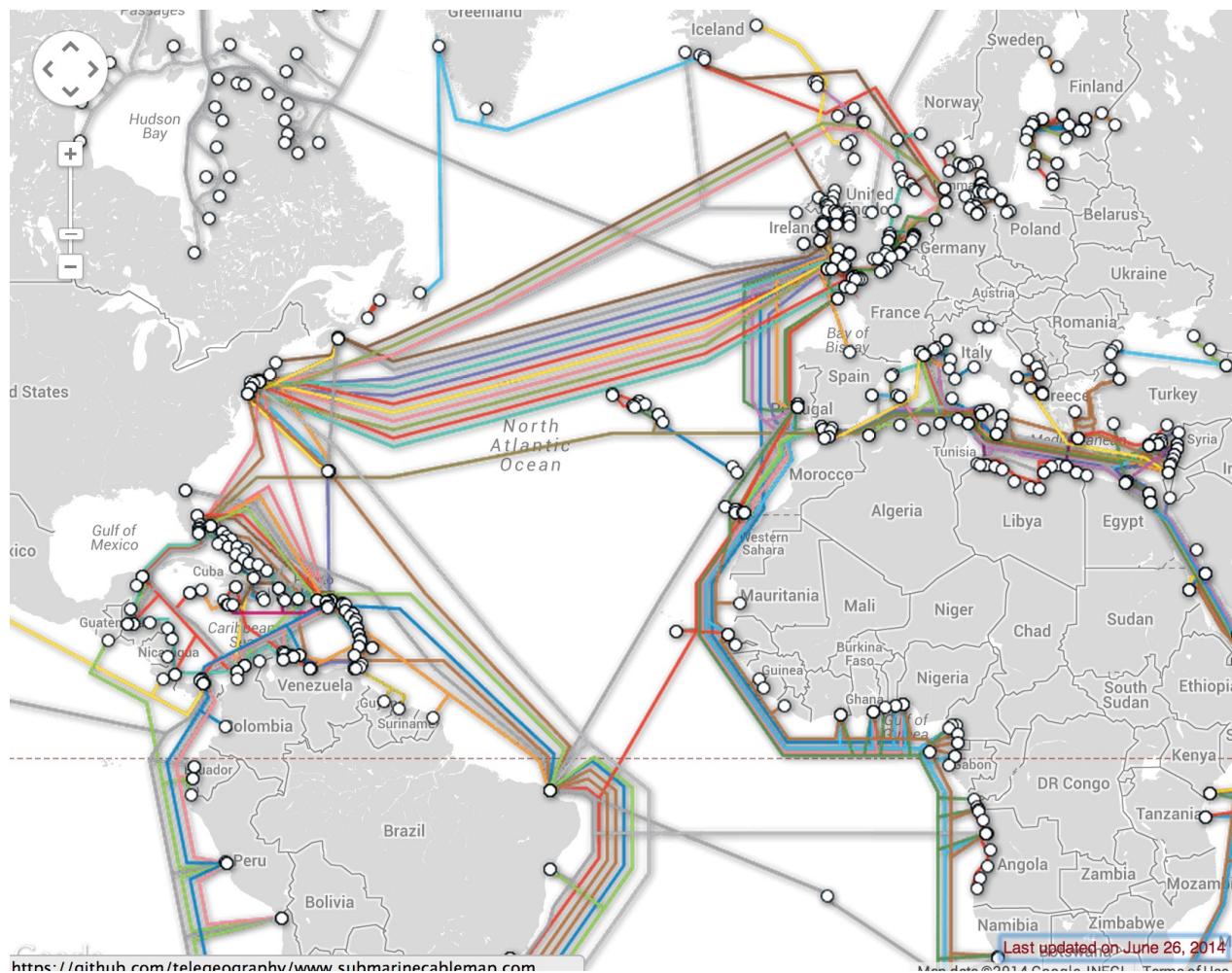
Come per tutti i file, dobbiamo archiviare i file HTML da qualche parte su un hard disk. Per l'Internet, utilizziamo computer speciali e potenti chiamati *servers*. Non hanno uno schermo, un mouse o una tastiera, perché il loro unico proposito è quello di archiviare i dati e fornirli. È per questo che vengono chiamati *servers* -- perché essi servono i tuoi dati.

OK, ma tu vuoi sapere com'è internet, vero?

Abbiamo creato un'immagine! Ecco com'è:

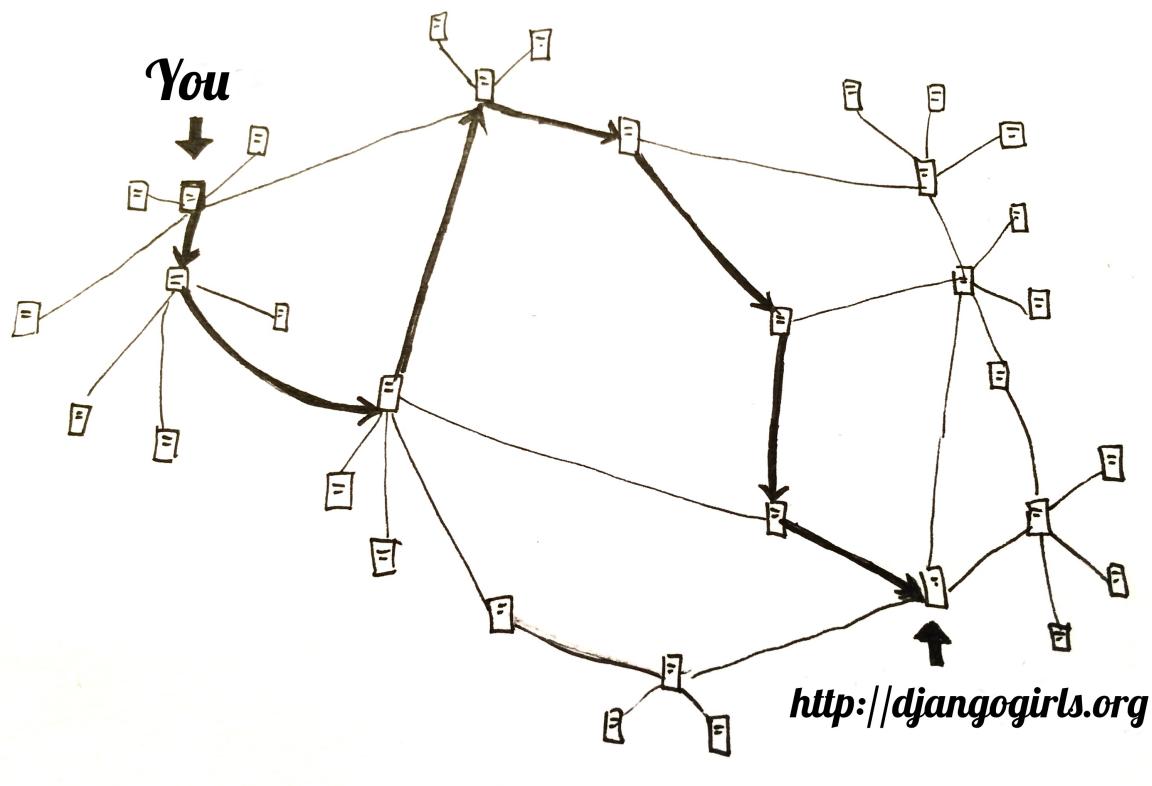


Sembra caotico, vero? Infatti è una rete di macchine collegate (i *servers* che abbiamo menzionato prima). Centinaia di migliaia di macchine! Molti, molti chilometri di cavi in tutto il mondo! Puoi visitare un sito di Submarine Cable Map (<http://submarinecablemap.com>) per vedere quanto è complicata la rete. Ecco uno screenshot dal sito web:



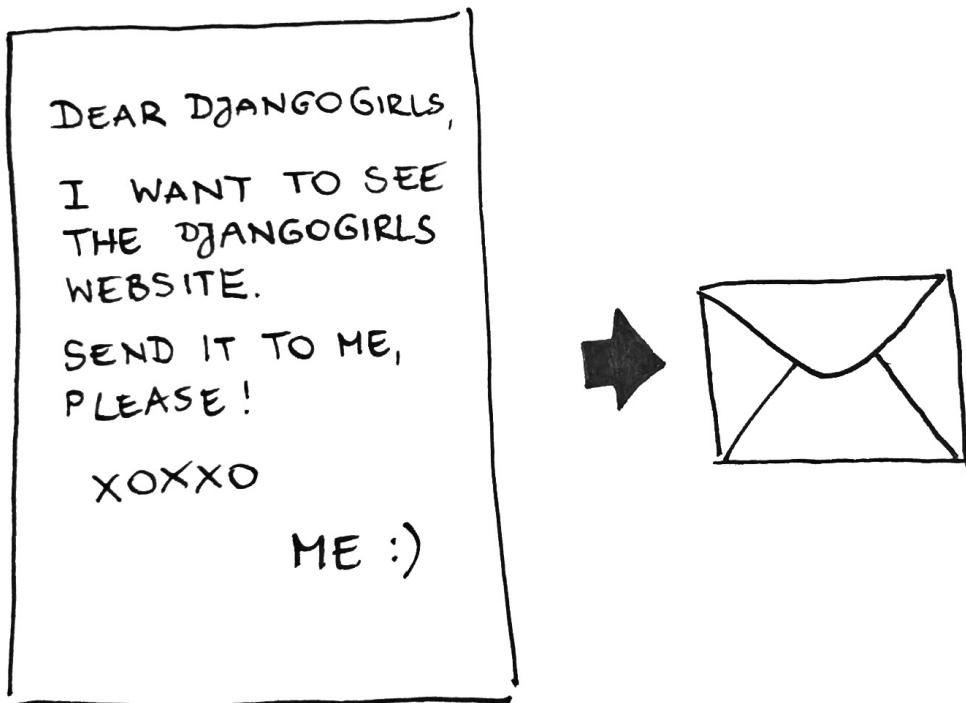
È affascinante, non è vero? Ma ovviamente, non è possibile avere un cavo fra ogni macchina collegata ad Internet. Quindi, per raggiungere una macchina (per esempio quella in cui è salvato [https://django.org](https://.djangoproject.org)) dobbiamo far passare una richiesta attraverso a molte, molte macchine diverse.

Assomiglia a questo:



Immagina che quando digitvi <https://djangogirls.org> invii una lettera che dice: "Caro Django Girls, voglio vedere il sito djangogirls.org. inviamelo, per favore!"

La tua lettera arriva nell'ufficio postale più vicino a te. Dopo di che va in un altro ufficio postale, che è un po' più vicino al tuo destinatario, poi in un altro ed in un altro ancora finché viene consegnato a destinazione. L'unica cosa è che se invii molte lettere (*pacchi di dati*) allo stesso posto, potrebbero attraversare uffici postali totalmente diversi (*routers*). Questo dipende da come vengono distribuiti presso ogni ufficio.



Si, è esattamente così. Tu invii messaggi e ti aspetti una risposta. Certo, invece di carta e penna usi i bytes di dati, ma l'idea è la stessa!

Al posto di indirizzi fisici, ovvero del nome della via, della città, del Cap, e del nome del Paese, usiamo indirizzi IP. Il tuo computer prima chiede il DNS (Domain Name System) per tradurre djangogirls.org in un indirizzo IP. Funziona un po' come i vecchi elenchi telefonici, dove cercando il nome della persona che volevi contattare potevi trovare il numero telefonico e l'indirizzo.

Quando invii una lettera, deve avere determinate caratteristiche per essere consegnata correttamente: un indirizzo, un timbro, ecc. Inoltre utilizzi un linguaggio che il destinatario è in grado di capire, vero? Lo stesso vale per i *pacchi di dati* che invii per vedere un sito Web. Usiamo un protocollo chiamato HTTP (Hypertext Transfer Protocol).

Quindi, praticamente, quando hai un sito, devi avere un *server* (macchina) dove archiviarlo. Quando il *server* riceve una *richiesta* (in una lettera), restituisce il tuo sito (in un'altra lettera).

Dal momento che questo è il tutorial di Django, ti chiederai cosa fa Django. Quando invii una risposta, non vuoi inviare la stessa cosa a tutti. È molto meglio se le tue lettere sono personalizzate, soprattutto per la persona che ti ha appena scritto, giusto? Django ti aiuta con la creazione di queste interessanti lettere personalizzate :).

Basta parlare, è arrivata l'ora di creare!

Introduzione alla linea di comando

Eccitante, vero? Scriverai la tua prima riga di codice in pochi minuti :)

Ti presentiamo il tuo primo nuovo amico: la linea di comando!

I prossimi passaggi ti mostreranno come utilizzare quella 'finestra nera' che tutti gli hacker utilizzano. Ti potrà sembrare un po' allarmante all'inizio, ma è solamente un prompt in attesa dei tuoi comandi.

Nota Nota bene: in tutto questo tutorial usiamo sia il termine 'directory' che 'cartella' ma sono la stessa cosa.

Cos'è la command line?

La finestra solitamente chiamata **command-line** o **interfaccia della command-line**, è un'applicazione basata su testo che ti permette di visualizzare, gestire e manipolare i file sul tuo computer. Molto simile a Windows Explorer o al Finder su Mac, ma senza l'interfaccia grafica. Altri nomi per la command line sono: *cmd*, *CLI*, *prompt*, *console* o *terminal*.

Aprire l'interfaccia di command-line

Per cominciare a sperimentare dobbiamo aprire l'interfaccia della nostra command-line.

Windows

Vai a menù Start → tutti i programmi → accessori → prompt dei comandi.

Mac OS X

Applicazioni → utilità → terminal.

Linux

Probabilmente è sotto Applicazioni → Accessori → Terminal, ma quello potrebbe dipendere dal tuo sistema. Se non è lì cerca lo su Google :)

Prompt

Ora dovresti essere in grado di vedere una finestra bianca o nera che è in attesa di ricevere un comando.

Se sei su Mac o Linux, probabilmente vedi `$`, proprio come questo:

```
$
```

Su Windows, è un segno `>`, come questo:

```
>
```

Ogni comando sarà preceduto da questo simbolo e da uno spazio, ma tu non hai bisogno di digitarlo. Il computer lo farà per te :)

Solo una piccola nota: nel tuo caso ci dovrebbe essere qualcosa come `C:\Users\ola>` oppure `olas-MacBook-Air:~ ola$` prima del segno di prompt. È corretto al 100%. In questo tutorial lo semplificheremo al massimo.

Il tuo primo comando (YAY!)

Cominciamo con qualcosa di veramente semplice. Digita questo comando:

```
$ whoami
```

oppure

```
> whoami
```

Premi `invio`. Questo è il nostro risultato:

```
$ whoami  
olasitarska
```

Come puoi vedere, il computer ha appena stampato il tuo nome utente. Bello, eh?:)

Prova a digitare ogni comando, non copiare ed incollare. Ti ricorderai di più in questo modo!

Nozioni di base

Ogni sistema operativo ha un insieme di comandi leggermente diverso per la command line, per cui assicurati di seguire le istruzioni per il tuo sistema operativo. Proviamo questo, ti va?

Cartella corrente

Sarebbe bello sapere dove siamo adesso, vero? Vediamo. Digita questo comando e premi `invio`:

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

Se sei su Windows:

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

Probabilmente vedrai qualcosa di simile sul tuo computer. Quando apri la command-line normalmente inizi sulla tua directory home.

Nota: 'pwd' sta per 'stampa directory di lavoro'.

Elenco di file e cartelle

Cosa c'è dentro? Sarebbe bello scoprirlo. Vediamo come:

```
$ 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  
...
```

Cambiare cartella corrente

Ora, andiamo nella nostra directory Desktop:

```
$ cd Desktop
```

Windows:

```
> cd Desktop
```

Controlla ora se ti sei veramente spostato/a:

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

Windows:

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

Ecco fatto!

Suggerimento PRO: se digit `cd D` e poi premi `tab` sulla tastiera, la command-line completerà automaticamente il resto del nome per cui puoi navigare più velocemente. Se c'è più di una cartella che comincia con "D", premi `tab` due volte per ottenere la lista con tutte le opzioni.

Creare una directory

Che ne dici di creare una directory di pratica sul tuo desktop? Puoi farlo in questo modo:

```
$ mkdir practice
```

Windows:

```
> mkdir practice
```

Questo breve comando creerà una cartella con il nome `practice` sul tuo desktop. Puoi controllare se è lì semplicemente guardando sul tuo desktop oppure eseguendo un command `ls` oppure `dir !` Provalo :)

Suggerimento PRO: se non vuoi digitare lo stesso comando tutte le volte, prova a premere `freccia in su` e `freccia in giù` sulla tua tastiera per scorrere tutti i comandi che hai usato fin ora.

Esercizio!

Piccola sfida per te: nella tua directory appena creata `practice` crea una directory chiamata `test`. Usa i comandi `cd` e `mkdir`.

Soluzione:

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

Windows:

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

Congratulazioni! :)

Facciamo ordine

Non vogliamo lasciare un pasticcio, per cui rimuoviamo tutto quello che abbiamo fatto fino a questo punto.

Per prima cosa dobbiamo tornare al Desktop:

```
$ cd ..
```

Windows:

```
> cd ..
```

Usando `..` con il comando `cd` cambierai la tua directory attuale alla directory padre (si tratta della cartella che contiene la tua directory attuale).

Controlla dove ti trovi ora:

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

Windows:

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

Adesso è l'ora di cancellare la directory `practice`:

Attenzione: cancellare un file usando `del`, `rmdir` o `rm` è irreversibile, i file cancellati andranno perduti per sempre! Per cui sii molto prudente nell'utilizzare questi comandi.

```
$ rm -r practice
```

Windows:

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

Fatto! Per essere sicuri che sia stato effettivamente cancellato, controlliamo:

```
$ ls
```

Windows:

```
> dir
```

Uscire dalla command line

Questo è tutto per ora! puoi tranquillamente chiudere la tua command line. Facciamolo alla maniera degli hacker, va bene?:)

```
$ exit
```

Windows:

```
> exit
```

Figo, eh?:)

Indice

Questo è un riepilogo di alcuni comandi utili:

Comandi(Windows)	Comandi (Mac OS / Linux)	Descrizione	Esempio
exit	exit	chiudi la finestra	exit
cd	cd	cambiare directory	cd test
dir	ls	elenco directory/file	dir
copy	cp	copia un file	copy c:\test\test.txt c:\windows\test.txt
move	mv	spostare un file	move c:\test\test.txt c:\windows\test.txt
mkdir	mkdir	creare una nuova directory	mkdir testdirectory
del	rm	eliminare un file/directory	del c:\test\test.txt

Questi sono solo alcuni dei comandi che puoi eseguire sulla tua command line, ma non ne userai altri oltre a quelli spiegati oggi.

Se sei curioso/a, ss64.com contiene una guida completa ai comandi per tutti i sistemi operativi.

Fatto?

Tuffiamoci in Python!

Iniziamo con Python

Finalmente siamo qui!

Ma prima, permettici di introdurre brevemente cos'è Python. Python è un linguaggio di programmazione molto popolare che può essere utilizzato per creare siti web, giochi, software scientifici, grafici e molto, molto altro.

Python ha avuto origine alla fine degli anni '80 ed il suo obiettivo principale è quello di essere leggibile dagli esseri umani (non solo dalle macchine!). Per questo motivo sembra molto più semplice di altri linguaggi di programmazione. Questo lo rende facile da imparare, ma non ti preoccupare, Python è anche molto potente!

Installazione di Python

Nota Se hai fatto la procedura di installazione, non c'è bisogno di farlo di nuovo - puoi saltare dritto/a al prossimo capitolo!

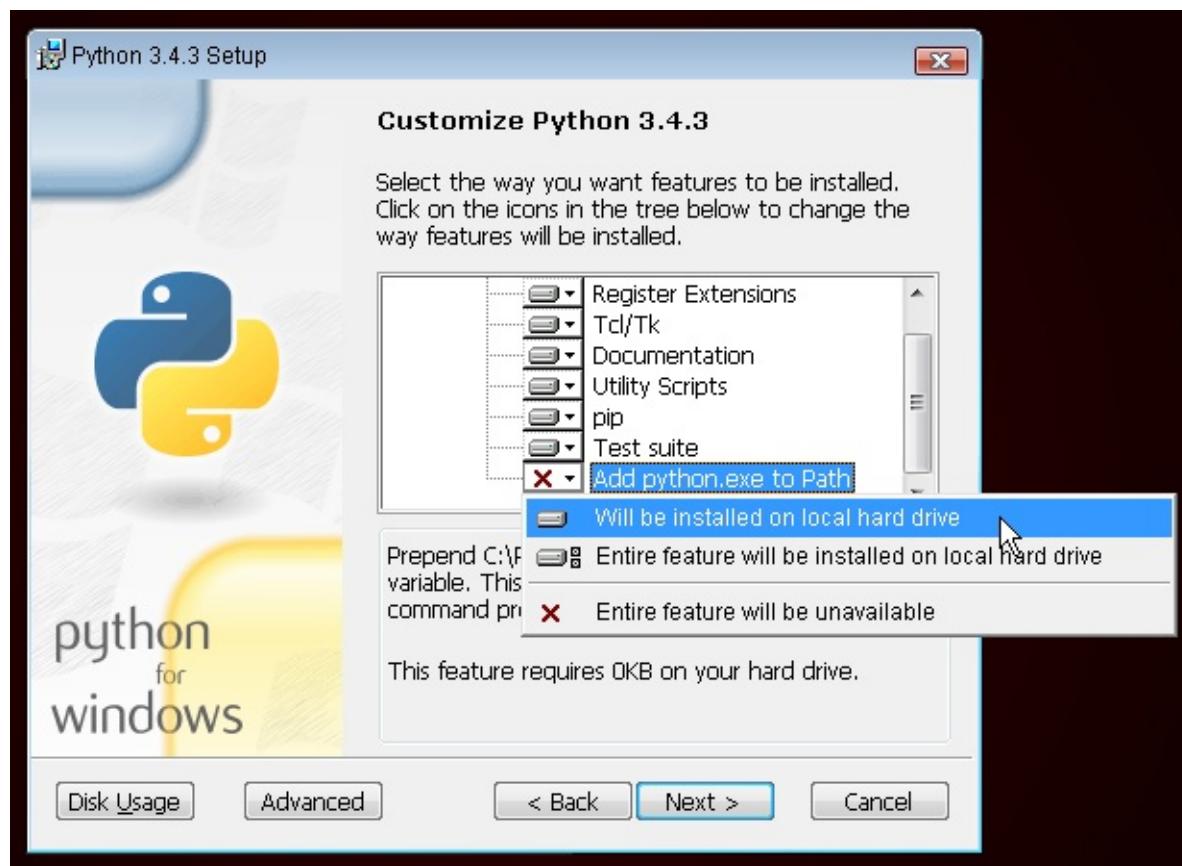
Questa sezione si basa su un tutorial fatto da Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>)

Django è scritto in Python. Abbiamo bisogno di Python per fare qualsiasi cosa in Django. Iniziamo con l'installazione! Vogliamo che sul tuo pc sia installato Python 3.4 quindi se hai una versione precedente, dovrà aggiornarlo.

Windows

Puoi scaricare Python per Windows dal sito web <https://www.python.org/downloads/release/python-343/>. Dopo aver scaricato il file ***.msi**, lo dovrà eseguire (cliccaci sopra due volte) e segui le istruzioni. È importante ricordare il percorso (la directory) dove ha installato Python. Più tardi sarà necessario!

Una cosa a cui fare attenzione: sulla seconda schermata dell'installazione guidata, contrassegnata "Customize", assicurati di scorrere verso il basso e di scegliere l'opzione "Add python.exe to the Path", come illustrato qui:



Linux

È molto probabile che tu abbia Python già installato di default. Per controllare se ce l'hai già installato (e quale versione è), apri una console e digita il seguente comando:

```
$ python3 --version  
Python 3.4.3
```

Se non hai Python installato o se vuoi una versione diversa, puoi installarla come segue:

Debian o Ubuntu

Digita questo comando nella tua console:

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

Fedora (fino a 21)

Usa questo comando nella tua console:

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

Fedora (22+)

Usa questo comando nella tua console:

```
$ sudo dnf install python3.4
```

openSUSE

Usa questo comando nella tua console:

```
$ sudo zypper install python3
```

OS X

Devi andare sul sito <https://www.python.org/downloads/release/python-343/> e scarica il programma d'installazione di Python:

- Scarica il file *Mac OS X 64-bit/32-bit installer*
- Fai doppio click su *python-3.4.3-macosx10.6.pkg* per eseguire il programma d'installazione.

Verifica che l'installazione si sia conclusa correttamente aprendo l'applicazione *Terminal* ed eseguendo il comando

```
python3 :
```

```
$ python3 --version
Python 3.4.3
```

Se hai dubbi o se qualcosa è andato storto e non hai idea di cosa fare dopo - chiedi al tuo insegnante! A volte le cose non vanno come dovrebbero ed è meglio chiedere aiuto a qualcuno con più esperienza.

Code Editor

Stai per scrivere la tua prima riga di codice, per cui è il momento di scaricare il tuo editor!

Nota Potresti aver già affrontato questo passaggio nel capitolo Installazione - se è così, puoi saltare direttamente al prossimo capitolo!

Sono disponibili diversi editor e la scelta di uno piuttosto che un altro dipende principalmente dal gusto personale. La maggior parte dei programmati Python usa complessi ma estremamente potenti IDE (ambienti di sviluppo integrati), come PyCharm. Tuttavia, dal momento che sei ancora agli inizi non è l'editor più appropriato; quelli che ti suggeriremo noi sono ugualmente potenti ma molto più semplici da utilizzare.

I nostri suggerimenti sono riportati qui di seguito, ma sentiti libero/a di chiedere al tuo coach quali sono le sue preferenze in materia di editor, in questo modo sarà più semplice per il tuo coach aiutarti.

Gedit

Gedit è un editor open-source e gratuito, disponibile per tutti i sistemi operativi.

[Scaricalo qui](#)

Sublime Text 3

Sublime Text è uno tra gli editor più utilizzati. Ha un periodo di prova gratuito. È molto facile da installare e da utilizzare ed è disponibile per tutti i sistemi operativi.

[Scaricalo qui](#)

Atom

Atom è un nuovo editor di codice creato da [GitHub](#). È gratuito, open-source, facile da installare e da usare. È disponibile per Windows, OSX e Linux.

[Scaricalo qui](#)

Perché installiamo un editor di codice?

Forse ti stai chiedendo per quale motivo installiamo questo editor di codice invece di usare un'applicazione come Word o Blocco Note.

Il primo motivo è che il codice deve essere **testo semplice**, e il problema con programmi come Word e Textedit è che in realtà non producono testo semplice. Producono testo RTF (con caratteri e formattazione), utilizzando formati personalizzati come [RTF \(Rich Text Format\)](#).

La seconda ragione è che i code editor sono specializzati per programmare, perciò hanno molte funzionalità utili, ad esempio diversi colori per evidenziare frammenti di codice con diversi significati, o l'inserimento automatico del secondo paio di virgolette.

Vedremo tutto ciò più tardi. Il tuo fidato code editor sarà presto uno dei tuoi strumenti preferiti :)

Introduzione a Python

Parte di questo capitolo è basato su esercitazioni di Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Scriviamo un pò codice!

La linea di comando di Python

Per iniziare a giocare con Python, devi avviare sul tuo computer una *linea di comando*. Dovresti già sapere come farlo -- l'hai imparato nel capitolo [Introduzione a Command Line](#).

Una volta pronta, segui le istruzioni riportate di seguito.

Vogliamo aprire una console Python, quindi digita `python` su Windows o `python3` su Mac OS/Linux e premi `invio`.

```
$ python3
Python 3.4.3 (...)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Il tuo primo comando Python!

Dopo aver eseguito il comando Python, il prompt è cambiato in `>>>`. Significa che per ora dovremmo utilizzare comandi nel linguaggio Python. Non devi digitare `>>>` - Python lo farà per te.

Se ad un certo punto vuoi uscire dalla console di Python, digita `exit()` oppure usa la scorciatoia `Ctrl + Z` per Windows e `Ctrl + D` per Mac/Linux. Allora non vedrai più `>>>`.

Per ora non vogliamo uscire dalla console Python. Vogliamo saperne di più. Cominciamo con qualcosa davvero semplice. Per esempio, prova un po' di matematica, come `2 + 3` e premi `invio`.

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

Fantastico! Hai visto come è comparsa la risposta? Python conosce la matematica! potresti provare altri comandi come: -

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

Divertiti con questo per un pò e dopo torna qui :).

Come puoi vedere, Python è una buona calcolatrice. Ora ti starai sicuramente chiedendo cos'altro è capace di fare...

Stringhe

Che ne dici di scrivere il tuo nome? Digitalo tra virgolette così:

```
>>> "ola"
'ola'
```

Hai appena creato la tua prima stringa! Una stringa è una sequenza di caratteri che possono essere elaborati da un computer. La stringa deve sempre iniziare e finire con lo stesso carattere. Che può essere una virgoletta semplice (`'`) o doppia (`"`) (non c'è differenza!) Le virgolette dicono a Python che il contenuto al loro interno è una stringa.

Le stringhe possono essere legate assieme. Prova questo:

```
>>> " Ciao " + "ola"
'Ciao Ola'
```

Puoi anche moltiplicare le stringhe con un numero:

```
>>> "Ola" * 3
'olaOlaOla'
```

Se devi mettere un apostrofo nella tua stringa, hai due modi per farlo.

Utilizzando le virgolette doppie:

```
>>> "Correre verso l'albero"
"Correre verso l'albero"
```

o facendo l'escape dell'apostrofo (cioè trattandolo come un carattere qualunque) con una barra rovesciata (\):

```
>>> 'Correre verso l\'albero'
"Correre verso l'albero"
```

Bello, eh? Per vedere il tuo nome in maiuscolo, digita:

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

Hai appena usato la funzione `upper` sulla tua stringa! Una funzione (come `upper()`) è una sequenza di istruzioni che Python deve eseguire su un determinato oggetto (`"Ola"`).

Se vuoi sapere il numero delle lettere presenti nel tuo nome, c'è una funzione anche per quello!

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

Ti stai chiedendo perché certe volte chiavi una funzione con un `.` alla fine di una stringa (come `"Ola".upper()`) ed in altri casi chiavi prima una funzione e poi metti la stringa tra parentesi? Beh, in alcuni casi, le funzioni appartengono ad oggetti, come `upper()`, che può essere eseguita solo su stringhe. In questo caso, chiamiamo la funzione **metodo**. Altre volte, le funzioni non appartengono a niente di specifico e possono essere utilizzate su diversi tipi di oggetti, proprio come `len()`. Ecco perché stiamo dando `"Ola"` come un parametro alla funzione `len`.

Indice

OK, basta con le stringhe. Ecco fino ad ora quanto hai imparato:

- **il prompt** - digitare i comandi (codice) nel prompt di Python restituisce risposte in Python
- **numeri e stringhe** - in Python i numeri vengono utilizzati per la matematica e le stringhe per oggetti testuali
- **operatori** - come `+` e `*`, combinano i valori per produrne di nuovi
- **funzioni** - come `upper()` e `len()`, eseguono azioni su oggetti.

Queste sono le basi di ogni linguaggio di programmazione che impari. Pronta per qualcosa di più complicato? Scommetto che lo sei!

Errori

Proviamo qualcosa di nuovo. Possiamo ottenere la lunghezza di un numero nella stessa forma in cui abbiamo potuto scoprire la lunghezza del nostro nome? Digita `len(304023)` e premi `Invio`:

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

Abbiamo ottenuto il nostro primo errore! Ci dice che gli oggetti di tipo "int" (integers, numeri interi) non hanno lunghezza. Quindi cosa possiamo fare? Forse possiamo scrivere il nostro numero come una stringa? Le stringhe hanno una lunghezza, vero?

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

Ha funzionato! Usiamo la funzione `str` all'interno della funzione `len`. `str()` converte tutto in stringhe.

- La funzione `str` converte le cose in **stringhe**
- La funzione `int` converte le cose in **numeri interi**

Importante: possiamo convertire i numeri in testo, ma non possiamo convertire il testo in numeri - cosa potrebbe essere `int('hello')` ?

Variabili

Un concetto importante nella programmazione è quello delle variabili. Una variabile è un nome per un qualcosa che deve essere utilizzato sucessivamente. I programmati usano queste variabili per archiviare dati, rendere il loro codice più leggibile e per non dover tenere a mente cosa sono queste cose.

Diciamo che vogliamo creare una nuova variabile chiamata `nome`:

```
>>> nome = "Ola"
```

Vedi? È facile! è semplicemente: `nome` è uguale a Ola.

Come avrai notato, il programma non ha ritornato nulla, diversamente da prima. Quindi come facciamo a sapere che la variabile esiste? Digita `nome` e premi `enter`:

```
>>> nome
'ola'
```

Evvai! La tua prima variabile :)! Puoi sempre modificare a cosa si riferisce:

```
>>> nome = "Sonja"
>>> nome
'Sonja'
```

La puoi utilizzare anche nelle funzioni:

```
>>> len(nome)
5
```

Fantastico, vero? Certo, le variabili possono essere qualsiasi cosa, così come i numeri! Prova questo:

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

Ma cosa succede se utilizziamo il nome sbagliato? Riesci a immaginare cosa succederebbe? Proviamo!

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

Un errore! Come puoi vedere, Python ha diversi tipi di errori e questo qui si chiama **NameError**. Python ti darà questo errore se provi ad utilizzare una variabile che non è stata ancora definita. Se incontri questo errore più tardi, controlla il tuo codice per vedere se hai digitato in modo errato i nomi.

Giocaci per un po' e vedi cosa puoi fare!

La funzione print

Prova questo:

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

Quando digiti `nome`, l'interprete di Python risponde con una stringa *rappresentazione* della variabile 'nome', che contiene le lettere M-a-r-i-a, circondate da singole virgolette, ". Quando dici `print(nome)`, Python "stamperà" i contenuti della variabile sullo schermo, senza le virgolette, che è più pulito.

Come vedremo dopo, `print()` è anche utile quando vogliamo stampare le cose dall'interno delle funzioni, oppure quando vogliamo stampare le cose in molteplici righe.

Liste

Oltre alle stringhe ed ai numeri interi, Python ha tanti tipi di oggetti. Ora ne introdurremo uno chiamato **lista**. Le liste sono fatte proprio come immagini: sono oggetti che sono liste di altri oggetti :)

Vai avanti e crea una lista:

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

Si, questa lista è vuota. Non serve a molto, giusto? Creiamo una lista di numeri della lotteria. Non vogliamo ripetere tutto ogni volta, quindi metteremo la lista in una variabile:

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

Abbiamo una lista! Cosa possiamo farne? Vediamo quanti numeri della lotteria ci sono nella lista. Hai idea di quale funzione potresti utilizzare per farlo? Lo abbiamo imparato insieme prima!

```
>>> len(lotteria)
6
```

Si! `len()` può darti il numero di oggetti in una lista. Utile, vero? Forse abbiamo risolto:

```
>>> lotteria.sort()
```

Questo comando non dà nessun risultato, ha semplicemente cambiato l'ordine in cui i numeri appaiono nella lista.

Stampiamo di nuovo la lista per vedere cosa è successo:

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

Come puoi vedere, adesso i numeri nella tua lista sono ordinati dal valore più basso a quello più alto. Congratulazioni!

Vogliamo invertire quell'ordine? Facciamolo!

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

Facile, vero? Se vuoi aggiungere qualcosa alla tua lista, puoi farlo digitando questo comando:

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

Se vuoi mostrare solo il primo numero, puoi farlo usando gli **indici**. L'indice è il numero che dice la posizione esatta dell'elemento all'interno di una lista. I programmati preferiscono iniziare a contare da 0, quindi il primo oggetto nella lista è all'indice 0, il successivo all'1, e così via. Prova questo:

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

Come puoi vedere, puoi accedere a diversi oggetti nella tua lista usando il nome della lista e l'indice dell'oggetto all'interno delle parentesi quadre.

Per eliminare qualcosa dalla lista dovrà usare **indexes** come abbiamo visto sopra, e lo statement `pop()`. Proviamo a fare qualcosa per rafforzare quanto imparato prima; elimineremo il primo numero della lista.

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

Ha funzionato a meraviglia!

Prova altri indici: 6, 7, 1000, -1, -6 o -1000. Prova a prevedere il risultato ancora prima di eseguire il comando. Hanno senso i risultati?

Per saperne di più su i metodi disponibili per le liste puoi consultare questo capitolo della documentazione Python:
<https://docs.python.org/3/tutorial/datastructures.html>

Dizionari

Un dizionario (dictionary) è simile a una lista, ma accedi ai valori cercando una chiave invece di un indice. Una chiave può essere qualsiasi stringa o numero. La sintassi per definire un dizionario vuoto è:

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

Questo dimostra che hai appena creato un dizionario vuoto. Evviva!

Ora, prova a scrivere il seguente comando (prova a sostituirlo con le tue informazioni):

```
>>> partecipante = {'nome': 'Ola', 'paese': 'Polonia', 'numeri_preferiti': [7, 42, 92]}
```

Con questo comando hai appena creato una variabile chiamata `partecipante` con tre coppie di chiavi-valori:

- La chiave `nome` va a indicare il valore `'Ola'` (un oggetto `stringa`),
- `paese` indica `'Polonia'` (un'altra `stringa`),
- e `numeri_preferiti` indica `[7, 42, 92]` (una `lista` con tre numeri al suo interno).

Puoi controllare il contenuto di chiavi individuali con questa sintassi:

```
>>> print(partecipante['nome'])
Ola
```

Vedi, assomiglia ad una lista. Ma non devi ricordare l'indice – solo il nome.

Cosa succede se chiediamo a Python il valore di una chiave che non esiste? Riesci a indovinarlo? Proviamo!

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

Guarda, un altro errore! Questo qua è un **KeyError**. Python è utile e ti dice che la chiave `'età'` non esiste in questo dizionario.

Quando usare un dizionario o una lista? Bella domanda. Prova a formulare una soluzione mentalmente prima di vedere la risposta nella prossima riga.

- Ha bisogno di una sequenza ordinata di elementi? Fai una lista.
- Hai bisogno di associare i valori alle chiavi, così che potrai cercarle agilmente (per chiave) dopo? Usa un dizionario.

I dizionari, come le liste, sono *mutable*, significa che possono essere cambiati dopo che sono stati creati. Si possono aggiungere nuove coppie chiave/valore ad un dizionario dopo averlo creato:

```
>>> partecipante['linguaggio_preferito'] = 'Python'
```

Così come succede se applicato alle liste, il metodo `len()` restituisce il numero di coppie chiave/valore anche quando applicato a un dizionario. Vai e digita il comando:

```
>>> len(partecipante)
4
```

Spero che abbia senso per te. :) Pronta per divertirti con i dizionari? Vai alla prossima riga per realizzare altre cose fantastiche.

Puoi usare il comando `pop()` per cancellare un elemento nella directory. Se vuoi cancellare la voce che corrisponde alla chiave `'numeri_preferiti'`, digita il seguente comando:

```
>>> partecipante.pop('numeri_preferiti')
>>> partecipante
{'paese': 'Polonia', 'linguaggio_preferito': 'Python', 'nome': 'Ola'}
```

Come puoi vedere dall'output, la coppia chiave-valore corrispondente alla chiave 'numeri_preferiti' è stata cancellata.

Puoi anche cambiare un valore associato ad una chiave già creata nel dizionario. Digita:

```
>>> partecipante['paese'] = 'Germania'
>>> partecipante
{'paese': 'Germania', 'linguaggio_preferito': 'Python', 'nome': 'Ola'}
```

Come puoi vedere, il valore della chiave 'paese' è stato cambiato da 'Polonia' a 'Germania' . :) Eccitante, vero? Evviva! Hai già imparato un'altra cosa fantastica.

Indice

Fantastico! ora sai molto sulla programmazione. In questa ultima parte hai imparato:

- **errori** - ora sai come leggere e capire gli errori che appaiono se Python non comprende un comando che gli hai dato
- **variabili** - nomi per oggetti. Ti permettono di scrivere codice più semplice e di renderlo più leggibile
- **liste** - liste di oggetti archiviati in un ordine particolare
- **dizionari** - oggetti archiviati come coppie di chiave-valore

Sei emozionato/a per la prossima parte? :)

Confrontare le cose

Larga parte della programmazione include il confrontare le cose. Qual è la cosa più semplice da confrontare? I numeri, senza dubbio. Vediamo come funziona:

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

Abbiamo dato a Python alcuni numeri da mettere a confronto. Come puoi vedere, Python può mettere a confronto non solo numeri, ma anche i risultati dei metodi. Forte, eh?

Ti sei chiesta perché abbiamo messo due simboli di uguale `==` uno vicino all'altro per confrontare i numeri? Usiamo un singolo `=` per assegnare valori alle variabili. Sempre, **sempre** devi mettere due `==` se vuoi controllare se le cose sono uguali. Possiamo affermare anche che le cose sono diverse tra di loro. Per dirlo, usiamo il simbolo `!=`, come mostrato nell'esempio sopra.

Dai a Python altri due compiti:

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

> e < sono facili, ma cosa significano `>=` e `<=`? Leggili così:

- `x > y` significa: x è maggiore di y
- `x < y` significa: x è minore di y
- `x <= y` significa: x è minore o uguale a y
- `x >= y` significa: x è maggiore o uguale a y

Fantastico! Vuoi farne due o tre? prova questo:

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

Puoi dare a Python tutti i numeri da confrontare che vuoi, ti darà sempre una risposta! Molto intelligente, vero?

- **and** - se usi l'operatore `and`, entrambe le cose confrontate devono essere True in modo tale che l'intero comando sia True
- **or** - se usi l'operatore `or`, solo una delle cose messe a confronto deve essere True in modo tale che l'intero comando sia True

Hai sentito parlare dell'espressione "comparare mele e arance"? Proviamo l'equivalente in Python:

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

Dall'espressione puoi capire che Python non è in grado di mettere a confronto un numero (`int`) e una stringa (`str`). Ci mostra invece un **TypeError** e ci dice che i due tipi non possono essere messi a confronto.

Booleano

Accidentalmente, hai appena imparato un nuovo tipo di oggetto in Python. Si chiama **Boolean** e probabilmente è il tipo più facile che ci sia.

Ci sono solo due oggetti Boolean:

- `True`
- `False`

Ma perché Python possa capirlo, bisogna sempre scrivere `True` (prima lettera maiuscola, le altre minuscole). **true**, **TRUE**, **tRUE** non funzionano -- solo **True** è corretto. (Lo stesso vale per `False`, ovviamente.)

I Boolean possono anche essere variabili! Guarda qui:

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

Lo puoi fare anche in questa forma:

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

Fai pratica e divertiti con i Boolean provando ad eseguire i seguenti comandi:

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

Congratulazioni! I valori Boolean sono tra le cose più interessanti della programmazione e tu hai appena imparato ad utilizzarli!

Salvalo!

Finora abbiamo scritto il codice python nell'interprete, che ci permette di inserire una linea di codice per volta. I programmi vengono salvati in file ed eseguiti dall'**interprete** del nostro linguaggio di programmazione o dal **compiler**. Fino ad ora abbiamo eseguito i nostri programmi una riga per volta nell' **interprete** di Python. Avremo bisogno di più di una riga di codice per i prossimi compiti, quindi dovremo fare queste cose velocemente:

- Uscire dall'interprete di Python
- Aprire l'editor di codice che abbiamo scelto
- Salvare un po' di codice in un file python
- Eseguirlo!

Per uscire dall'interprete di Python che è quello che stavamo utilizzando, digita la funzione `exit()` :

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

Questo ti immetterà nel prompt dei comandi.

Prima, abbiamo preso un editore di codice dalla sezione [code editor](#). Dovremo aprire l'editor ora e scrivere un po' di codice in un nuovo file:

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

Nota Dovresti notare una delle cose più belle degli editori di codice: i colori! Nella console Python ogni cosa era dello stesso colore, mentre ora dovresti visualizzare la funzione `print` di un colore differente rispetto alla stringa che la segue. Questa viene chiamata "sintassi evidenziata", ed è veramente utile quando si scrive codice. Il colore serve come suggerimento, ad esempio per una stringa che non è chiusa, o un errore di digitazione di una keyword (come la `def` per le funzioni, che vedremo più avanti). Questo è uno dei motivi per cui usiamo un editor di codice :)

Ovviamente a questo punto sei una programmatrice Python senior, quindi sentiti libera di scrivere un po' del codice che hai imparato oggi.

Ora dobbiamo salvare il file e dargli un nome descrittivo. Chiama il file **python_intro.py** e salvalo sulla tua scrivania. Puoi chiamare il file come vuoi, ma è importante assicurarsi che finisca con **.py**. L'estensione **.py** dice al Sistema Operativo che questo è un **file eseguibile python** e che Python può eseguirlo.

È ora di eseguire il file! Usando le nozioni che hai imparato nella sezione command line, usa il terminal per **cambiare cartella** alla scrivania.

Su un Mac, il comando assomiglierà a questo:

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

Su Linux, sarà come questo (la parola "Desktop" potrebbe essere tradotta nella tua lingua):

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

E su windows, sarà come questo:

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

Se rimani bloccata, chiedi aiuto.

Ora usa Python per eseguire il codice nel file:

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

Perfecto! Hai appena eseguito il tuo primo programma Python salvato su un file. Grande, no?

Ora puoi continuare con uno strumento essenziale nella programmazione:

If...elif...else

Molte cose dovrebbero essere eseguite soltanto quando si incontrano certe condizioni. È per questo che Python ha gli **if statements**.

Sostituisci il codice nel file **python_intro.py** con questo:

```
if 3 > 2:
```

Se salviamo questo codice e lo eseguiamo, incontriamo un errore come questo:

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

Python si aspetta che gli vengano fornite ulteriori istruzioni che saranno eseguite se la condizione `3 > 2` risulterà vera (o `True` se vogliamo). Proviamo a fare in modo che Python stampi "Funziona!". Modifica il tuo codice nel tuo file **python_intro.py** con questo:

```
if 3 > 2:
    print('Funziona!')
```

Vedi come abbiamo indentato la riga successiva usando 4 spazi? Si deve fare così in modo tale che Python sappia quale codice eseguire se il risultato è True. Puoi fare uno spazio, ma circa tutti i programmati di Python ne fanno 4 per far vedere le cose più ordinate. Anche un singolo `tab` conta come 4 spazi.

Salvalo ed eseguilo di nuovo:

```
$ python3 python_intro.py
Funziona!
```

E se una condizione non è Vera?

In esempi precedenti, il codice è stato eseguito solo quando le condizioni erano True. Ma Python ha anche gli `elif` e `else` statements:

```
if 5 > 2:
    print('5 è infatti maggiore di 2')
else:
    print('5 non è maggiore di 2')
```

Quando viene lanciato, mostrerà:

```
$ python3 python_intro.py
5 è infatti maggiore di 2
```

Se 2 fosse un numero maggiore di 5, allora andrebbe in esecuzione il secondo comando. Facile, vero? Andiamo a vedere come funziona `elif`:

```
nome = 'Sonja'
if nome == 'Ola':
    print('Ciao Ola!')
elif nome == 'Sonja':
    print('Ciao Sonja!')
else:
    print('Ciao anonimo!')
```

ed eseguito:

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

Hai visto cosa è successo? `elif` ti consente di aggiungere condizioni supplementari che verranno eseguite se nessuna delle condizioni precedenti viene soddisfatta.

Allo statement iniziale `if` puoi far seguire tutti gli statement `elif` che vuoi. Per esempio:

```
volume = 57
if volume < 20:
    print("Piuttosto basso.")
elif 20 <= volume < 40:
    print("Adatto per musica di sottofondo")
elif 40 <= volume < 60:
    print("Perfetto, posso apprezzare ogni dettaglio")
elif 60 <= volume < 80:
    print("Ideale per le feste")
elif 80 <= volume < 100:
    print("Un po' altino!")
else:
    print("Oddio, le mie orecchie! :(")
```

Python esegue ogni singolo test in sequenza e scrive:

```
$ python3 python_intro.py
Perfetto, posso apprezzare ogni dettaglio
```

Indice

Nei tre esercizi precedenti hai imparato:

- come **confrontare le cose** - in Python puoi mettere a confronto le cose usando `>`, `>=`, `==`, `<=`, `<` e gli operatori `e`, `o`
- i **Boolean** - una tipologia di oggetto che può avere solo uno di questi due valori: `True` o `False`
- come **Salvare file** - archiviare codice nei file in modo da poter eseguire programmi più lunghi.
- **if...elif...else** - affermazioni che ti permettono di eseguire codice solo quando vengono incontrate certe condizioni.

È ora dell'ultima parte del capitolo!

Le funzioni personalizzate!

Ti ricordi quelle funzioni che puoi eseguire in Python come `len()`? Beh, buone notizie, ora imparerai a scrivere delle funzioni tutte tue!

Una funzione è una sequenza di istruzioni che Python dovrebbe eseguire. Ogni funzione in Python inizia con la parola chiave `def`, viene assegnato un nome e può avere alcuni parametri. Iniziamo con una facile. Sostituisci il codice nel file `python_intro.py` con il seguente:

```
def ciao():
    print('Ciao!')
    print('Come stai?')

ciao()
```

Okay, la nostra prima funzione è pronta!

Ti starai chiedendo perché abbiamo scritto il nome della funzione alla fine del file. Perché Python legge il file e lo esegue dall'alto verso il basso. Quindi per poter utilizzare la nostra funzione, dobbiamo riscriverla alla fine.

Eseguiamolo e vediamo cosa succede:

```
$ python3 python_intro.py
Ciao!
Come stai?
```

È stato facile! Costruiamo la nostra prima funzione con parametri. Useremo l'esempio precedente - una funzione che dice 'ciao' alla persona che lo esegue - aggiungendo il nome:

```
def ciao(nome):
```

Come puoi vedere, abbiamo dato alla nostra funzione un parametro chiamato `nome`:

```
def ciao(nome):
    if nome == 'Ola':
        print('Ciao Ola!')
    elif nome == 'Sonja':
        print('Ciao Sonja!')
    else:
        print('Ciao anonimo!')

ciao()
```

Ricorda: La funzione `print` è rientrata di 4 spazi rispetto allo statement `if`. Infatti, la funzione viene eseguita quando la condizione viene soddisfatta. Vediamo ora come funziona:

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

Ops, un errore. Fortunatamente, Python ci fornisce un messaggio di errore che ci può servire. Ci dice che la funzione `ciao()` (quella che abbiamo definito) ha un argomento richiesto (chiamato `nome`) e che ci siamo dimenticati di metterlo quando abbiamo chiamato la funzione. Sistemiamolo alla fine del file:

```
ciao("Ola")
```

Ed eseguiamo di nuovo:

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

E se cambiamo il nome?

```
ciao("Sonja")
```

Ed eseguo:

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

Ora, cosa pensi che succederà se scrivi un altro nome? (non Ola o Sonja) Provaci e vedi se la tua ipotesi è giusta.

Dovrebbe stampare questo:

```
Ciao anonimo!
```

Fantastico, vero? In questo modo non devi ripetere tutto ogni volta che vuoi modificare il nome della persona che la funzione dovrebbe salutare. Ed è esattamente per questo che abbiamo bisogno delle funzioni - non vuoi ripetere il tuo codice!

Facciamo una cosa più intelligente -- ci sono più di due nomi, e scrivere una condizione per ognuno sarebbe complicato, vero?

```
def ciao(nome):
    print('Ciao ' + nome + '!')

ciao("Rachel")
```

Ora chiamiamo il codice:

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

Congratulazioni! Hai appena imparato a scrivere delle funzioni :)

Loop

È l'ultima parte. Abbiamo fatto in fretta, vero? :)

I programmati non amano ripetere ciò che scrivono. La programmazione mira a automatizzare le cose, non vorremo mica salutare ognuno col suo nome manualmente? Ecco un caso in cui i loop ci tornano comodi.

Ti ricordi ancora delle liste? Facciamo una lista di ragazze:

```
ragazze = ['Rachel', 'Monica', 'Phoebe', 'Ola', 'Tu']
```

Vogliamo salutare tutte loro per nome. Abbiamo la funzione `ciao` per farlo, quindi usiamola in loop:

```
for nome in ragazze:
```

Lo statement `for` si comporta in modo simile allo statement `if`; il codice sottostante deve essere rientrato di quattro spazi.

Qua c'è l'intero codice che sarà nel file:

```

def ciao(nome):
    print('Ciao ' + nome + '!')

ragazze = ['Rachel', 'Monica', 'Phoebe', 'Ola', 'Tu']
for nome in ragazze:
    ciao(nome)
    print('Prossima ragazza')

```

E quando lo eseguiamo:

```

$ python3 python_intro.py
Ciao Rachel!
Prossima ragazza
Ciao Monica!
Prossima ragazza
Ciao Phoebe!
Prossima ragazza
Ciao Ola!
Prossima ragazza
Ciao You!
Prossima ragazza

```

Come puoi vedere, tutto quello che metti all'interno di un `for` statement con una spaziatura si ripeterà per ogni elemento della lista `girls`.

Puoi anche utilizzare `for` su numeri usando la funzione `range`:

```

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

```

Che stamperà:

```

1
2
3
4
5

```

`range` è una funzione che crea una lista di numeri che si seguono uno dopo l'altro (questi numeri vengono forniti da te come parametri).

Nota che il secondo di questi due numeri non è incluso nella lista prodotta da Python (ciò significa che `range(1, 6)` conta da 1 a 5, ma non include il numero 6). Questo perché "range" è mezzo-aperto e con ciò intendiamo che include il primo valore, ma non l'ultimo.

Indice

È tutto. **Sei grande!** Questo capitolo non era affatto facile, puoi essere orgogliosa di te stessa. Noi siamo fierissimi di te per avercela fatta fino a qui!

Potresti desiderare di fare brevemente qualcos'altro - stiracchiati, fai due passi, riposa gli occhi - prima di continuare con il prossimo capitolo. :)



Che cos'è Django?

Django (/dʒæŋgoʊʃ/jang-goh) è un framework per applicazioni web gratuito e open source, scritto in Python. Un web framework è un insieme di componenti che ti aiuta a sviluppare siti web più velocemente e facilmente.

Quando si costruisce un sito web, si ha sempre bisogno di un insieme di componenti simili: un sistema per gestire l'autenticazione dell'utente (registrazione, accesso, logout), un pannello di amministrazione per il tuo sito web, un sistema per caricare i file, ecc.

Fortunatamente, alcune persone diverso tempo fa si sono rese conto che gli sviluppatori web incontrano problemi simili ogni volta che costruiscono un sito internet. Per questo motivo si sono uniti e hanno costruito dei framework (di cui Django fa parte) che offrono componenti già pronti per l'uso.

I frameworks esistono per evitare che tu debba reinventare la ruota e ti semplificano il lavoro quando crei un nuovo sito.

Perché ho bisogno di un framework?

Per capire Django, abbiamo bisogno di dare un'occhiata più da vicino ai server. La prima cosa che il server deve sapere è che vuoi che ti fornisca una pagina web.

Immagina una cassetta delle lettere che monitora tutte le lettere in entrata (richieste). Questo è ciò che fa un web server. Il web server legge le lettere, e invia una risposta con una pagina web. Ma quando vuoi inviare qualcosa, hai bisogno di avere qualche contenuto. E Django è ciò che ti aiuta a creare questo contenuto.

Cosa succede quando qualcuno richiede un sito Web dal tuo server?

Quando una richiesta arriva al web server, viene passata a Django che prova a capire che cosa è stato veramente richiesto. Django prende l'indirizzo della pagina web e cerca di capire cosa deve fare. Questa parte viene svolta da Django **urlresolver** (nota che l'indirizzo di una pagina web si chiama URL - Uniform Resource Locator -per cui il nome *urlresolver* acquista significato). Non è molto intelligente -ha bisogno di una serie di schemi e in seguito prova a far corrispondere l'URL. Django controlla questi modelli o schemi da cima a fondo e se qualcosa corrisponde a quel punto Django passa la richiesta alla funzione associata (che si chiama *view*).

Immagina un postino con una lettera. Sta camminando per la strada e controlla ogni numero civico mettendolo a confronto con quello sulla lettera. Se corrisponde, mette lì la lettera. Questo è il modo in cui l'urlresolver funziona!

Nella funzione *view* avvengono tutte le cose più interessanti: possiamo consultare un database alla ricerca di qualche informazione. Forse l'utente ha richiesto di cambiare qualcosa all'interno dei suoi dati? È come una lettera che dice "Per favore cambia la descrizione del mio lavoro". La *view* può controllare se sei autorizzata a farlo, poi aggiorna la descrizione del lavoro per te e manda un messaggio: "Fatto!". In seguito, la *view* genera una risposta e Django la può inviare al browser dell'utente.

Naturalmente, la descrizione qui sopra è molto semplificata, ma per il momento non hai bisogno di sapere nel dettaglio tutti gli aspetti tecnici. Avere il senso generale per il momento è abbastanza.

Per cui invece di perderci troppo nei dettagli, creeremo semplicemente qualcosa usando Django e apprenderemo i concetti fondamentali lungo la strada!

Installazione di Django

Nota Se hai già eseguito i passaggi per l'Installazione, allora hai già fatto quanto serve e puoi andare direttamente al prossimo capitolo!

Una parte di questo capitolo si basa sui tutorial delle Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Una parte di questo capitolo di base sul [django-marcador tutorial](#) sotto licenza Creative Commons Attribution-ShareAlike 4.0 International License. Il tutorial di django-marcador è protetto da copyright di Markus Zapke-Gründemann et al.

Ambiente virtuale

Prima di installare Django, ti vogliamo far installare uno strumento estremamente utile per tenere in ordine l'ambiente in cui programmerai sul tuo computer. Potresti saltare questo passaggio, ma è caldamente consigliato soffermarsi. Se inizi con la migliore configurazione possibile, ti risparmierai un sacco di problemi per il futuro!

Per cui, creiamo ora un **ambiente virtuale** (chiamato anche un *virtualenv*). Virtualenv isolerà la tua configurazione di Python/Django in base ai diversi progetti. Questo significa che qualunque modifica farai su un sito non avrà alcun effetto su tutti gli altri che stai sviluppando. Chiaro ora?

Tutto quello che devi fare è trovare una cartella in cui vuoi creare il `virtualenv`; la tua home directory, ad esempio. Su Windows potrebbe essere `C:\Users\Name` (dove `Name` è il nome del tuo login).

Per questo tutorial useremo una nuova directory `djangogirls` dalla tua home directory:

```
mkdir djangogirls
cd djangogirls
```

Ora creeremo un virtualenv dal nome `myvenv`. Questo è il formato del comando generale:

```
python3 -m venv myvenv
```

Windows

Per creare un nuovo `virtualenv` è necessario aprire la console (ti abbiamo spiegato come fare nei capitoli precedenti, ricordi?) ed esegui `C:\Python34\python -m venv myvenv`. Il risultato somiglierà a questo:

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

dove `C:\Python34\python` è la directory in cui precedentemente hai installato Python e `myvenv` è il nome del tuo `virtualenv`. Puoi utilizzare qualsiasi altro nome, ma attieniti a utilizzare le minuscole, a non usare spazi, accenti o caratteri speciali. È meglio usare un nome breve dal momento che dovrà digitarlo spesso!

Linux e OS X

Creare un `virtualenv` su Linux e OS X è semplice, basta digitare: `python3 -m venv myvenv`. Il risultato sarà simile a questo:

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

`myvenv` è il nome del tuo `virtualenv`. Puoi usare qualsiasi altro nome, ma utilizza solo minuscole e niente spazi. Sarebbe meglio se il nome fosse corto perché lo dovrà digitare molte volte!

NOTA: Avviare il `virtualenv` su Ubuntu 14.04 in questa maniera al momento darà il seguente errore:

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

Per aggirare il problema utilizza, invece del precedente, il comando `virtualenv`.

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

Lavorare con `virtualenv`

Il comando sopra specificato, creerà una cartella dal nome `myvenv` (o col nome che hai scelto) che contiene il tuo virtual environment (ovvero un mucchio di files e cartelle).

Windows

Avvia il tuo `virtualenv` digitando:

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

Linux e OS X

Avvia il tuo `virtualenv` digitando:

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

Ricordati di sostituire `myvenv` con il nome `virtualenv` che hai scelto!

Nota: a volte il comando `source` potrebbe non essere disponibile. In quel caso prova ad usare questo:

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

Saprai con certezza che hai avviato `virtualenv` quando vedrai che il prompt dei comandi nella console si presenta così:

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

oppure:

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

Fai caso al prefisso `(myvenv)` !

Quando si lavora all'interno di un ambiente virtuale, `python` farà automaticamente riferimento alla versione corretta da utilizzare. Per cui puoi digitare `python` invece `python3`.

OK, abbiamo tutte le dipendenze importanti pronte. Finalmente possiamo installare Django!

Installare Django

Ora che hai iniziato ad utilizzare il tuo `virtualenv`, puoi installare Django usando `pip`. Nella console, esegui `pip install django==1.8` (nota che usiamo un doppio simbolo di uguale: `==`).

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

Su Windows

Se ottieni un errore quando chiami pip sulla piattaforma Windows controlla se il pathname del tuo progetto contiene spazi, accenti o caratteri speciali (i.e. `c:\Users\User Name\djangogirls`). Se è così ti conviene spostarlo in un altro path senza spazi, accenti o caratteri speciali (il suggerimento è: `c:\djangogirls`). Dopo averlo spostato, prova ad eseguire di nuovo il comando di cui sopra.

su Linux

Se ottieni un errore quando esegui il comando pip su Ubuntu 12.04, prova ad eseguire `python -m pip install -U --force-reinstall pip` per risolvere il problema.

Questo è tutto! Sei (finalmente) pronto/a a creare un'applicazione Django!

Il tuo primo progetto Django!

Parte di questo capitolo è basato su esercitazioni di Geek Girls (<https://github.com/ggcarrots/django-carrots>).

Parti di questo capitolo sono basate sul [django-marcador tutorial](#) sotto la licenza Creative Commons Attribution-ShareAlike 4.0 International License. Il tutorial di django-marcador è protetto da copyright di Markus Zapke-Gründemann et al.

Creeremo un semplice blog!

Il primo passo è quello di iniziare un nuovo progetto di Django. Fondamentalmente, questo significa che eseguiremo alcuni script forniti da Django che creerà per noi lo scheletro di un progetto Django. Si tratta solo di un insieme di directory e file che verranno utilizzati dopo.

I nomi di alcuni file e cartelle sono molto importanti per Django. Non dovresti modificare i nomi dei file che stiamo per creare. Neanche spostarli in un altro posto è una buona idea. Django deve mantenere una determinata struttura per essere in grado di trovare le cose importanti.

Ricordati di eseguire tutto nel virtualenv. Se non vedi un prefisso `(myvenv)` nella tua console devi attivare il tuo virtualenv. Abbiamo spiegato come farlo nel capitolo [installazione Django](#) nella parte [Lavorando con virtualenv](#). Digitando `myvenv\Scripts\activate` su Windows oppure `source myvenv/bin/activate` su Mac OS / Linux farà questo per te.

Dovresti eseguire nella tua console MacOS o Linux il seguente comando; **non dimenticarti di aggiungere il punto `.` alla fine**:

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

Su Windows; **non dimenticarti di aggiungere il punto `.` alla fine**:

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

Il punto `.` è cruciale perché dice allo script d'installare Django nell'attuale directory (quindi il punto `.` è un riferimento di abbreviazione)

Nota Quando digitai i comandi sopra, ricorda che si digita soltanto la parte che inizia con `django-admin` oppure `django-admin.py`. Le parti mostrate qui come `(myvenv) ~/djangogirls$` e `(myvenv) C:\Users\Name\djangogirls>` sono solo esempi del prompt che starà invitando il tuo input sulla tua command line.

`django-admin.py` è uno script che creerà le cartelle ed i file per te. Adesso dovresti avere una struttura di directory simile a questa:

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

`manage.py` è uno script che aiuta a gestire il sito. Usandolo saremo in grado di avviare un web server sul nostro computer senza dover installare nient'altro, tra l'altro.

Il file `settings.py` contiene la configurazione del tuo sito web.

Ricordi quando abbiamo parlato di un postino che controlla dove rilasciare la lettera? il file `urls.py` contiene una lista di schemi usati da `urlresolver`.

Ignoriamo gli altri file per ora dal momento che non li modificheremo. L'unica cosa da ricordare è di non cancellarli per sbaglio!

Modificare le impostazioni

Facciamo qualche cambiamento in `mysite/settings.py`. Apri il file usando il code editor che hai installato prima.

Sarebbe bello avere l'ora corretta sul nostro sito Web. Vai alla [lista di fusi orari di wikipedia](#) e copia il tuo fuso orario (TZ). (es. `Europe/Berlin`)

In `settings.py`, trova la riga che contiene `TIME_ZONE` e modificala per scegliere il tuo fuso orario:

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

Modifica "Europe/Berlin" nel modo corretto

Avrai anche bisogno di aggiungere un percorso per i file statici (scopriremo tutto su file statici e CSS più avanti nell'esercitazione). Scendi fino alla *fine* del file e sotto la voce `STATIC_URL`, aggiungi un nuovo percorso chiamato

`STATIC_ROOT` :

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

Imposta un database

Ci sono un sacco di software di database diversi che possono immagazzinare dati per il tuo sito. Noi useremo quello di default, `sqlite3`.

È già impostato in questa parte del file `mysite/settings.py`:

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

Per creare un database per il nostro blog, eseguiamo questo nella console: `python manage.py migrate` (abbiamo bisogno di essere nella directory `djangogirls` che contiene il file `manage.py`). Se funziona, dovresti vedere qualcosa di simile:

```
(myvenv) ~/djangogirls$ python manage.py migrate
Operations to perform:
  Synchronize unmigrated apps: messages, staticfiles
    Apply all migrations: contenttypes, sessions, admin, auth
Synchronizing apps without migrations:
  Creating tables...
    Running deferred SQL...
  Installing custom SQL...
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002.Alter_permission_name_max_length... OK
  Applying auth.0003_Alter_user_email_max_length... OK
  Applying auth.0004_Alter_user_username_opts... OK
  Applying auth.0005_Alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying sessions.0001_initial... OK
```

E abbiamo finito! Tempo di avviare il server web e vedere se il nostro sito Web funziona!

Devi essere nella directory che contiene il file di `manage.py` (la directory `djangogirls`). Nella console, possiamo avviare il server web eseguendo `python manage.py runserver`:

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

Se sei su Windows e non funziona con `UnicodeDecodeError`, usa questo comando:

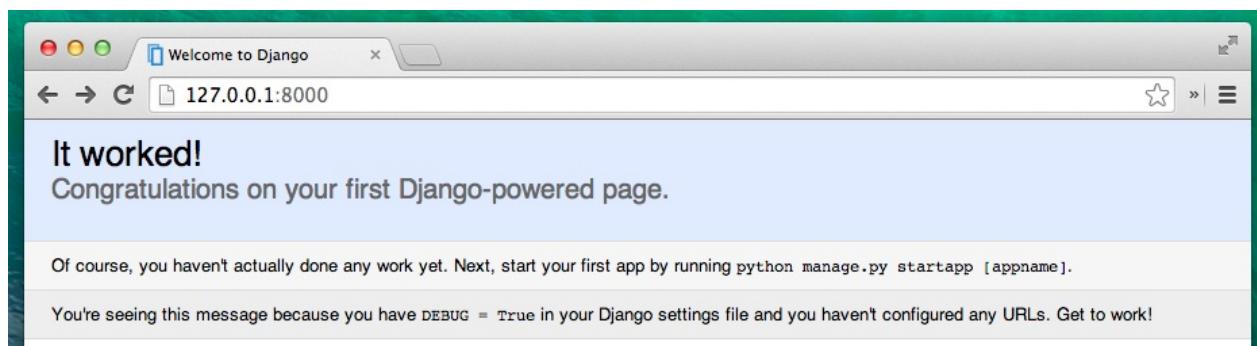
```
(myvenv) ~/djangogirls$ python manage.py runserver 0:8000
```

Ora tutto quello che devi fare è controllare che il tuo sito sia in esecuzione. Apri il tuo browser (Firefox, Chrome, Safari, Internet Explorer o qualsiasi altro tu usi) e digita l'indirizzo:

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

Il server web assumerà il controllo del command prompt finché non lo fermi. Per digitare più comandi mentre è in esecuzione apri una nuova finestra di terminale e attiva il tuo virtualenv. Per fermare il server web, torna alla finestra dove è in esecuzione e premi i pulsanti CTRL+C - Control e C insieme (su Windows, è probabile che tu deva premere Ctrl+Break).

Congratulazioni! Hai appena creato il tuo primo sito e l'hai avviato usando un web server! Non è fantastico?



Pronto/a per il prossimo passo? È ora di creare un po' di contenuti!

Modelli Django

Vogliamo creare un qualcosa che raccoglierà tutti i post del nostro blog. Per farlo, però, dobbiamo prima introdurre i cosiddetti `oggetti`.

Oggetti

Esiste un concetto nella programmazione chiamato `programmazione orientata agli oggetti`. L'idea è che invece di scrivere tutto come una noiosa sequenza di istruzioni di programmazione, possiamo modellare le cose e definire come esse interagiscono fra di loro.

Quindi cos'è un oggetto? È un insieme di proprietà ed azioni. Suona strano, ma ti faremo un esempio.

Se vogliamo modellare un gatto creeremo un oggetto `Gatto` che ha qualche proprietà, i.e. `colore`, `età`, `umore` (i.e. bravo, cattivo, sonnolento ;)), `padrone` (che è un oggetto `Persona` oppure, nel caso di un gatto randagio, questa proprietà è vuota).

E poi il `Gatto` ha alcune azioni: `fusa`, `graffiare` oppure `alimentare` (nella quale daremo al gatto un po' di `cibo` per `gatti`, che potrebbe essere un oggetto diverso con delle proprietà, i.e. `gusto`).

```
Gatto
-----
colore
età
umore
padrone
fare le fusa()
graffiare()
alimentare(cat_food)
```

Quindi in pratica l'idea è quella di descrivere cose vere in codice con delle proprietà (chiamate `proprietà di oggetti`) e azioni (chiamate `metodi`).

Quindi come faremo a modellare i post del blog? vogliamo costruire un blog, giusto?

Dobbiamo rispondere alla domanda: cos'è un post? Quali proprietà dovrebbe avere?

Beh, sicuramente il nostro post ha bisogno di qualche testo con il suo contenuto ed un titolo, vero? Sarebbe bello sapere chi l'ha scritto - quindi abbiamo bisogno di un autore. Infine, vogliamo sapere quando il post è stato creato e pubblicato.

```
Post
-----
titolo
testo
autore
data_creazione
data_pubblicazione
```

Che tipo di cose si potrebbero fare con un post? Sarebbe bello avere qualche `metodo` che pubblica il post, vero?

Quindi avremo bisogno di un metodo `pubblicare`.

Dal momento che sappiamo già cosa vogliamo ottenere, iniziamo a modellarlo in Django!

Modello Django

Sapendo cos'è un oggetto, possiamo creare un modello Django per il nostro post.

Un modello in Django è uno speciale tipo di oggetto - è salvato nel `database`. Un database è un insieme di dati. È un posto in cui archivierai informazioni sui tuoi utenti, sui tuoi post, ecc. Useremo un database SQLite per archiviare i nostri dati. Questo è l'adattatore Django di database predefinito -- ci basterà per adesso.

Puoi pensare ad un modello nel database come ad un foglio elettronico con colonne (campi) e righe (dati).

Creazione di un'applicazione

Per mantenere tutto ordinato, creeremo un'applicazione diversa all'interno del nostro progetto. È molto bello avere tutto organizzato fin dall'inizio. Per creare un'applicazione abbiamo bisogno di eseguire il seguente comando nella console (dalla cartella `djangogirls` dove si trova il file `manage.py`):

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

Noterà che si è creata una nuova cartella `blog` e che ora contiene alcuni file. Le nostre cartelle ed i nostri file nel nostro progetto si dovrebbero vedere così:

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

Dopo aver creato un'applicazione dobbiamo dire a Django che dovrebbe utilizzarla. Lo facciamo nel file `mysite/settings.py`. Dobbiamo trovare `INSTALLED_APPS` ed aggiungere una riga che contenga `'blog'`, appena sopra). Quindi il prodotto finale dovrebbe assomigliare a questo:

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

Creazione di un modello blog post

Nel file `blog/models.py` definiamo tutti gli oggetti chiamati `Models` - Questo è il posto dove definiremo il nostro blog post.

Apriamo `blog/models.py`, cancella tutto quello che è lì e scrivi un codice come questo:

```

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

class Post(models.Model):
    author = models.ForeignKey('auth.User')
    title = models.CharField(max_length=200)
    text = models.TextField()
    created_date = models.DateTimeField(
        default=timezone.now)
    published_date = models.DateTimeField(
        blank=True, null=True)

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

    def __str__(self):
        return self.title

```

Ricontrolla se stai utilizzando due caratteri di sottolineatura (`_`) su ciascun lato di `str`. Questa convenzione viene utilizzata spesso in Python e a volte li chiamiamo anche "dunder" (abbreviazione di "doppio carattere di sottolineatura").

Sembra spaventoso, vero? ma non ti preoccupare, ti spiegheremo cosa significano queste righe!

Tutte le righe che iniziano con `from` oppure con `import` sono righe che aggiungono alcuni pezzi da altri file. Quindi invece di copiare e incollare le stesse cose in ogni file, possiamo includere alcune parti con `from ... import ...`.

`class Post(models.Model):` - questa riga definisce il nostro modello (è un oggetto).

- `class` è una parola chiave speciale che indica che stiamo definendo un oggetto.
- `Post` è il nome del nostro modello. Possiamo dargli un nome diverso (ma dobbiamo evitare caratteri speciali e spazi). Inizia sempre il nome di una classe con un lettera maiuscola.
- `models.Model` significa che il Post è un modello Django, quindi Django sa che dovrebbe essere salvato nel database.

Ora definiamo le proprietà di cui stavamo parlando: `titolo`, `testo`, `data_creazione`, `data_pubblicazione` e `autore`. Per fare ciò dobbiamo definire un tipo per ogni campo (è un testo? Un numero? Una data? Una relazione con un altro oggetto, i.e. un utente?).

- `models.CharField` - così si definisce un testo con un numero limitato di lettere.
- `models.TextField` - questo è il codice per definire un testo senza un limite. Sembra l'ideale per i contenuti di un post, vero?
- `models.DateTimeField` - questo per la data ed l'ora.
- `models.ForeignKey` - questo è un link a un altro modello.

Non spiegheremo ogni pezzo di codice perchè ci vorrebbe troppo tempo. Dovresti dare un'occhiata alla documentazione di Django se vuoi saperne di più sui campi di un modello e come definire altre cose rispetto a quelle descritte sopra (<https://docs.djangoproject.com/en/1.8/ref/models/fields/#field-types>).

Che dire di `def publish(self):`? È esattamente il metodo `pubblicare` di cui stavamo parlando prima. `def` significa che questa è una funzione/metodo e `publish` è il nome del metodo. Puoi modificare il nome del metodo, se vuoi. La regola per la denominazione è usare lettere minuscole e caratteri di sottolineatura al posto degli spazi. Per esempio, un metodo che calcola il prezzo medio potrebbe essere chiamato `calculate_average_price`.

I metodi spesso restituiscono qualcosa. C'è un esempio nel metodo `__str__`. In questo caso, quando chiamiamo `__str__()` otterremo un testo (**stringa**) con il titolo del Post.

Se c'è qualcosa di poco chiaro sui modelli, sentiti libera/o di chiedere al tuo coach! Sappiamo che è complicato, soprattutto quando impari cosa sono gli oggetti e le funzioni allo stesso tempo. Ma speriamo che sembri un po' meno magico per te per adesso!

Crea tabelle per i modelli nel tuo database

L'ultimo passo è quello di aggiungere un nuovo modello al nostro database. Prima dobbiamo far sapere a Django che ci sono alcuni cambiamenti nel nostro modello (l'abbiamo appena creato!). Digita `python manage.py makemigrations blog`. Il risultato somiglierà a questo:

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

Django ci ha preparato un file di migrazione che dobbiamo applicare nel nostro database. Digita `python manage.py migrate blog` e l'output dovrebbe essere:

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

Evviva! Il nostro modello Post ora è nel database! Sarebbe bello poterlo vedere, vero? Vai al prossimo capitolo per vedere com'è il tuo Post!

Django admin

Per aggiungere, modificare e cancellare i post che abbiamo appena strutturato useremo Django admin.

Apri il file `blog/admin.py` e sostituisci il suo contenuto con:

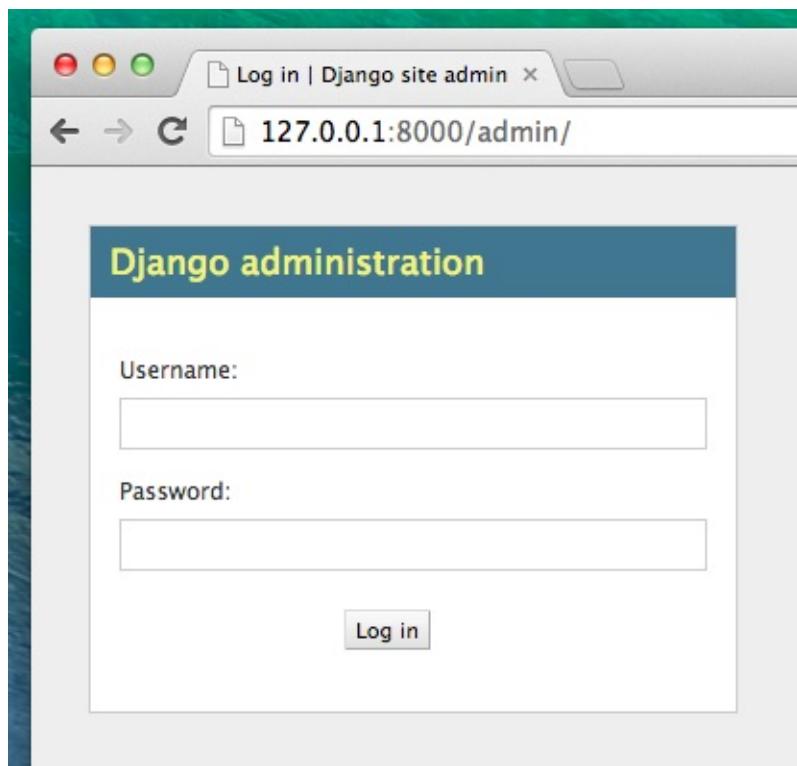
```
from django.contrib import admin
from .models import Post

admin.site.register(Post)
```

Come puoi vedere, stiamo importando (include) il modello di Post che abbiamo definito nel capitolo precedente. Per far sì che il nostro modello sia visibile nella pagina di admin, dobbiamo registrare questo modello con

```
admin.site.register(Post) .
```

OK, è tempo di guardare il nostro modello Post. Ricorda di eseguire `python manage.py runserver` nella console per avviare il web server. Vai nel browser e scrivi l'indirizzo <http://127.0.0.1:8000/admin/> Vedrai una pagina di login come questa:



Per accedere, devi creare un **superuser** - un utente che ha il controllo su tutto nel sito. Torna alla command-line e digita `python manage.py createsuperuser`, e premi invio. Quando richiesto, digita il tuo username (minuscole, senza spazi), indirizzo e-mail e password. Non ti preoccupare se non riesci a vedere la password che stai digitando - è così che dovrebbe essere. Basta digitalo e premere `invio` per continuare. L'output dovrebbe essere così (dove il nome utente e l'email dovrebbero essere i tuoi):

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

Torna nel tuo browser e fai il log in con le credenziali di superuser che hai scelto, dovresti vedere la dashboard d'amministrazione di Django.

The screenshot shows the Django administration interface. At the top, there's a header bar with the title "Site administration | Django" and the URL "127.0.0.1:8000/admin/". Below this is a dark blue header bar with the text "Django administration". The main content area has a light blue header "Site administration". Underneath, there are two rows of links. The first row includes "Auth" (disabled), "Groups" with "Add" and "Change" buttons, and "Users" with "Add" and "Change" buttons. The second row includes "Blog" (disabled), "Posts" with "Add" and "Change" buttons, and a "Logout" link at the bottom right.

Vai su Post ed sperimenta un po'. Aggiungi cinque o sei post. Non preoccuparti del contenuto - puoi semplicemente fare il copia-incolla di parti di testo da questo tutorial per risparmiare tempo :).

Assicurati che almeno due o tre post (ma non tutti) abbiano la data in cui sono stati pubblicati. Ti sarà utile più tardi.

The screenshot shows the "Add post" form in the Django administration interface. The title bar says "Add post | Django site admin" and the URL is "127.0.0.1:8000/admin/blog/post/add/". The main title is "Django administration" with a "Welcome, olasitarska. Change password / Log out" message. The form has several fields: "Author" dropdown set to "olasitarska" with an "Add" button; "Title" input field containing "Cras justo odio, dapibus ac facilisis in, eu"; "Text" rich text editor containing placeholder text about fermentum and tempus; "Created date" field with "Date: 2014-07-07 Today" and "Time: 23:07:34"; and "Published date" field with "Date: 2014-07-07" and "Time: 23:07:34". At the bottom are buttons for "Save and add another", "Save and continue editing", and "Save".

Se vuoi sapere di più riguardo l'admin di Django, dovrà dare un'occhiata alla documentazione di Django:
<https://docs.djangoproject.com/en/1.8/ref/contrib/admin/>

Questo è un buon momento per andare a prendere un caffè (o tè) o qualcosa da mangiare per riprendere le forze. Hai creato il tuo primo modello Django - ti meriti una piccola pausa!

Deploy!

Nota Il seguente capitolo è abbastanza difficile da capire fino in fondo. Non mollare e cerca di portarlo a termine; deployment (termine abbastanza complicato da tradurre, ma indica tutto ciò che tu rendi LIVE, accessibile sul web e non più solo dal tuo computer) è una parte importante nel processo di costruzione di un sito web. Questo capitolo è inserito a metà del tutorial per far sì che il tuo tutor possa aiutarti con il processo leggermente più complesso di messa online del sito. Questo significa che puoi ancora finire il tutorial da sola se sei a corto di tempo.

Fino ad ora il tuo sito è accessibile solo dal tuo computer, ma ora imparerai come metterlo online! Deploying è il processo di pubblicazione online del tuo progetto in modo tale che sia visibile anche da altre persone :).

Come hai già visto, un sito internet ha sede in un server. Ci sono tantissimi server providers disponibili su internet. Noi ne useremo uno che ha un processo di deployment relativamente semplice: [PythonAnywhere](#). Questo provider è gratuito per piccole applicazioni che non hanno troppi visitatori. Sarà quindi perfetto per te al momento.

L'altro servizio esterno che useremo è [GitHub](#), che è un servizio di hosting di codice. Ne esistono altri, ma di questi tempi quasi tutti i programmati hanno un account GitHub e ora lo avrai anche tu!

Useremo GitHub come trampolino di lancio per importare ed esportare il nostro codice su PythonAnywhere.

Git

Git è un "sistema di controllo versione" utilizzato da un sacco di programmati. Questo software può tracciare le modifiche nel corso del tempo ad i file, in questo modo puoi ripristinare successivamente una specifica versione. Un pò come l'opzione "traccia modifiche" in Microsoft Word, ma molto più potente.

Installare Git

Nota Se hai già fatto la procedura di installazione, non devi farlo di nuovo - si può passare alla sezione successiva e iniziare a creare il tuo repository Git.

Windows

È possibile scaricare Git da [git-scm.com](#). Puoi saltare tutti i passaggi tranne uno. Nel quinto passaggio, dal titolo "Regolazione della variabile PATH di sistema", scegli "Esegui Git e gli strumenti Unix associati dalla riga di comando di Windows" (l'opzione in basso). A parte questo, i valori predefiniti vanno bene. 'Checkout Windows-style' e 'commit Unix-style line endings' vanno bene.

MacOS

Scarica Git da [git-scm.com](#) e segui le istruzioni.

Linux

Se non è già installato, git dovrebbe essere disponibile tramite il gestore di pacchetti, prova a cercare:

```
sudo apt-get install git
# oppure
sudo yum install git
# oppure
sudo zypper install git
```

Inizializzare un repository Git

Git tiene traccia delle modifiche a un particolare insieme di file in quello che è chiamato repository di codice (o "repo" in breve). Iniziamone uno per il nostro progetto. Apri la console ed esegui questi comandi nella directory `djangogirls`:

Nota Controlla la directory su cui stai lavorando adesso con il comando `pwd` (OSX/Linux) oppure `cd` (Windows) prima di iniziare il repository. Dovresti essere nella cartella `djangogirls`.

```
$ git init
Initialized empty Git repository in ~/djangogirls/.git/
$ git config --global user.name "Your Name"
$ git config --global user.email you@example.com
```

Dobbiamo inizializzare il repository git solo una volta per ogni progetto (e non dovrà più inserire il nome utente e l'email).

Git memorizzerà le modifiche a tutti i file e le cartelle in questa directory, ma ci sono alcuni file che vogliamo ignorare. Si fa creando un file chiamato `.gitignore` nella directory di base. Apri il tuo editor e crea un nuovo file con questo contenuto:

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

E salvalo come `.gitignore` all'interno della cartella "djangogirls".

Nota Il punto all'inizio del nome del file è importante! Se hai difficoltà nel crearlo (ad esempio, ai Mac non piace quando crei file che iniziano con un punto tramite il Finder), allora usa la funzionalità "Salva con nome" nel tuo editor, è a prova di bomba.

È una buona idea usare il comando `git status` prima di `git add` oppure ogni volta che non sei sicuro di cosa sia cambiato. Questo aiuterà ad evitare eventuali brutte sorprese, come file sbagliati che vengono aggiunti o a cui viene fatto il commit. Il comando `git status` restituisce informazioni riguardanti qualsiasi file non tracciato/modificato/in staging, lo stato del branch e molto altro. L'output dovrebbe essere simile a:

```
$ git status
On branch master

Initial commit

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

    .gitignore
    blog/
    manage.py
    mysite/

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

E finalmente salviamo le nostre modifiche. vai alla tua console ed esegui questi comandi:

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

Pubblichiamo il nostro codice su GitHub

Vai su [GitHub.com](#) e registrati per ottenere un nuovo account gratuito. (Se l'hai già fatto nella preparazione di laboratorio, è fantastico!)

Quindi, crea un nuovo repository, dandogli il nome "my-first-blog". Lascia deselezionata la casella di controllo "initialise with a README", lascia l'opzione di .gitignore vuota (lo abbiamo fatto manualmente) e License su None.

Owner  **hjwp** / **Repository name** 

Great repository names are short and memorable. Need inspiration? How about [ducking-octo-tyrion](#).

Description (optional)

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

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

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

Nota Il nome `my-first-blog` è importante -- potresti scegliere qualcos'altro, ma si ripeterà un sacco di volte nelle istruzioni qui sotto, e dovrà sostituirlo ogni volta. Probabilmente è più facile mantenere il nome `my-first-blog`.

Nella schermata successiva, ti verrà mostrato l'URL per clonare il tuo repo:

The screenshot shows a GitHub repository page for 'hjwp/my-first-blog'. The top navigation bar includes links for 'Explore', 'Gist', 'Blog', and 'Help'. On the right, there are icons for starring, watching, and sharing the repository. The repository name 'hjwp / my-first-blog' is displayed with a blue star icon indicating it's starred. A 'Quick setup' section provides instructions for cloning the repository via HTTPS or SSH. It also recommends including a README, LICENSE, and .gitignore file. Below this, a command-line snippet shows how to initialize a new repository and push it to GitHub. Another section shows how to push an existing repository from the command line.

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

or [HTTPS](https://github.com/hjwp/my-first-blog.git) [SSH](ssh://github.com/hjwp/my-first-blog.git) <https://github.com/hjwp/my-first-blog.git>

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

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

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

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

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

Code Issues Pull requests Wiki Pulse Graphs Settings

Ora abbiamo bisogno di collegare il repository Git sul tuo computer a quello su GitHub.

Digita quanto segue sulla tua console (Sostituisci `<your-github-username>` con il nome utente che hai inserito quando hai creato il account GitHub, ma senza le parentesi angolari):

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

Inserisci il tuo username e la tua password di GitHub. Dovresti vedere qualcosa di simile:

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

Adesso il tuo codice è su GitHub. Ora controlla! Scoprirai che è in bella compagnia - [Django](#), il [Django Girls Tutorial](#) e molti altri grandi progetti open source software usano GitHub per ospitare il loro codice :)

Configurare il nostro blog su PythonAnywhere

Nota Potresti aver già creato un account di PythonAnywhere precedentemente, durante i passaggi di installazione - se è così, non c'è bisogno di farlo nuovamente.

Ora è il momento di registrarsi per un account gratuito "Beginner" su PythonAnywhere.

- www.pythonanywhere.com

Nota Quando scegli il tuo nome utente, tieni conto che l'URL del tuo blog diventerà `iltuouesname.pythonanywhere.com`, quindi scegli il tuo nickname oppure un nome ispirato a ciò su cui si basa il tuo blog.

Scaricare il nostro codice su PythonAnywhere

Quando ti sarai registrata su PythonAnywhere, verrai portata alla tua dashboard o alla pagina «Console». Scegli l'opzione per iniziare una console "Bash" -- quella è la versione PythonAnywhere di una console, proprio come quella sul tuo computer.

Nota PythonAnywhere è basato su Linux, quindi se sei su Windows, la console apparirà un po' diversa da quella sul tuo computer.

Scarichiamo il nostro codice da GitHub e su PythonAnywhere creando un "clone" del nostro repo. Digita quanto segue nella console su PythonAnywhere (non dimenticare di usare il tuo nome utente di GitHub al posto di `<your-github-username>`):

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

Questo scaricherà una copia del tuo codice su PythonAnywhere. Dai un'occhiata digitando `tree my-first-blog` :

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

Creare un virtualenv su PythonAnywhere

Proprio come hai fatto sul tuo computer, puoi creare un virtualenv su PythonAnywhere. Nella console di Bash, digita:

```
cd my-first-blog

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

$ source myvenv/bin/activate

(mvenv) $ pip install django==1.8 whitenoise==2.0
Collecting django
[...]
Successfully installed django-1.8.2 whitenoise-2.0
```

Nota Il passaggio `pip install` può richiedere un paio di minuti. Sii paziente! Ma se richiede più di 5 minuti, c'è qualcosa di sbagliato. Chiedi al tuo coach.

Raccogliere file statici.

Ti stavi chiedendo cos'è quel "whitenoise"? È uno strumento per fornire i cosiddetti "file statici". I file statici sono i file che di solito non cambiano o non eseguono codice di programmazione, come i file HTML o CSS. Funzionano in modo diverso sui server e abbiamo bisogno di uno strumento come "whitenoise" per fornirli.

Scopriremo un po' di più sui file statici più tardi nell'esercitazione, quando modificheremo il CSS per il nostro sito.

Per ora abbiamo solo bisogno di eseguire un comando supplementare chiamato `collectstatic`, sul server. Dice a Django di raccogliere tutti i file statici di cui ha bisogno sul server. Al momento questi sono per lo più file che fanno apparire carino il sito di amministrazione.

```
(mvenv) $ python manage.py collectstatic

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

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

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

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

Digita "yes" e si parte! Non ti piace far stampare al computer pagine e pagine di testo incomprensibile? Faccio sempre piccoli versi per accompagnarla. Brp, brp brp...

```
Copying '/home/edith/my-first-blog/mvenv/lib/python3.4/site-packages/django/contrib/admin/static/admin/js/actions.min.js'
Copying '/home/edith/my-first-blog/mvenv/lib/python3.4/site-packages/django/contrib/admin/static/admin/js/inlines.min.js'
[...]
Copying '/home/edith/my-first-blog/mvenv/lib/python3.4/site-packages/django/contrib/admin/static/admin/css/changelist.css'
Copying '/home/edith/my-first-blog/mvenv/lib/python3.4/site-packages/django/contrib/admin/static/admin/css/base.css'
62 static files copied to '/home/edith/my-first-blog/static'.
```

Creare il database su PythonAnywhere

Ecco un'altra differenza tra il tuo computer ed il server: usa un database diverso. Quindi gli account utente ed i post possono essere diversi sul server rispetto a come appaiono sul tuo computer.

Possiamo inizializzare il database sul server proprio come abbiamo fatto sul tuo computer, con `migrate` e `createsuperuser`:

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

(mvenv) $ python manage.py createsuperuser
```

Pubblicare il nostro blog come una web app

Ora il nostro codice è su PythonAnywhere, il nostro virutualenv è pronto, i file statici sono stati raccolti, ed il database è stato inizializzato. Siamo pronti a pubblicarlo come una web app!

Torna alla dashboard di PythonAnywhere cliccando sul suo logo, e clicca sulla scheda **Web**. Infine, premi **Add a new web app**.

Dopo aver confermato il nome del dominio, scegli **manual configuration** (NB non l'opzione "Django") nella finestra di dialogo. Successivamente, scegli **Python 3.4** e clicca su Avanti per completare la procedura guidata.

Nota assicurati di aver scelto l'opzione "Manual configuration", non l'opzione "Django". Siamo troppo cool per l'installazione di Django di PythonAnywhere di default;-)

Impostare il virtualenv

Verrai portato alla schermata di configurazione PythonAnywhere per tua webapp, che è dove dovrai andare ogni volta che desideri apportare modifiche all'applicazione sul server.

Actions:

Code:
What your site is running.

Source code: You can use the [Files tab](#) to navigate to your app's source code.
 WSGI configuration file: `/var/www/edith_pythonanywhere_com_wsgi.py`
 Python version: 3.4

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

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

Nella sezione "Virtualenv", clicca sul testo rosso che dice "Enter the path to a virtualenv" ed immetti: `/home/<your-username>/my-first-blog/myvenv/`. Clicca sul riquadro blu con il segno di spunta per salvare il percorso prima di andare avanti.

Sostituisci il tuo nome utente come appropriato. se commetti un errore, PythonAnywhere ti avverterà.

Configurare il file WSGI

Django funziona usando il "protocollo WSGI", uno standard per fornire siti Web utilizzando Python, che PythonAnywhere supporta. Il modo in cui configuriamo PythonAnywhere per riconoscere il nostro blog in Django è modificando un file di configurazione WSGI.

Clicca sul link "WSGI configuration file" (nella sezione "Code" nella parte superiore della pagina -- avrà un nome tipo `/var/www/<il-tuo-username>_pythonanywhere_com_wsgi.py`), e accederai ad un editor.

Elimina tutti i contenuti e sostituisce con qualcosa di simile:

```
import os
import sys

path = '/home/<il-tuo-username>/my-first-blog' # usa il tuo username qui
if path not in sys.path:
    sys.path.append(path)

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

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

Nota non dimenticare di mettere il tuo nome utente dove dice `<il-tuo-username>`

Il compito di questo file è dire a PythonAnywhere dove si trova la nostra web app e qual è il nome del file che contiene le impostazioni Django. Inoltre, configura lo strumento di file statici "whitenoise".

Clicca **Save** e poi torna alla scheda **Web**.

Abbiamo finito! Premi il grande pulsante verde **Reload** e potrai vedere la tua applicazione. Troverai un link nella parte superiore della pagina.

Suggerimenti per il debug

Se vedi un errore quando provi a visitare il tuo sito, il primo posto dove cercare qualche info per il debugging è nel tuo **error log**. Troverai un link nella [scheda Web](#) di PythonAnywhere. Vedi se ci sono messaggi di errore lì; i più recenti sono alla fine. I problemi comuni includono:

- Dimenticare uno dei passi che abbiamo fatto nella console: creare il virtualenv, attivarlo, installarci Django, eseguire collectstatic, inizializzazione del database.
- Commettere un errore nel percorso del virtualenv sulla scheda Web -- di solito c'è un piccolo messaggio di errore in rosso, se c'è un problema.
- Commettere un errore nel file di configurazione WSGI -- il percorso che hai trovato per tua cartella my-first-blog è corretto?
- Hai adottato la stessa versione di Python per il tuo virtualenv come hai fatto per la tua app web? entrambe dovrebbero essere 3.4.
- Ci sono alcuni [consigli generali per il debugging sulla wiki di PythonAnywhere](#).

E ricorda, il tuo coach è qui per aiutarti!

Sei live!

La pagina predefinita per il tuo sito dovrebbe dire "Welcome to Django", esattamente come sul tuo Pc locale. Prova ad aggiungere `/admin/` alla fine della URL, e verrai portata al sito di amministrazione. Accedi con il tuo username e password, e vedrai che puoi aggiungere nuovi Post sul server.

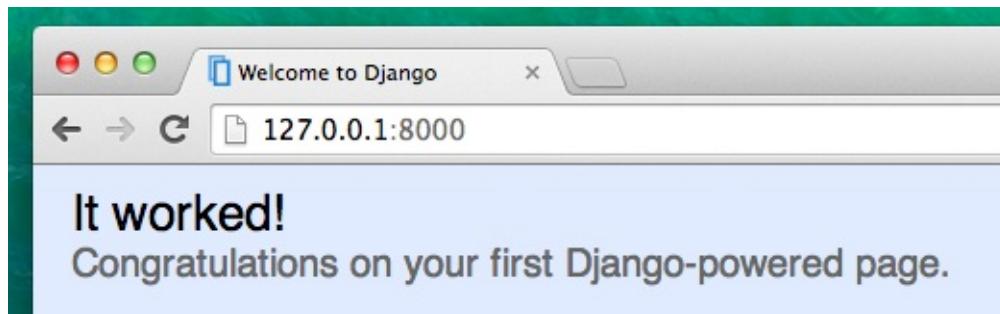
Dà a te stessa un' *ENORME* pacca sulla schiena! Il deploy dei server è tra le parti più complicate dello sviluppo web e di solito le persone ci impiegano svariati giorni prima di farli funzionare. Ma hai pubblicato il tuo sito su Internet senza sforzo!

Django URL

Stiamo per costruire la nostra pagina web: una homepage per il tuo blog! Ma prima, impariamo un po' di più sulle url di Django.

Che cos'è un URL?

Una URL è semplicemente un indirizzo web. Puoi vedere una URL ogni volta che visiti un sito web - si vede nella barra degli indirizzi del tuo browser. (sì! `127.0.0.1:8000` is a URL! Anche `https://djangogirls.org` è una URL):



Ogni pagina internet ha bisogno della sua URL. In questo modo la tua applicazione sa cosa deve mostrare a un utente che visita una URL. In Django usiamo qualcosa chiamato `URLconf` (configurazione dell'URL). URLconf è un insieme di modelli che Django cercherà di far corrispondere con l'URL ricevuta per trovare la view giusta.

Come funzionano le URL in Django?

Apriamo il file `mysite/urls.py` nel code editor che hai scelto e vediamo com'è:

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

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

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

Come puoi vedere, Django ha già predisposto qualcosa per noi in questo file.

Le righe che iniziano con `#` sono commenti - questo significa che non verranno lette da Python. Comodo, non è vero?

L'admin URL , che hai visto nel capitolo precedente è già qui:

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

Questo significa che per ogni URL che comincia con `admin/` Django troverà la corrispondente `view`. In questo caso stiamo includendo un sacco di admin URL così che non sia tutto imballato in questo piccolo file - è più leggibile e più pulito.

Regex

Per caso ti stai chiedendo come fa Python a far corrispondere URL e view? Beh, questa parte è difficile. Django usa `regex`, abbreviazione di "espressioni regolari". Regex ha molte, (moltissime!) regole che costituiscono un modello di ricerca. Dal momento che i regex sono un argomento avanzato, non analizzeremo nel dettaglio come funzionano.

Se vuoi capire come abbiamo creato i patterns, eccoti un esempio del procedimento - ci servirà solo un sottoinsieme limitato di regole per esprimere il modello che stiamo cercando, vale a dire:

```
^ per l'inizio del testo
$ per la fine del testo
\d per una cifra
+ per indicare che l'elemento precedente dovrebbe essere ripetuto almeno una volta
() per catturare parte del pattern
```

Il resto nella definizione dell'url sarà preso alla lettera.

Ora immagina di avere un sito Web con un indirizzo così: `http://www.mysite.com/post/12345/`, dove `12345` è il numero del tuo post.

Scrivere view separate per ogni numero del post sarebbe veramente noioso. Con l'espressione regolare possiamo creare un modello che corrisponde all'url ed estrae il numero per noi: `^ post/(\d+)/ $`. Scomponiamo pezzo per pezzo per vedere che cosa stiamo facendo:

- `^ post /` sta dicendo a Django di prendere tutto ciò che ha `post /` all'inizio dell'url (subito dopo `^`)
- `(\d+)` significa che ci sarà un numero (composto da una o più cifre) e che noi vogliamo che il numero venga catturato ed estratto
- `/` dice a django che ci dovrebbe essere un altro carattere a seguire `/`
- infine, `$` indica la fine dell'URL. Significa che solo le stringhe che terminano con `/` corrisponderanno a questo modello

La tua prima Url Django!

È ora di creare la tua prima URL. Vogliamo usare `http://127.0.0.1:8000/` come homepage per il nostro blog e visualizzare il nostro elenco di post.

Vogliamo anche mantenere il file di `mysite/urls.py` pulito, quindi importeremo le url dalla nostra applicazione `blog` sul file principale `mysite/urls.py`.

Vai avanti, elimina le righe commentate (che cominciano con `#`) e aggiungi una riga che importerà `blog.urls` nella url principale (`''`).

Il tuo file `mysite/urls.py` ora dovrebbe avere questo aspetto:

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

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

Django reindirizzerà ora tutto ciò che viene da `http://127.0.0.1:8000/` verso `blog.urls` e cercherà ulteriori istruzioni in questo file.

Quando si scrivono espressioni regolari in Python lo si fa sempre con `r` davanti alla stringa. Questo è un suggerimento utile per Python che la stringa possa contenere caratteri speciali che non sono destinati per lo stesso Python, ma per l'espressione regolare.

blog.urls

Crea un nuovo file `blog/urls.py`. Perfetto! Ora aggiungi queste prime due righe:

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

Stiamo solo importando metodi che appartengono a Django e tutte le nostre `views` dalla nostra app `blog` (non abbiamo ancora nulla all'interno, ma rimedieremo a questo in un minuto!)

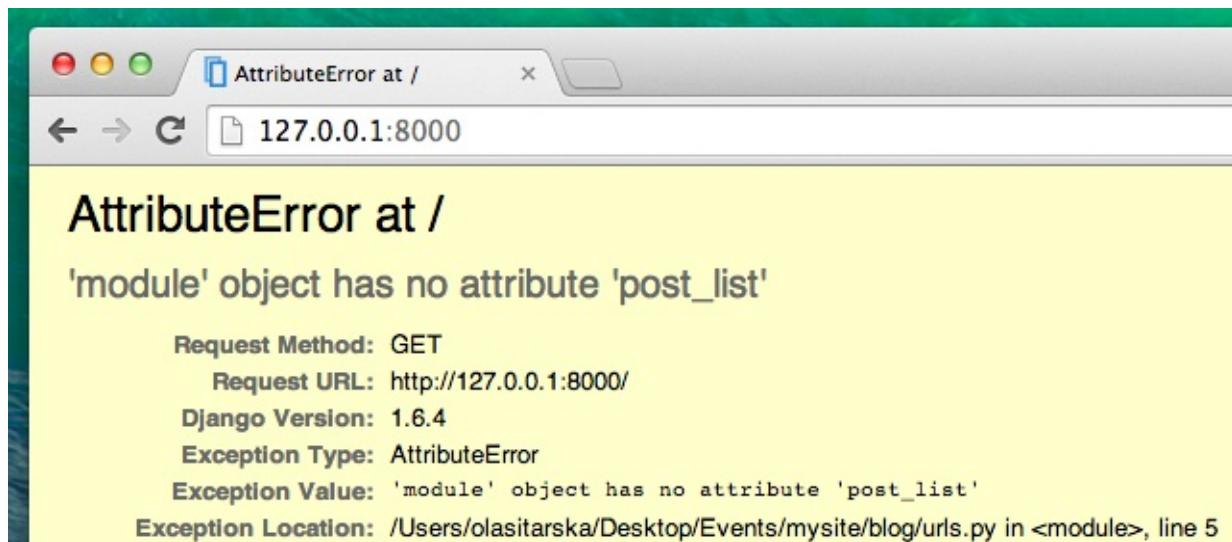
Dopo di che, possiamo aggiungere il nostro primo modello di URL:

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

Come vedi, stiamo assegnando una `view` nominata `post_list` alla URL `^$`. Questa espressione regolare combinerà `^` (un inizio) seguito da `$` (una fine) - cosicché solo una stringa vuota possa combaciare. È giusto, perché nei resolver di URL di Django, '<http://127.0.0.1:8000/>' non è una parte dell'URL. Questo schema dirà a Django che `views.post_list` è il posto giusto dove andare se qualcuno entra nel tuo sito all'indirizzo '<http://127.0.0.1:8000/>'.

L'ultima parte `name='post_list'` è il nome dell'URL che verrà usata per identificare la view. Può avere lo stesso nome della view, ma può anche essere qualcosa di completamente diverso. Useremo le URL rinominate successivamente nel progetto quindi è importante dare un nome a ciascuna URL nell'app. Inoltre dovremmo cercare di mantenere i nomi delle URL unici e facili da ricordare.

Tutto fatto? Apri <http://127.0.0.1:8000/> nel tuo browser per vedere il risultato.



Non funziona, vero? Non ti preoccupare, è solo una pagina di errore, niente di cui spaventarsi! In realtà sono molto utili:

Leggerai che **non c'è un attributo 'post_list'**. Il `post_list` ti ricorda qualcosa? Abbiamo chiamato la nostra view proprio così! Questo significa che è tutto a posto. Semplicemente non abbiamo ancora creato la nostra view. Non ti preoccupare, ci arriveremo.

Se vuoi sapere di più sulla configurazione di URL Django, vai alla documentazione ufficiale:
<https://docs.djangoproject.com/en/1.8/topics/http/urls/>

Le views di Django - è arrivata l'ora di creare!

È ora di liberarsi di quel bug che abbiamo creato nel capitolo precedente :)

Una `view` è un posto dove viene messa la "logica" della nostra applicazione. Essa richiederà informazioni dal `modello` che hai creato prima e lo passerà ad un `template`. Creeremo un template nel prossimo capitolo. Le views sono solo metodi di Python un po' più complicati di quelli che abbiamo descritto nel capitolo **Introduzione a Python**.

Le views vengono collocate nel file `views.py`. Noi aggiungeremo le nostre views nel file `blog/views.py`.

blog/views.py

OK, apriamo questo file e scopriamo cosa c'è dentro:

```
from django.shortcuts import render

# Create your views here.
```

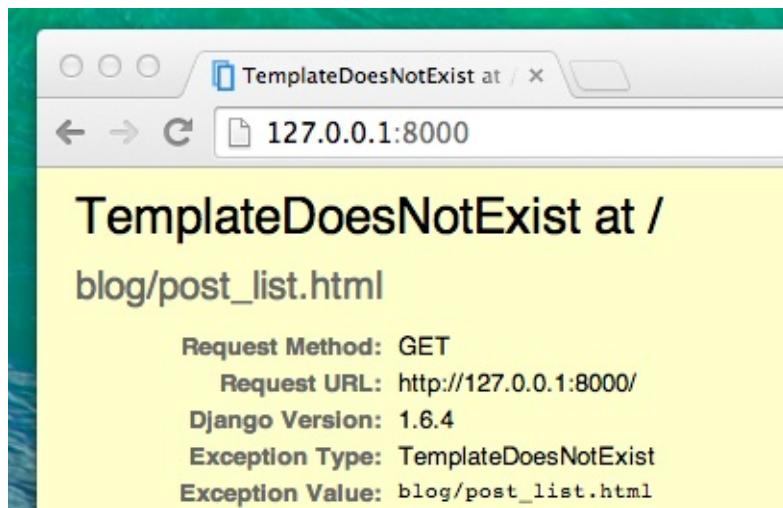
Non c'è molto per ora. La `view` più semplice può essere simile a questa.

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

Come puoi vedere, abbiamo creato un metodo (`def`) chiamato `post_list` che prende `request` e restituisce un metodo `render` che ci fornirà (metterà insieme) il nostro template `blog/post_list.html`.

Salva il file, vai su <http://127.0.0.1:8000> e guarda cosa abbiamo ottenuto.

Un altro errore! Leggi cosa sta succedendo adesso:



Questo è facile: `TemplateDoesNotExist`. Sistemiamo il bug e creiamo un template nel prossimo capitolo!

Impara di più sulle views di Django leggendo la documentazione ufficiale:

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

Introduzione all'HTML

Ti potresti chiedere, cos'è un template?

Un template è un file che possiamo riutilizzare per presentare informazioni diverse in un formato consistente - per esempio, potresti utilizzare un template per aiutarti a scrivere una lettera, perché anche se ciascuna lettera potrebbe contenere un messaggio diverso ed essere a sua volta indirizzata ad una persona diversa, condivideranno lo stesso formato.

Un template Django viene descritto in un linguaggio chiamato HTML (è lo stesso HTML che abbiamo menzionato nel primo capitolo **Come funziona l'Internet**).

Cos'è l'HTML?

HTML è un semplice codice che viene interpretato dal tuo browser - come Chrome, Firefox o Safari - per rendere un sito web visibile all'utente.

HTML sta per "HyperText Markup Language". **HyperText** significa che è un tipo di testo che supporta i collegamenti ipertestuali tra le pagine. **Markup** significa che abbiamo preso un documento e l'abbiamo contrassegnato con il codice per dire a qualcosa (in questo caso, un browser) come interpretare la pagina. Il codice HTML è costruito con **tags**, ognuno inizia con `<` e finisce con `>`. Questi tag rappresentano gli **elementi** di markup.

Il tuo primo template!

Creare un template significa creare un file template. Tutto è un file, vero? Probabilmente l'hai già notato.

I template vengono salvati in una cartella `blog/templates/blog`. Quindi prima crea una directory chiamata `templates` nella directory del tuo blog. Quindi crea un'altra directory chiamata `blog` all'interno della tua directory `templates`:

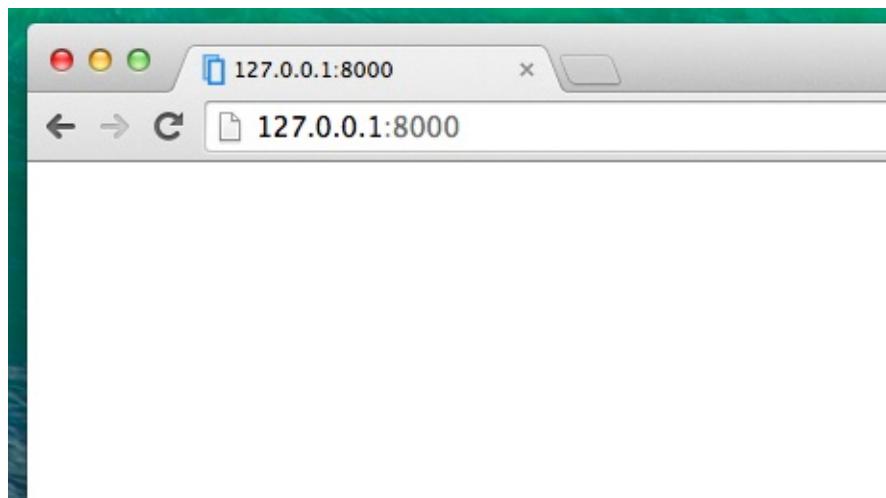
```
blog
└── templates
    └── blog
```

(Ti chiederai perché abbiamo bisogno di due directory chiamate entrambe `blog` - come scoprirai più tardi, si tratta semplicemente di una denominazione convenzionale che serve a rendere la vita più facile.)

E ora crea un file `post_list.html` nella directory `blog/templates/blog` (lascialo in bianco per adesso).

Guarda che aspetto ha il tuo sito adesso: <http://127.0.0.1:8000/>

Se continui ad avere un errore `TemplateDoesNotExist`, prova a riavviare il server. Vai nella command line, arresta il server premendo Ctrl+C (I tasti Control e C insieme) e riavvia il utilizzando il comando `python manage.py runserver`.

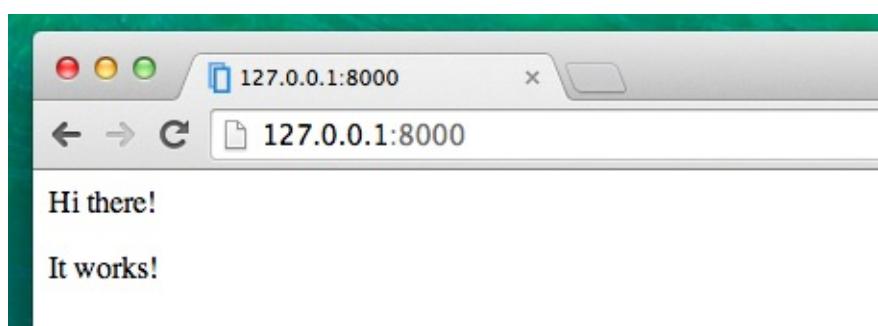


L'errore non c'è più! Congratulazioni :) Tuttavia, il tuo sito in realtà non sta pubblicando niente eccetto una pagina vuota, perché anche il tuo template è vuoto. Dobbiamo sistemarlo.

Aggiungi quanto segue nel tuo file template:

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

Quindi come appare il tuo sito ora? Clicca per scoprilo: <http://127.0.0.1:8000/>



Ha funzionato! Ottimo lavoro :)

- Il comando più basico, `<html>`, è sempre l'inizio di ogni pagina web e `</html>` è sempre la fine. Come puoi vedere, l'intero contenuto del sito va tra il tag iniziale `<html>` ed il tag conclusivo `</html>`
- `<p>` è un tag per gli elementi paragrafo; `</p>` conclude ogni paragrafo

Head & body

Ciascuna pagina HTML è a sua volta divisa in due elementi: **head** e **body**.

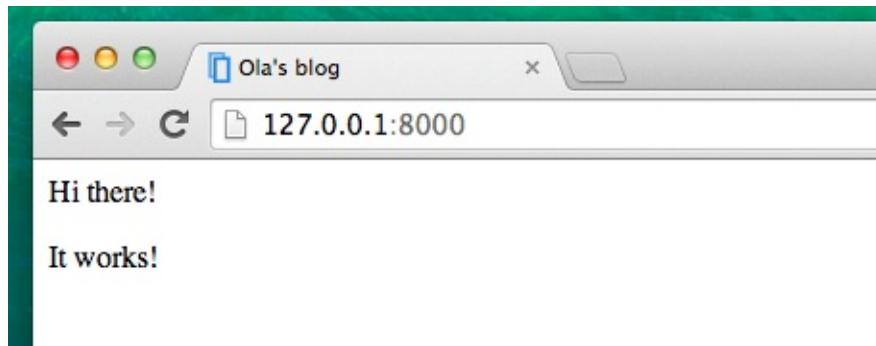
- **head** è un elemento che contiene informazioni sul documento non visibili sullo schermo.
- **body** è l'elemento che contiene tutto ciò che invece viene visualizzato come parte della pagina web.

Utilizziamo `<head>` per dire al browser come interpretare la configurazione della pagina, e `<body>` per dirgli in realtà cosa c'è nella pagina.

Per esempio, puoi mettere un elemento di titolo all'interno di `<head>`, così:

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

Salva il file e aggiorna la tua pagina.



Hai notato come il browser ha capito che "Il blog di Ola" è il titolo della tua pagina? Ha interpretato `<title>Il blog di ola</title>` ed ha messo il testo nella barra del titolo sul tuo browser (sarà anche utilizzato per i segnalibri e così via).

Probabilmente hai anche notato che ogni tag di apertura è abbinato ad un *tag di chiusura*, con un `/`, e che gli elementi sono *annidati* (i.e. non puoi chiudere un tag particolare fino a quando tutti quelli che erano al suo interno sono stati chiusi a loro volta).

È come mettere le cose in delle scatole. Hai una grossa scatola, `<html></html>`; al suo interno c'è `<body></body>`, che contiene scatole ancora più piccole: `<p></p>`.

Devi seguire queste regole di tag *di chiusura*, e di elementi *annidati* - se non lo fai, il browser potrebbe non essere in grado di interpretarli correttamente e la tua pagina verrà visualizzata incorrettamente.

Personalizza il tuo template

Ora puoi divertirti un po' e provare a personalizzare il tuo template! Qua ci sono un po' di tag utili per quello:

- `<h1>Un'intestazione</h1>` - per la tua intestazione più importante
- `<h2>Un sottotitolo</h2>` per un titolo di livello inferiore
- `<h3>Un sottotitolo più piccolo</h3>` ... e così via, fino a `<h6>`
- `text` enfatizza il tuo testo
- `text` enfatizza fortemente il tuo testo
- `
` va in un'altra riga (puoi mettere qualsiasi cosa dentro br)
- `link` crea un link
- `primo elementosecondo elemento` fa una lista, proprio come questa qui!
- `<div></div>` definisce una sezione della pagina

Qui c'è un esempio di un template completo:

```

<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 po
rta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut fermentum mass
a justo sit amet risus.</p>
    </div>

    <div>
      <p>published: 14.06.2014, 12:14</p>
      <h2><a href="">My second post</a></h2>
      <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id elit non mi po
rta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condimentum nibh, ut f.</p>
    </div>
  </body>
</html>

```

Abbiamo creato tre sezioni `div` qui.

- Il primo elemento `div` contiene il titolo del nostro blog - è un'intestazione ed un link
- Altri tre elementi `div` contengono i nostri post con la loro data di pubblicazione, `h2` con il titolo di un post che è cliccabile e due `p` (paragrafi) di testo, uno per la data e l'altro per i nostri post.

Ci dà questo effetto:



Yaaay! Ma fino adesso, il nostro template mostra esattamente **la stessa informazione** - mentre prima stavamo dicendo che i template ci permettono di mostrare **diverse** informazioni nello **stesso formato**.

Quello che vogliamo è visualizzare i veri post aggiunti nel nostro Django admin - è quello che faremo adesso.

Un'ultima cosa: il deploy!

Sarebbe bello vedere tutto questo live su Internet, giusto? Facciamo un altro deploy su PythonAnywhere:

Committa e pubblica il tuo codice su GitHub

Prima di tutto, vediamo quali file sono cambiati dall'ultimo deploy (esegui questi comandi localmente, non su PythonAnywhere):

```
$ git status
```

Assicurati di essere nella directory `djangogirls` e diciamo a `git` di includere tutte le modifiche in questa directory:

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

Nota `-A` (diminutivo di "all") significa che `git` riconoscerà anche il fatto che hai cancellato dei file (per impostazione predefinita, esso riconosce soltanto i file nuovi/modificati). Ricorda anche (dal capitolo 3) che `..` significa la directory attuale.

Prima di caricare tutti i file, proviamo a controllare cosa caricherà `git` (tutti i file che caricherà `git` ora appariranno in verde):

```
$ git status
```

Ci siamo quasi, ora è il momento di dirgli di salvare questa modifica nella cronologia. Gli daremo un "messaggio di commit" dove descriviamo ciò che abbiamo modificato. Puoi digitare tutto quello che vuoi a questo punto, sarebbe utile scrivere qualcosa di descrittivo in modo da ricordare in futuro cos'hai fatto.

```
$ git commit -m "Ho cambiato l'HTML per questo sito."
```

Nota Assicurati di usare doppie virgolette attorno al messaggio di commit.

Quando hai finito, caricheremo (push) le nostre modifiche su Github:

```
git push
```

Scarica il tuo nuovo codice su PythonAnywhere, e ricarica la tua web app

- Apri la [pagina Console PythonAnywhere](#) e vai alla tua **Bash console** (o iniziane una nuova). Quindi, esegui:

```
$ cd ~/my-first-blog
$ source myenv/bin/activate
(myenv)$ git pull
[...]
(myenv)$ python manage.py collectstatic
[...]
```

Ed osserva il tuo codice mentre viene scaricato. Se vuoi controllare che sia arrivato, puoi fare un salto alla scheda **Files** e vedere il tuo codice su PythonAnywhere.

- Infine, fai un salto alla [scheda Web](#) e premi **Reload** sulla tua web app.

Il tuo aggiornamento dovrebbe essere applicato! Vai avanti ed aggiorna il tuo sito nel brower. Le modifiche dovrebbero essere visibili :)

Django ORM e i QuerySet

In questo capitolo imparerai come Django si collega al database e archivia i dati al suo interno. Tuffiamoci!

Cos'è un QuerySet?

Un QuerySet, in sostanza, è una lista di oggetti di un determinato Modello. Il QuerySet ti permette di leggere il dato dal database, filtrarlo e ordinarlo.

È più facile impararlo con un esempio. Proviamo, ti va?

La shell di Django

Apri la tua console locale (non su PythonAnywhere) e digita questo comando:

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

L'effetto dovrebbe essere come questo:

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

Ora ti trovi nella consolle interattiva di Django. È come il prompt di python ma con un po' di magia di Django in più :). Qui puoi anche utilizzare tutti i comandi Python, ovviamente.

Tutti gli oggetti

Proviamo a rendere visibili tutti i nostri post prima. Puoi farlo con il seguente comando:

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

Ops! È comparso un errore. Ci dice che non c'è nessun Post. È corretto -- ci siamo dimenticati di importarlo!

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

È semplice: importiamo il modello `Post` da `blog.models`. Proviamo a rendere di nuovo visibili tutti i post:

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

È una lista di post che abbiamo creato prima! Abbiamo creato questi post usando l'interfaccia di amministrazione di Django. Comunque sia, ora vogliamo creare nuovi post usando Python, quindi come lo facciamo?

Creare oggetti

Così si crea un nuovo oggetto Post nel database:

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

Ma manca un ingrediente qui: `me`. Dobbiamo passare un'istanza del modello `User` come un autore. Come si fa?

Importiamo il modello User prima:

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

Quali utenti abbiamo nel nostro database? Prova questo:

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

È il superuser che abbiamo creato prima! Ora prendiamo un'istanza del user:

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

Come puoi vedere, ora prendiamo (`get`) un `User` con un `username` che è uguale a 'ola'. Ben fatto, devi cambiarlo con il tuo username.

Adesso possiamo finalmente creare il nostro post:

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

Evviva! Vuoi controllare se funziona?

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

Eccolo, un altro post nell'elenco!

Aggiungi altri post

Ora puoi divertirti un po' ed aggiungere altri post per vedere come funziona. Aggiungi altri 2 o 3 e vai alla prossima parte.

Filtrare gli oggetti

Larga parte parte dei QuerySet consiste nell'abilità di filtrarli. Diciamo che vogliamo trovare tutti i post che hanno come autore l'Utente ola. Useremo `filter` invece di `all` in `Post.objects.all()`. Tra parentesi affermeremo le condizioni che un blog post deve soddisfare per finire nel nostro queryset. Nella nostra situazione è `autore` che è uguale a `me`. Il modo di scriverlo in Django è `autore=me`. Ora il nostro pezzo di codice ha questo aspetto:

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

O magari vogliamo vedere tutti i post che contengono la parola 'titolo' nel campo `titolo`?

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

Nota ci sono due caratteri di sottolineatura (`_`) tra `titolo` e `contains`. L'ORM di Django usa questa sintassi per separare i nomi dei campi ("titolo") ed operazioni o filtri ("contiene"). Se usi solo un carattere di sottolineatura, otterrai un errore come "FieldError: non è possibile risolvere la parola chiave `title_contains`".

Puoi anche ottenere una lista di tutti i post pubblicati. Lo facciamo filtrando tutti i post che hanno una `published_date` impostata in passato:

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

Purtroppo, il post che abbiamo aggiunto dalla console Python non è ancora pubblicato. Possiamo modificarlo! In primo luogo ottenere un'istanza di un post che vogliamo pubblicare:

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

Ora pubblico con il nostro metodo `publish`!

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

Ora cerca di ottenere di nuovo l'elenco dei post pubblicati (premere il pulsante di freccia in su 3 volte e premere `invio`):

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

Ordinare gli oggetti

I QuerySet ti permettono anche di ordinare le liste di oggetti. Proviamo a ordinarli in base al campo `created_date`:

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

Possiamo anche invertire l'ordine aggiungendo `-` all'inizio:

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

QuerySet di concatenamento

Puoi anche combinare QuerySet **concatenandole** insieme:

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

È davvero potente e ti permette di scrivere query piuttosto complesse.

Fantastico! Ora sei pronta per la prossima parte! Per chiudere la shell, digita questo:

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

I dati dinamici nei templates

Abbiamo diversi pezzi: il modello `Post` è definito in `models.py`, abbiamo `post_list` nel file `views.py` ed abbiamo aggiunto il template. Ma come faremo a far comparire i nostri post nel nostro template HTML? Perché questo è quello che vogliamo: prendere qualche contenuto (modelli salvati nel database) e farlo vedere in modo carino nel nostro template, giusto?

Questo è esattamente quello che le `views` dovrebbero fare: collegare i modelli ed i template. Nella nostra `post_list view` avremo bisogno di prendere i modelli che vogliamo far vedere e passarli nel template. Quindi praticamente nella `view` decidiamo cosa (modello) renderemo visibile nel template.

OK, quindi come facciamo a farlo?

Dobbiamo aprire il nostro `blog/views.py`. Per ora `post_list view` si vede così:

```
from django.shortcuts import render

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

Ricordi quando abbiamo parlato di includere codice scritto in diversi file? Ora è il momento di includere il model che abbiamo scritto in `models.py`. Aggiungeremo questa riga `from .models import Post` così:

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

Il punto dopo il `from` significa *directory attuale* oppure *applicazione attuale*. Dal momento che `views.py` e `models.py` sono nella stessa directory possiamo semplicemente utilizzare `.` ed il nome del file (senza `.py`). Allora importiamo il nome del modello (`Post`).

Cos'altro bisogna fare? Per poter prendere i post del blog dal modello `Post` ci serve una cosa chiamata `QuerySet`.

QuerySet

Dovresti già sapere come funziona `QuerySet`. Ne abbiamo parlato nel capitolo [Django ORM \(QuerySets\)](#).

Quindi ora ci interessa una lista di post del blog che sono pubblicati e organizzati da `published_date`, giusto? Lo abbiamo già fatto nel capitolo sulle `QuerySet`!

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

Adesso mettiamo questo pezzo di codice nel file `blog/views.py` aggiungendolo alla funzione `def post_list(request):`:

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

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

Nota che abbiamo creato una *variabile* per il nostro `QuerySet`: `posts`. Vedila come il nome del nostro `QuerySet`. Da qui in avanti possiamo riferirci ad esso con questo nome.

Il codice utilizza anche la funzione `timezone.now()`, quindi dobbiamo aggiungere un import per `timezone`.

L'ultima cosa che manca è passare la QuerySet `posts` nel template (ci occuperemo di come renderlo visibile nel prossimo capitolo).

Nella funzione `render` abbiamo già un parametro con `request` (quindi tutto quello che riceviamo dal nostro utente via internet) e un file template `'blog/post_list.html'`. Nell'ultimo parametro, che è simile a questo: `{}` possiamo aggiungere cose che il template possa utilizzare. Dobbiamo dargli un nome (ci atterremo a `'posts'` per il momento :)). Si vede così: `{'posts': posts}`. Ti preghiamo di notare che la parte prima di `:` è una stringa; devi metterla tra virgolette `''`.

Il nostro file `blog/views.py` dovrà risultare così:

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

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

È tutto! Ora di tornare al nostro template e rendere visibile questo QuerySet!

Se vuoi leggere di più sui QuerySets in Django dovresti dare un'occhiata qui:

<https://docs.djangoproject.com/en/1.8/ref/models/querysets/>

I templates di Django

È l'ora di visualizzare alcuni dati! Django ci dà alcuni **template tags** già pronti per farlo.

Cosa sono i template tags?

In HTML non puoi scrivere codice Python, perché i browser non lo capiscono. Essi conoscono solo l'HTML. Noi sappiamo che l'HTML è piuttosto statico, mentre Python è molto più dinamico.

I **Django template tags** ci permettono di trasferire le cose simili a Python in HTML, in modo che tu possa costruire siti web in modo più veloce e facile. Accidenti!

Mostra il template con la lista di post

Nel capitolo precedente abbiamo dato al nostro template una lista di posts nella variabile `posts`. Adesso lo mostreremo nell'HTML.

Per stampare una variabile nel template Django, usiamo doppie parentesi graffe con il nome della variabile all'interno, così:

```
 {{ posts }}
```

Prova questo nel tuo template `blog/templates/blog/post_list.html`. Sostituisci tutto dal secondo `<div>` al terzo `</div>` con `{{ posts }}` . Salva il file e aggiorna la pagina per vedere i risultati:



Come vedi, quello che abbiamo è:

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

Significa che Django lo vede come una lista di oggetti. Ricordi dalla **Introduzione a Python** come possiamo rendere visibili le liste? Sì, con for loops! In un template Django si fanno così:

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

Prova ad inserirlo nel tuo template.

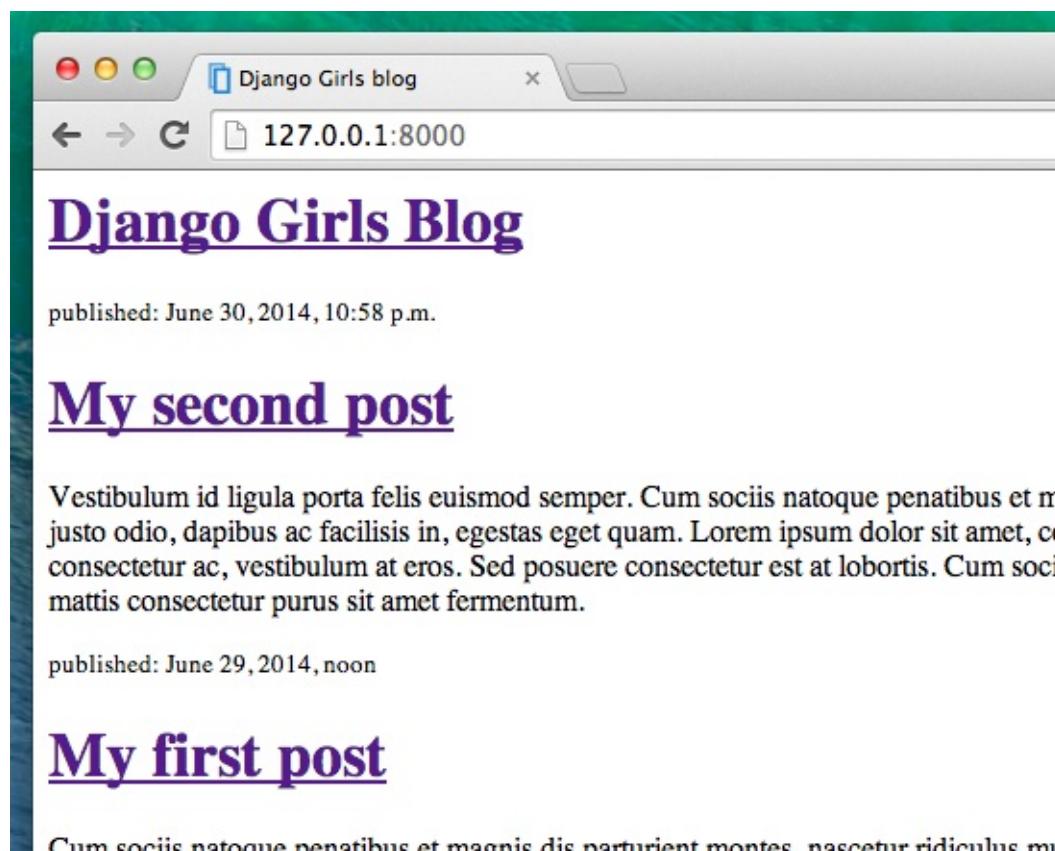


Funziona! Ma noi vogliamo che vengano mostrate come i post statici che abbiamo creato prima nel capitolo **Introduzione ad HTML**. Puoi mischiare i tag HTML con quelli di template. Il nostro `body` avrà questo aspetto:

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

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

Tutto quello che hai messo tra `{% for %}` e `{% endfor %}` Sarà ripetuto per ciascun oggetto della lista. Aggiorna la tua pagina:



Ti sei accorto che abbiamo utilizzato una notazione leggermente diversa questa volta `{{ post.title }}` oppure `{{ post.text }}`? Stiamo introducendo i dati in ciascuno dei campi definiti nel nostro modello `Post`. Inoltre le `|linebreaksbr` stanno spingendo il testo dei post attraverso un filtro per trasformare le line-breaks in paragrafi.

Un' ultima cosa

Sarebbe bello vedere se il tuo sito funziona ancora su Internet, giusto? Proviamo a fare il deploy su PythonAnywhere di nuovo. Ecco un riepilogo dei passaggi...

- Prima di tutto, fai il push del tuo codice verso Github

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

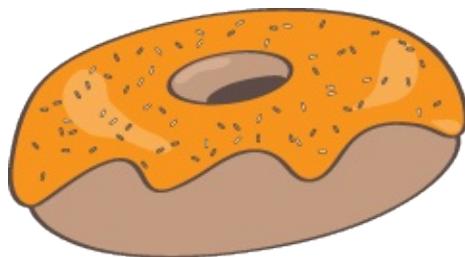
- Poi, ritorna su [PythonAnywhere](#) e vai alla tua **console di Bash** (o iniziane una nuova) ed esegui:

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

- Infine, vai sulla [Web tab](#) e premi **Reload** sulla tua web app. L'aggiornamento dovrebbe essere live!

Congratulazioni! Ora vai avanti e prova ad aggiungere un nuovo post nel tuo Admin Django (ricorda di aggiungere una `published_date`!), sucessivamente aggiorna il tuo sito per vedere se il post compare.

Funziona come un incantesimo? Ne siamo fieri! Staccati dal computer per un po', ti sei guadagnato/a una pausa. :)



CSS - dagli un bel aspetto!

Il nostro blog sembra ancora un pochino brutto, vero? È arrivato il momento di abbellirlo! Per fare ciò useremo CSS.

Che cos'è CSS?

Cascading Style Sheets (CSS) è un linguaggio usato per descrivere l'aspetto e la formattazione di un sito scritto in un linguaggio di markup (come HTML). Vedilo come il trucco del nostro sito ;).

Ma non vogliamo ricominciare da capo, giusto? Ancora una volta useremo qualcosa preparato da altri programmati disponibile su Internet gratuitamente per tutti. Si sa, re-inventare la ruota non è molto divertente.

Usiamo Bootstrap!

Bootstrap è uno dei più popolari framework HTML e CSS per costruire bellissimi siti internet: <https://getbootstrap.com/>
È stato scritto da programmati che lavoravano per Twitter ed è ora sviluppato da volontari da ogni parte del mondo.

Installa Bootstrap

Per installare Bootstrap, avrai bisogno di aggiungere questo nel tag `<head>` del tuo file `.html`
(`blog/templates/blog/post_list.html`):

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

Le due linee sopra riportate non aggiungono nessun file al tuo progetto. Stanno solo reindirizzando ad alcuni files che esistono in internet. Ma non ti preoccupare troppo, vai avanti e apri la tua pagina web e ricarica la pagina. Ecco qui!



File statici in Django

Infine, daremo uno sguardo più approfondito a quelli che abbiamo chiamato **file statici**. I file statici sono tutti i tuoi CSS e le immagini -- file che non sono dinamici, il loro contenuto non dipende dal contesto della richiesta e sarà lo stesso per ogni utente.

Dove mettere i file statici in Django

Come hai visto quando abbiamo eseguito `collectstatic` sul server, Django sa già dove trovare i file statici per l'app built-in "admin". Ora dobbiamo solo aggiungere alcuni file statici per la nostra app, `blog`.

Lo facciamo creando una cartella denominata `static` all'interno della nostra app `blog`:

```
djangogirls
└── blog
    ├── migrations
    └── static
└── mysite
```

Django troverà automaticamente tutte le cartelle chiamate "static" dentro le cartelle delle tua app, e sarà in grado di utilizzare il loro contenuto come file statici.

Il tuo primo file CSS!

Ora, creiamo un file CSS, per poter aggiungere il tuo stile personale al tuo sito. Crea una nuova cartella dal nome `css` all'interno della tua cartella `static`. Poi, crea un nuovo file all'interno della tua cartella `css` e chiamalo `blog.css`. Fatto?

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

È giunto il momento di scrivere un po' di CSS! Apri il file `static/css/blog.css` nel tuo editor di codice.

Non ci dilungheremo troppo sul come personalizzare e imparare CSS in questo momento, dal momento che è abbastanza facile e puoi impararlo da solo/a alla fine di questo workshop. Ti raccomandiamo caldamente di seguire questo corso [Codecademy HTML & CSS course](#) per imparare tutto quello che serve sapere per poter rendere più bello un sito internet con CSS.

Facciamo comunque un esempio. Perchè non cambiare il colore del nostro header? Per decifrare i colori, i computer usano dei codici speciali. Questi codici cominciano con `#` a cui fanno seguito 6 caratteri, sia lettere (A-F) che numeri (0-9). Puoi trovare vari esempi di codici colore qui: <http://www.colorpicker.com/>. Puoi anche usare [colori predefiniti](#) come ad esempio `red` e `green`.

Aggiungi il seguente codice nel tuo file `static/css/blog.css`:

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

`h1 a` è un esempio di selettori CSS. Questo significa che stiamo cercando di cambiare lo stile su tutti gli elementi `a` all'interno di un elemento `h1` (in questo caso, abbiamo una linea di codice così strutturata: `<h1>link</h1>`). In questo caso, stiamo cercando di cambiare il colore con `#FCA205`, che corrisponde all'arancione. Ovviamente puoi mettere il codice di qualsiasi altro colore tu preferisca!

In un file CSS definiamo lo stile degli elementi presenti nel file HTML. Gli elementi in questione vengono identificati con il nome (ad esempio `a`, `h1`, `body`) oppure con l'attributo `class` o l'attributo `id`. Class e id sono i nomi che assegni agli elementi. Le classi definiscono gruppi di elementi mentre gli id indicano uno specifico elemento. Ad esempio, il seguente

elemento può essere identificato nel CSS utilizzando il nome del tag `a`, la classe `external_link` oppure l'id `link_to_wiki_page`:

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

Per saperne di più puoi leggere [CSS Selectors in w3schools](#).

Infine, dobbiamo anche far sapere al nostro template in HTML che abbiamo effettivamente aggiunto un po' di CSS. Apri il file `blog/templates/blog/post_list.html` e aggiungi la seguente riga di testo:

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

Per ora stiamo solamente caricando tutti i nostri static files :). Aggiungi questa riga di testo tra `<head>` e `</head>`, subito dopo il link al file CSS di Bootstrap (il browser legge i file nell'ordine in cui sono dati, per cui il codice nei nostri files può sovrascrivere il codice presente nei files di Bootstrap):

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

Stiamo dicendo al nostro template dove trovare i nostri file CSS.

Il tuo file dovrebbe avere questo aspetto:

```
{% load staticfiles %}

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

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

OK, salviamo il file e ricarichiamo la pagina web!

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

Ben fatto! Adesso potremmo dare un po' più d'aria alla nostra pagina web e aumentare il margine nella parte sinistra. Proviamo!

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

Aggiungi questo al tuo CSS, salva il file e guarda il risultato!

The screenshot shows the same browser window after applying the CSS rule. The left margin for the main content area has been increased, creating more space on the left side of the page.

Potremmo anche personalizzare lo stile calligrafico nel nostro header. Incolla quanto segue all'interno del tag `<head>` che si trova nel file `blog/templates/blog/post_list.html`:

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

Questa riga consente di importare un font chiamato *Lobster* da Google Fonts (<https://www.google.com/fonts>).

Ora aggiungi `font-family: 'Lobster';` nel file CSS `static/css/blog.css` all'interno del blocco `h1 a` (il codice tra le due parentesi graffe, `{ e }`) e ricarica la pagina:

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



Grandioso!

Come già accennato, il CSS utilizza il concetto di 'classe' che in pratica ti permette di assegnare uno specifico nome ad una parte del tuo documento HTML e di applicare uno stile solo a questa parte senza cambiare il resto. È di grande aiuto quando hai due div che hanno funzioni differenti (ad esempio uno è un header e l'altro un post), e non vuoi che appaiano uguali.

Prova a dare dei nomi ad alcune parti dell'HTML. Aggiungi una classe chiamata `page-header` al tuo `div` che contiene l'intestazione così:

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

E ora aggiungi una classe `post` al tuo `div` che contiene un articolo del blog.

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

Ora aggiungiamo dei blocchi di codice ai nostri nuovi selettori. I selettori che iniziano con `.` indicano una classe. Online ci sono molti tutorial e spiegazioni sul CSS che possono aiutarti a comprendere il codice che stiamo per scrivere. Per ora, copia e incolla quanto segue nel tuo file `mysite/static/css/blog.css`:

```
.page-header {  
    background-color: #ff9400;  
    margin-top: 0;  
    padding: 20px 20px 20px 40px;  
}  
  
.page-header h1, .page-header h1 a, .page-header h1 a:visited, .page-header h1 a:active {  
    color: #ffffff;  
    font-size: 36pt;  
    text-decoration: none;  
}  
  
.content {  
    margin-left: 40px;  
}  
  
h1, h2, h3, h4 {  
    font-family: 'Lobster', cursive;  
}  
  
.date {  
    float: right;  
    color: #828282;  
}  
  
.save {  
    float: right;  
}  
  
.post-form textarea, .post-form input {  
    width: 100%;  
}  
  
.top-menu, .top-menu:hover, .top-menu:visited {  
    color: #ffffff;  
    float: right;  
    font-size: 26pt;  
    margin-right: 20px;  
}  
  
.post {  
    margin-bottom: 70px;  
}  
  
.post h1 a, .post h1 a:visited {  
    color: #000000;  
}
```

Ora aggiungi all'esterno del codice HTML riguardante i posts all'interno del blog alcuni elementi con definizione di classi. Sostitisci questo:

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

nel file `blog/templates/blog/post_list.html` con quanto segue:

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

Salva entrambi i file e ricarica la pagina web.



Woohoo! È fantastico, vero? Il codice che abbiamo appena inserito non è poi così difficile da comprendere, dovresti riuscire a capirne la maggior parte semplicemente leggendolo.

Non farti spaventare, sperimenta con i CSS e prova a cambiare alcune cose. Se rompi qualcosa, non ti preoccupare, puoi sempre farlo tornare come era prima!

Compito per casa post-workshop: ti consigliamo caldamente di seguire [il corso su HTML & CSS di Codecademy](#). Così potrai imparare tutto ciò di cui hai bisogno per rendere i tuoi siti web più belli sfruttando il CSS.

Pronta per il prossimo capitolo?! :)

Estendere il template

Un'altra cosa bella di Django è l'**estensione del template**. Cosa significa? Significa che puoi usare le stesse parti del tuo HTML per pagine diverse del tuo sito.

Così non hai bisogno di ripetere le stesse informazioni/layout in ogni file. E se vuoi cambiare qualcosa, non devi cambiarlo in ogni templates, ma soltanto una volta!

Crea un template di base

Un template base è il template più semplice. Lo puoi estendere su ogni pagina del tuo sito.

Creiamo un file `base.html` in `blog/templates/blog/`:

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

Poi aprilo e copia tutto da `post_list.html` e incollalo sul file `base.html`, così:

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

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

Poi nel `base.html`, rimpiazza tutto il tuo `<body>` (tutto quello che si trova tra `<body>` e `</body>`) con questo:

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

Abbiamo praticamente rimpiazzato tutto quello tra `{% for post in posts %}{% endfor %}` con:

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

Che cosa significa? Che hai appena creato un `blocco`, ovvero un tag di template che ti permette di inserire l'HTML presente in questo blocco all'interno di altri template che estendono `base.html`. Ti mostreremo come farlo tra un attimo.

Ora salvalo, e apri il tuo `blog/templates/blog/post_list.html` di nuovo. Cancella tutto quello che non è all'interno del body e poi cancella anche `<div class="page-header"></div>`, in modo che il file appaia così:

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

Ed ora aggiungi questa riga all'inizio del file:

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

Significa che stiamo estendendo il template `base.html` in `post_list.html`. Rimane solo una cosa da fare: metti tutto (tranne la riga che abbiamo appena aggiunto) tra `{% block content %}` e `{% endblock content %}`. Come questo:

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

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

È tutto! Controlla se il tuo sito sta ancora funzionando correttamente :)

Se hai un errore `TemplateDoesNotExist` che dice che non c'è un file `blog/base.html` e hai `runserver` in esecuzione nella console, prova a fermarlo (premendo Ctrl+C - I tasti Control e C insieme) e riavvia mettendo in esecuzione il comando `python manage.py runserver`.

Estendi la tua applicazione

Abbiamo completato i passi necessari per la creazione del nostro sito: sappiamo come scrivere un modello, una url, una view ed un template. Sappiamo anche come far diventare carino il nostro sito.

Ora di far pratica!

La prima cosa di cui abbiamo bisogno nel nostro blog è, ovviamente, una pagina per rendere visibile un post, vero?

Abbiamo già un modello dei `Post`, quindi non abbiamo bisogno di aggiungere niente in `models.py`.

Creare un link di template verso la pagina di dettaglio di un post

Cominceremo aggiungendo un link all'interno del file `blog/templates/blog/post_list.html`. Per ora dovrebbe avere questo aspetto:

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

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

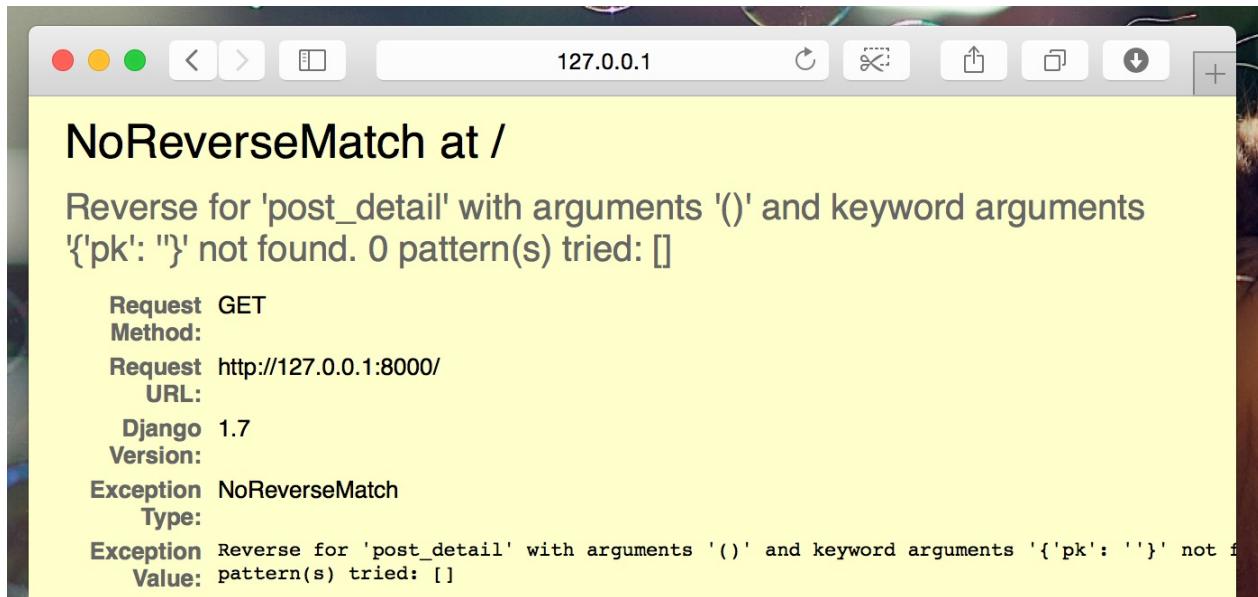
Vogliamo creare un link che dal titolo di un post facente parte dell'elenco di articoli porti alla pagina di dettaglio. Cambiamo `<h1>{{ post.title }}</h1>` così che linki alla pagina di dettaglio del post:

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

È arrivata l'ora di spiegare il misterioso `{% url 'post_detail' pk=post.pk %}`. Come avrai capito, il simbolo `{% %}` significa che stiamo usando i tag del template di Django. Questa volta ne useremo uno che creerà una URL per noi!

`blog.views.post_detail` è un percorso per arrivare alla `post_detail` view che vogliamo creare. Nota bene: `blog` è il nome della nostra applicazione (la directory `blog`), `views` viene dal nome del file `views.py` e l'ultima cosa - `post_detail` - è il nome della `view`.

Adesso quando andremo all'indirizzo: <http://127.0.0.1:8000/> avremo un errore (come sapevamo, dal momento che non abbiamo una URL oppure una `view` per `post_detail`). Avrà questo aspetto:



Crea una URL per i dettagli di un post

Creiamo una URL in `urls.py` per il nostro `post_detail` view!

Vogliamo che il nostro primo post venga visualizzato a questo URL : <http://127.0.0.1:8000/post/1/>

Facciamo sì che l'URL nel file `blog/urls.py` punti Django ad una view chiamata `post_detail`, che mostrerà un intero post. Aggiungi la riga `url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail, name='post_detail')`, al file `blog/urls.py`. Il file dovrebbe assomigliare a questo:

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

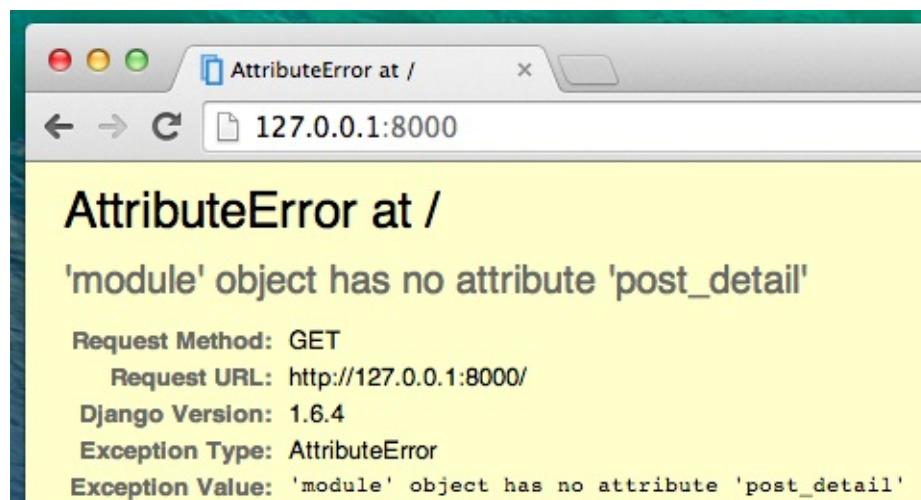
urlpatterns = [
    url(r'^$', views.post_list, name='post_list'),
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail, name='post_detail'),
]
```

Questa parte `^post/(?P<pk>[0-9]+)/$` sembra spaventosa, ma non preoccuparti - te la spiegheremo: - inizia ancora con `^` -- "l'inizio" - `post/` semplicemente significa che dopo l'inizio, l'URL dovrebbe contenere la parola **post** e `/`. Fin qui tutto bene. - `(?P<pk>[0-9]+)` - questa parte è più complicata. Significa che Django prenderà tutto quello che hai messo qui e lo trasferirà ad una view come variabile denominata `pk`. `[0-9]` ci dice anche che la variabile può essere solo un numero, non una lettera (quindi tutto tra 0 e 9). `+` significa che ci devono essere una o più cifre. Quindi qualcosa di simile a `http://127.0.0.1:8000/post//` non è valido, ma `http://127.0.0.1:8000/post/1234567890/` è perfetto! - `/` Quindi ci serve `/` di nuovo - `$` - "fine"!

Ciò significa che se digiti `http://127.0.0.1:8000/post/5/` nel tuo browser, Django capirà che stai cercando una view chiamata `post_detail` e trasferirà l'informazione che `pk` è uguale a `5` a quella view.

`pk` è un diminutivo di `primary key`. Questo nome viene frequentemente utilizzato nei progetti Django. Ma puoi chiamare la tua variabile come vuoi (ricorda: minuscole e `_` invece degli spazi!). Per esempio invece di `(?P<pk>[0-9]+)` potremmo avere la variabile `post_id` quindi questo pezzettino dovrebbe assomigliare a: `(?P<post_id>[0-9]+)`.

Ok, abbiamo aggiunto un nuovo schema di URL a `blog/urls.py` ! Aggiorniamo la pagina: <http://127.0.0.1:8000/> Boom! Ancora un altro errore! Come previsto!



Ti ricordi di quale è il prossimo passo? Ma certo: aggiungere una view!

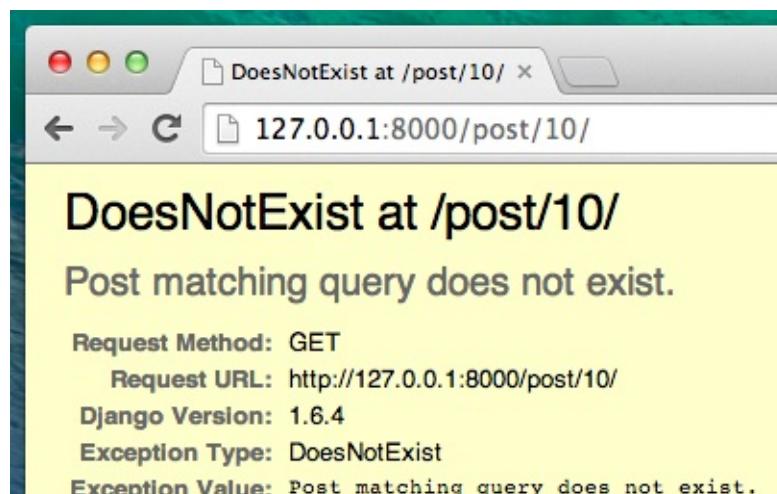
Aggiungi una view del post

Questa volta alla nostra view viene data un altro parametro `pk`. La nostra view deve prenderlo, vero? Quindi definiremo la nostra funzione come `def post_detail(request, pk):`. Dobbiamo utilizzare esattamente lo stesso nome che abbiamo specificato in urls (`pk`). Omettere questa variabile è sbagliato e genererà un errore!

Ora, noi vogliamo ottenere un unico post. Per farlo possiamo utilizzare le queryset così:

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

Ma questo codice presenta un problema. Se non c'è `Post` con `primary key` (`pk`) otterremo un errore bruttissimo!



Noi non lo vogliamo! Ma, senza dubbio, Django ha qualcosa che si occuperà del problema per noi: `get_object_or_404`. Nel caso in cui non ci sia `Post` con la data `pk` mostrerà una pagina molto più carina (chiamata `Page Not Found 404`).



La buona notizia è che in realtà puoi creare la tua pagina `Page not found` modificarla come vuoi e darle un bell'aspetto. Ma non è importantissimo in questo momento, quindi salteremo questa parte.

Ok, è arrivata l'ora di aggiungere una *view* al nostro file `views.py`!

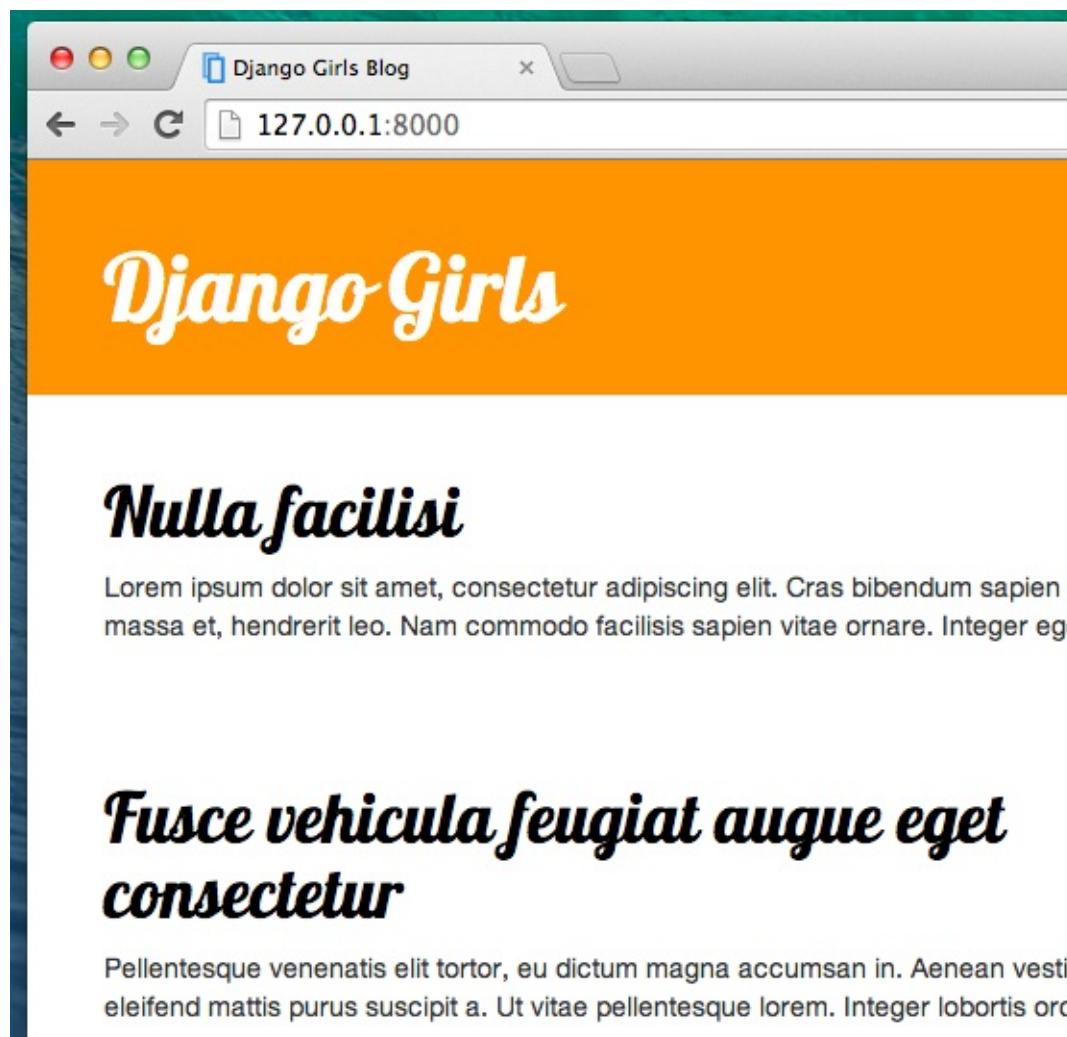
Dovremo aprire `blog/views.py` ed aggiungere il seguente codice:

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

Vicino ad altre righe `from` ed alla fine del file aggiungeremo la nostra *view*:

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

Si, è giunta l'ora di aggiornare la pagina: <http://127.0.0.1:8000/>



Django Girls

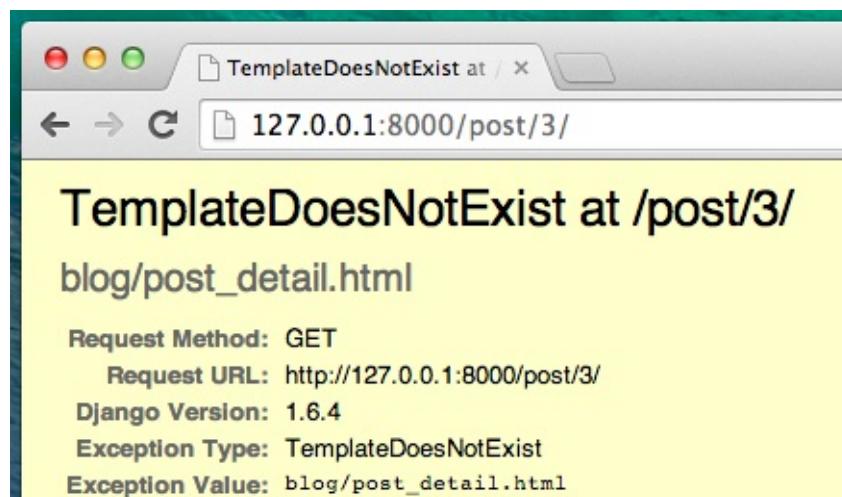
Nulla facilisi

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras bibendum sapien i
massa et, hendrerit leo. Nam commodo facilisis sapien vitae ornare. Integer ege

Fusce vehicula feugiat augue eget consectetur

Pellentesque venenatis elit tortor, eu dictum magna accumsan in. Aenean vestit
eleifend mattis purus suscipit a. Ut vitae pellentesque lorem. Integer lobortis orci

Ha funzionato! Ma cosa succede se clicchi su un link nel titolo del post?



TemplateDoesNotExist at /post/3/

blog/post_detail.html

Request Method: GET
Request URL: http://127.0.0.1:8000/post/3/
Django Version: 1.6.4
Exception Type: TemplateDoesNotExist
Exception Value: blog/post_detail.html

Oh no! Un altro errore! Ma sappiamo già come occuparcene, giusto? Dobbiamo aggiungere un template!

Crea un template per il post detail

Creeremo un file in `blog/templates/blog` chiamato `post_detail.html`.

Il risultato somiglierà a questo:

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

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

Stiamo estendendo ancora una volta il template di base. `base.html`. Nel blocco `content` vogliamo mostrare una `published_date` del post (se esiste), un titolo ed il testo. Ma dovremmo discutere di alcune cose importanti, vero?

`{% if ... %} ... {% endif %}` è un tag di template che possiamo utilizzare quando vogliamo controllare qualcosa (ricordi `if ... else ..` dal capitolo **Introduzione a Python?**). In questo caso vogliamo controllare che la `published_date` di un post non sia vuota.

Ok, possiamo aggiornare la nostra pagina e vedere se `Page not found` non c'è più.



Si! Ha funzionato!

Ultima cosa: ora di fare il deploy!

Sarebbe bello vedere se il tuo sito Web sarà ancora funzionante in PythonAnywhere, vero? Proviamo a fare un altro deploy.

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

- Poi, in una [console PythonAnywhere Bash](#):

```
$ cd my-first-blog  
$ source myvenv/bin/activate  
(myenv)$ git pull  
[...]  
(myenv)$ python manage.py collectstatic  
[...]
```

- Infine, vai su il [Web tab](#) e premi **Reload**.

Fatto! Congratulazioni :)

I form di Django

Infine vogliamo creare un bel modo per poter aggiungere e cambiare in nostri blog posts. Django `admin` è bello, ma è alquanto difficile da personalizzare e rendere carino. Con i `forms` avremo il potere assoluto sull'aspetto della nostra pagina web - possiamo fare praticamente qualsiasi cosa vogliamo!

La bella cosa dei Django forms è che possiamo sia inventare un nuovo form da zero che creare un `ModelForm` che salverà il risultato del form sul nostro modello.

Questo è esattamente quello che stiamo per fare: stiamo per creare un form per il nostro modello dei `Post`.

Come ogni parte importante di Django, i forms hanno il proprio file: `forms.py`.

Dobbiamo creare un file con questo nome all'interno della cartella `blog`.

```
blog
└── forms.py
```

OK, apriamo questo file e inseriamo:

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

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

Dobbiamo importare prima di tutto i Django Forms (`from django import forms`) e, ovviamente, il nostro `Post` model (`from .models import Post`).

`PostForm`, come probabilmente hai intuito, è il nome del nostro form. Dobbiamo ora dire a Django che questa form è una `ModelForm` (così Django farà qualche magia per noi) - `forms.ModelForm` è il comando per farlo.

Successivamente, abbiamo `class Meta`, con cui diciamo a Django quale model utilizzare per creare questo form (`model = Post`).

Finalmente possiamo indicare uno o più campi che il nostro form deve avere. In questo caso vogliamo che solamente `title` e `text` siano visibili - `author` è la persona attualmente connessa (tu!) e `created_date` dovrebbe generarsi da sola ogni volta che creiamo un post (cioè nel nostro programma), giusto?

E questo è tutto! Tutto quello che dobbiamo fare ora è usare il form nella nostra `view` e visualizzarlo nel template.

Ricapitolando, creeremo: link che punti alla pagina, una URL, una view e un model.

Link ad una pagina usando il form

È tempo di aprire `blog/templates/blog/base.html`. Aggiungeremo un link nel `div` chiamato `page-header`:

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

Nota che vogliamo chiamare la nostra nuova view `post_new`.

Dopo aver aggiunto quanto detto, il tuo file html dovrebbe essere simile a questo:

```
{% load staticfiles %}

<html>
  <head>
    <title>Django Girls blog</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
    <link href='//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext' rel='stylesheet' type='text/css'>
  >
    <link rel="stylesheet" href="{% static 'css/blog.css' %}">
  </head>
  <body>
    <div class="page-header">
      <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
      <h1><a href="/">Django Girls Blog</a></h1>
    </div>
    <div class="content container">
      <div class="row">
        <div class="col-md-8">
          {% block content %}
          {% endblock %}
        </div>
      </div>
    </div>
  </body>
</html>
```

Dopo aver salvato e aggiornato la pagina <http://127.0.0.1:8000> vedrai ovviamente un errore familiare `NoReverseMatch`, giusto?

URL

Apri il file `blog/urls.py` e aggiungi:

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

Il risultato finale sarà:

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

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

Dopo aver aggiornato il sito, vedremo un `AttributeError` dal momento che non abbiamo ancora creato `post_new`. Aggiungiamolo adesso.

post_new view

Apri il file `blog/views.py` e aggiungi quanto segue con il resto delle importazioni `from`:

```
from .forms import PostForm
```

e la nostra `view`:

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

Per creare un nuovo `Post` form, dobbiamo chiamare il metodo `PostForm()` e passarlo nel nostro template. Torneremo poi sulla view, ma per ora creiamo un veloce template per il nostro form.

Template

All'interno della cartella `blog/templates/blog` dobbiamo creare il file `post_edit.html`. Per far sì che il nostro form funzioni abbiamo bisogno di diverse cose:

- dobbiamo rendere il form visibile. Per farlo possiamo usare semplicemente `{{ form.as_p }}`.
- le righe scritte sopra hanno bisogno di 'essere avvolte' da un HTML tag: `<form method="POST">...</form>`
- ci serve un `Save` pulsante. Possiamo fare ciò con HTML button: `<button type="submit">Save</button>`
- infine, subito dopo l'apertura del tag `<form ...>`, dobbiamo aggiungere `{% csrf_token %}`. Questo passaggio è molto importante dal momento che rende il nostro form sicuro! Django si lamenterebbe se ti dimentichi di inserire questa parte e provi comunque a salvare ciò che è contenuto nel form:

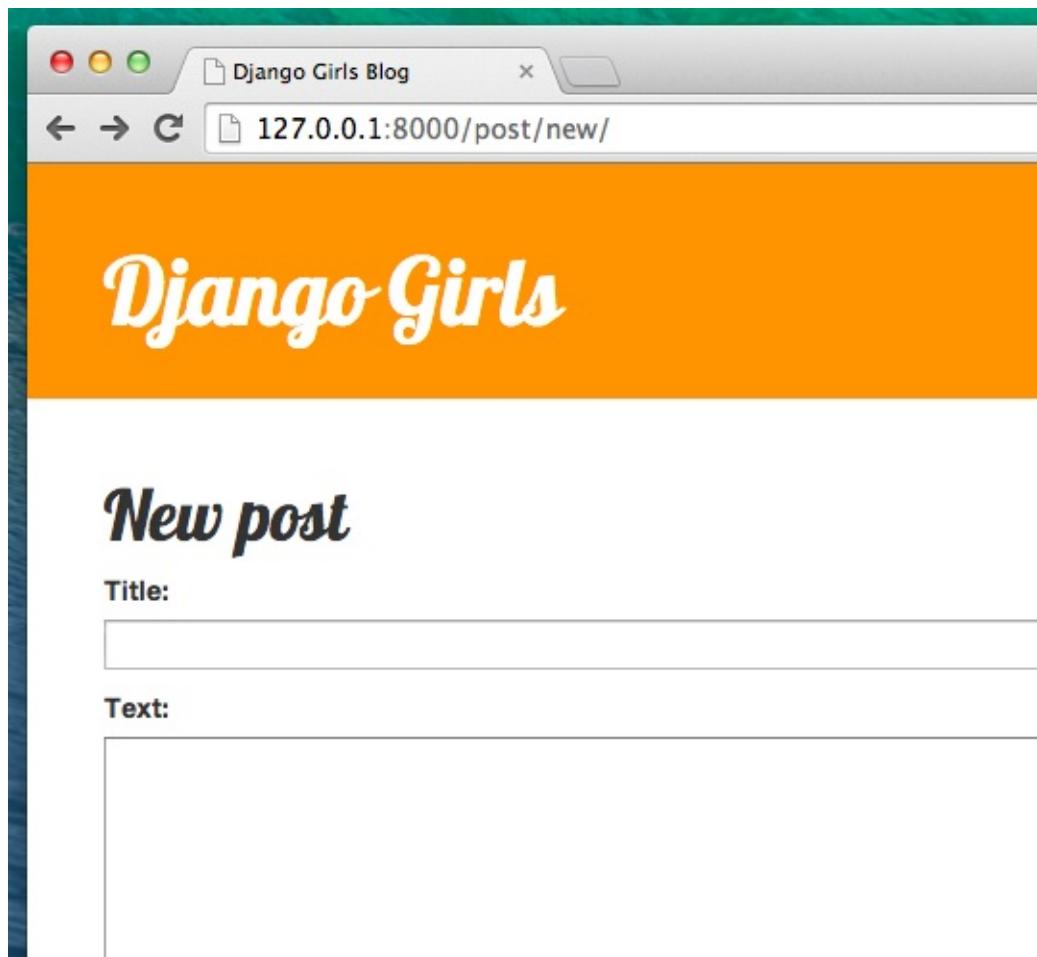


OK, il tuo HTML `post_edit.html` dovrebbe apparire così:

```
{% 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 %}
```

Ora aggiorna la pagina! Yeah! Puoi ora visualizzare il tuo form!



Ma, aspetta un momento! Se provi a scrivere qualcosa in `title` e `text` e cerchi di salvare ciò che hai scritto, che cosa succede?

Niente! Siamo di nuovo sulla stessa pagina di prima e il nostro testo è sparito...e non compare nessun nuovo post. Che cosa abbiamo sbagliato?

La risposta è: nulla. Dobbiamo solo fare un po' di lavoro in più nella nostra `view`.

Salvare il form

Apri `blog/views.py` di nuovo. Attualmente tutto ciò che abbiamo nella view `post_new` è:

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

Quando inviamo il form, veniamo riportati alla stessa view, ma questa volta abbiamo più dati in `request`, in particolare in `request.POST` (il nome non ha nulla a che vedere con un blog "post", bensì con l'inglese "posting", ovvero inviare, in questo caso dati). Ti ricordi che nel nostro file HTML il nostro `<form>` aveva la variabile `method="POST"`? Per cui ora, tutto quello che l'utente ha inserito nel form è disponibile in `method="POST"`. Non è necessario rinominare `POST` in nessuna altra maniera (l'unico altro valore valido per `method` è `GET`, ma al momento non abbiamo abbastanza tempo per spiegare la differenza).

Per cui nella nostra `view` abbiamo due diverse situazioni da gestire. Prima: quando accediamo alla pagina per la prima volta e vogliamo un form che sia vuoto. Seconda: quando torniamo alla `view` con tutti i dati appena inseriti nel form. Per cui dobbiamo aggiungere una condizione(useremo `if`).

```

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

```

È ora di completare i puntini [...] . Se il `method` è `POST` allora vogliamo costruire il nostro `PostForm` con i dati appena inseriti dall'utente, giusto? Raggiungeremo questo risultato con:

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

Facile! Come prossima cosa dobbiamo controllare se il form è corretto (per cui che tutti i campi necessari siano stati impostati e che non ci siano valori sbagliati). Possiamo fare questo con `form.is_valid()` .

Se il form viene ritenuto valido verrà salvato!

```

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

```

In pratica, ci sono due cose da fare: salviamo il form con `form.save` e aggiungiamo un autore (dal momento che non c'era nessun campo autore `author` nel form `PostForm` e questo campo non può essere lasciato bianco!). `commit=False` significa che non vogliamo salvare `Post` model per il momento-vogliamo prima assicurarc di aggiungere un autore. Per la maggior parte del tempo userai `form.save()`, senza `commit=False`, ma in questo caso abbiamo bisogno di questa extra specificazione. `post.save()` salverà le modifiche (aggiunta di autore) e il nuovo post del tuo blog è stato finalmente creato!

Infine, non sarebbe fantastico se potessimo immediatamente essere indirizzati alla pagina `post_detail` del nuovo blog post appena creato? Per fare ciò dobbiamo importare:

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

Aggiungilo all'inizio del file. E ora possiamo dire: vai alla pagina `post_detail` per il post appena creato.

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

`blog.views.post_detail` è il nome della view che vogliamo visitare. Ti ricordi che questa view ha bisogno della variabile `pk`? Per passarla alla nostra views utilizziamo `pk=post.pk`, dove `post` è il post appena creato!

Ok,abbiamo parlato abbastanza ora e forse sei curioso/a di vedere l'aspetto della nostra view, giusto?

```

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

```

Vediamo se funziona. Vai sulla pagina <http://127.0.0.1:8000/post/new/>, aggiungi un `title` e un `text`, salvalo... e voilà! Il tuo nuovo post è stato aggiunto e noi siamo stati reindirizzati automaticamente alla pagina `post_detail` !

Forse avrai notato che stiamo impostando una data di pubblicazione prima di salvare il post. Più tardi introdurremo l'uso del *publish button* in **Django Girls Tutorial: Extensions**.

Fantastico!

Validazione del Form

E adesso ti dimostreremo quanto siano belli i form di Django. Sappiamo che un post ha bisogno di `title` e `text`. Nel nostro `Post` model non abbiamo detto (al contrario di quello che abbiamo fatto per `published_date`) che questi campi non sono obbligatori, per cui Django si aspetta che vengano impostati.

Prova a salvare il form senza `title` e `text`. Indovina che cosa accade!

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

Django si sta prendendo cura di controllare che tutti i campi nel nostro form siano corretti. Non è fantastico?

Dato che poco fa stavamo usando l'interfaccia di Django admin, Django pensa che siamo ancora connessi. Ci sono alcune situazioni che potrebbero portarci a fare un log out(chiudere il browser, riavviare il DB etc.). Se ti trovi nella situazione di avere errori, quando crei un post, relativi alla mancanza di un utente, vai alla admin page <http://127.0.0.1:8000/admin> ed effettua il log in di nuovo. Questo risolverà temporaneamente il problema. C'è una correzione permanente che ti aspetta nella sezione degli esercizi extra alla fine del tutorial principale **Compiti: aggiungi sicurezza al tuo sito web!**.

Modifica il form

Ora sappiamo come aggiungere un form. Ma cosa succede se ne vogliamo cambiare uno esistente? È un processo abbastanza simile a quelli che abbiamo appena fatto. Creiamo alcune cose importanti rapidamente (se non capisci qualcosa, chiedi aiuto al tuo coach o guarda i capitoli precedenti, dal momento che abbiamo coperto tutti questi passaggi precedentemente).

Apri `blog/templates/blog/post_detail.html` e aggiungi:

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

per cui il tuo template sarà così:

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

{% block content %}


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


```

In `blog/urls.py` aggiungi:

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

Riutilizzeremo il model `blog/templates/blog/post_edit.html`, quindi l'ultima cosa che manca è una view.

Apriamo `blog/views.py` e aggiungiamo alla fine del 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.published_date = timezone.now()
            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})
```

Questo sembra quasi esattamente la view `post_new`, giusto? Ma non del tutto. Prima cosa: si passa un parametro supplementare `pk` dall'URL. Dopo di che: prendiamo il modello `Post` che vogliamo modificare con `get_object_or_404(Post, pk=pk)` e in seguito, quando creeremo un form, passeremo questo post come `instance`, sia quando salviamo il form:

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

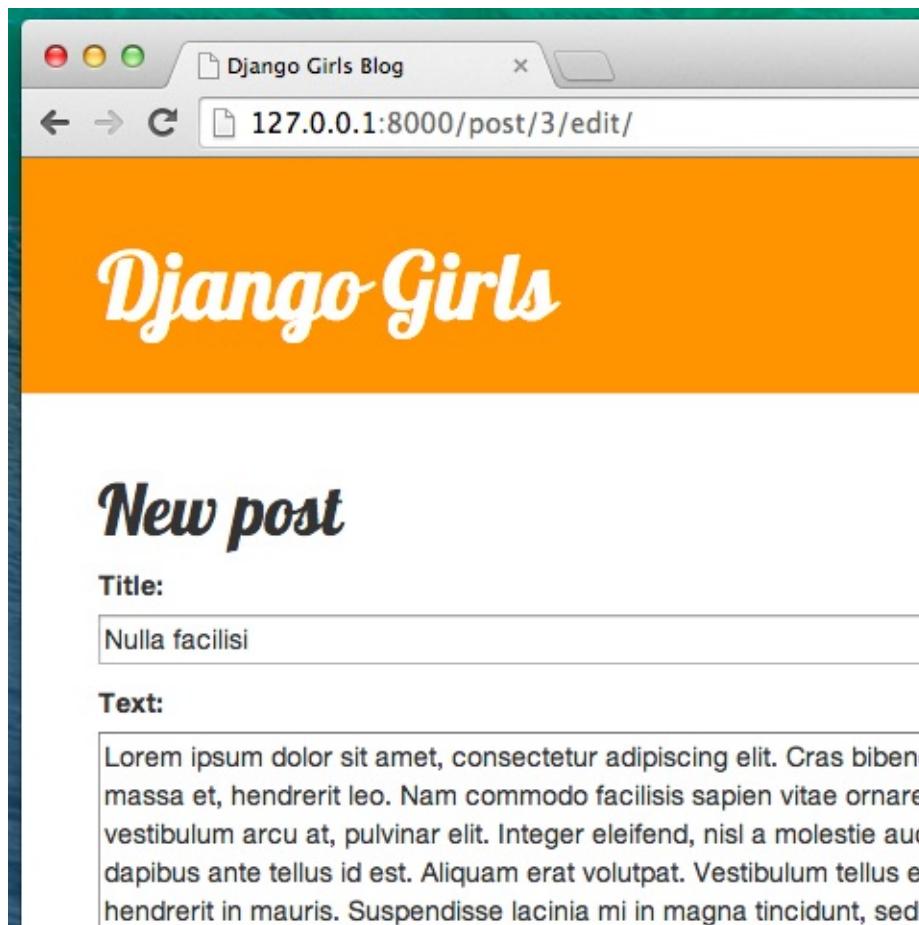
e quando abbiamo appena aperto un form con questo post da modificare:

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

Ok, testiamo se funziona! Andiamo alla pagina `post_detail`. Dovrebbe esserci un pulsante modifica nell'angolo superiore destro:



Quando ci clicchi, vedrai il modulo con i nostri post del blog:



Sentiti libero di cambiare il titolo o il testo e salva le modifiche!

Complimenti! La tua application è sempre più completa!

Se ti servono altre informazioni sui forms di Django dovresti leggere la documentazione:

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

Sicurezza

Riuscire a creare nuovi post semplicemente cliccando su un link è bellissimo! Ma, al momento, chiunque visiti il tuo sito potrebbe pubblicare un nuovo post nel tuo blog e probabilmente non vuoi che ciò accada. Facciamo in modo che questo tasto sia visibile solo per te e non per altri.

Vai al tuo `blog/templates/blog/base.html` e trova il `page-header` `div` con il tag di tipo anchor che hai messo prima. Dovrebbe apparire così:

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

Vogliamo aggiungere qui un altro tag del tipo `{% if %}` che farà comparire il link solo per gli utenti connessi come admin. Per ora, solo tu! Cambia il tag `<a>` in modo che risulti così:

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

Grazie a questo `{% if %}`, il link verrà inviato al browser solo se l'utente che prova ad accedere alla pagina è connesso come admin. Con questa condizione non ci siamo protetti del tutto dalla creazione di nuovi post, ma abbiamo fatto un buon passo avanti. Nelle lezioni estensione ci occuperemo di più della sicurezza.

Ora, dato che probabilmente sei connessa/o, se aggiorni la pagina non troverai alcuna differenza. Ma prova a ricaricare la pagina in un altro browser o in una finestra di navigazione in incognito e vedrai che il link non c'è!

Ultima cosa: ora di fare il deploy!

Vediamo se funziona su PythonAnywhere. È l'ora di un altro deploy!

- Innanzitutto, fai un commit del tuo nuovo codice e fai il push per caricarlo su Github

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

- Poi, in una [console PythonAnywhere Bash](#):

```
$ cd my-first-blog  
$ source myvenv/bin/activate  
(myenv)$ git pull  
[...]  
(myenv)$ python manage.py collectstatic  
[...]
```

- Infine, vai sulla [Web tab](#) e premi **Reload**.

Fatto! Congratulazioni :)

Quali sono le prospettive?

Complimenti! **Sei veramente incredibile.** Siamo orgogliosi! < 3

Cosa si fa ora?

Prenditi una pausa e rilassati. Hai veramente fatto un passo da gigante.

Dopo di che, assicurati di:

- Seguire Django Girls su [Facebook](#) o [Twitter](#) per rimanere aggiornata

Mi puoi consigliare ulteriori risorse?

Sì! Vai avanti e prova il nostro libro dal titolo [Django Girls Tutorial: estensioni](#).

Più avanti potrai provare le risorse elencate qui sotto. Sono tutte molto consigliate!

- [Django - tutorial ufficiale](#)
- [Tutorial per i nuovi Coder](#)
- [Corso Code Academy Python](#)
- [Corso Code Academy HTML & CSS](#)
- [Tutorial Django Carrots](#)
- [Il libro "Learn Python The Hard Way"](#)
- [Lezioni video "Inizia con Django"](#)
- [Il libro "Two Scoops of Django: Best Practices for Django 1.8 book"](#)