

Tutorial

djangogirls

Table des matières

Introduction	1.1
Installation	1.2
Comment fonctionne l'Internet ?	1.3
Introduction à l'interface en ligne de commande	1.4
Installation de Python	1.5
L'éditeur de texte	1.6
Introduction à Python	1.7
Qu'est-ce que Django?	1.8
Installation de Django	1.9
Votre premier projet Django !	1.10
Les modèles dans Django	1.11
Django admin	1.12
Déployer !	1.13
Les urls Django	1.14
Créons nos vues Django!	1.15
Introduction au HTML	1.16
Django ORM (Querysets)	1.17
Données dynamiques dans les templates	1.18
Templates Django	1.19
CSS - Rendez votre site joli	1.20
Héritage de template	1.21
Finaliser votre application	1.22
Formulaires Django	1.23
La suite ?	1.24

Tutoriel de Django Girls

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

Cette œuvre est mise à disposition selon les termes de la Licence Creative Commons Attribution-ShareAlike 4.0 International. Pour obtenir une copie de cette licence, visitez <https://creativecommons.org/licenses/by-sa/4.0/>

Translation

This tutorial has been translated from English to French by a group of awesome volunteers. Special thanks for help go to Lucie Daeye, Georges Dubus, Emmanuelle Delescolle, Leila, MadCath, Mélanie Chauvel and Sébastien Barbier. <3 <3

Introduction

Avez-vous déjà eu l'impression que la technologie prend une place de plus en plus importante, mais que vous êtes en quelque sorte laissée à la traîne ? Avez-vous déjà été curieuse de comment créer un site web, sans jamais avoir le courage de vous plonger dedans ? Vous êtes-vous déjà dit que le monde du logiciel est trop compliqué pour savoir ne serait-ce que par où l'attaquer ?

Hé bien, bonne nouvelle ! Programmer n'est pas aussi dur que ça en a l'air, et nous sommes là pour vous montrer à quel point ça peut être amusant.

Ce tutoriel ne va pas vous transformer en programmeuse du jour au lendemain. Devenir vraiment bonne peut prendre des mois, voire même des années d'apprentissage et de pratique. Mais nous voulons vous montrer que programmer ou créer des sites web n'est pas aussi compliqué que ça en ait l'air. Nous allons essayer de vous expliquer différents morceaux afin de rendre la technologie moins intimidante.

Nous espérons arriver à vous faire aimer la technologie autant que nous l'aimons !

Qu'apprendrez-vous au cours de ce tutoriel ?

À la fin de ce tutoriel, vous aurez une application toute simple et pleinement fonctionnelle : votre propre blog. Nous allons vous montrer comment le mettre en ligne afin de pouvoir montrer le résultat à d'autres personnes !

Ça devrait ressembler plus ou moins à ça :

The screenshot shows a web browser window titled "Django Girls Blog" with the URL "127.0.0.1:8000". The page has a yellow header with the "Django Girls" logo and navigation icons. Below the header, there are three blog post 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...

Si vous travaillez sur ce tutoriel dans votre coin et que vous n'avez pas de coach pour vous aider, venez sur le chat :

[gitter](#) [join chat](#) en cas de problème. Nous avons demandé aux coachs et participant·e·s des précédentes éditions de passer de temps en temps pour aider les autres avec le tutoriel. N'allez pas peur et allez poser vos questions !

Alors, [commençons par le commencement...](#)

À propos et contributions

Ce tutoriel est maintenu par [DjangoGirls](#). Si vous rencontrez des erreurs ou souhaitez simplement suggérer une amélioration du tutoriel, il est important de respecter [les règles de contribution](#).

Voulez-vous nous aider à traduire le tutoriel dans d'autres langues ?

Pour l'instant, les traductions sont stockées sur la plate-forme [crowdin.com](#) dans :

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

Si votre langue n'est pas sur crowdin, veuillez [ouvrir un ticket](#) avec la langue, pour que nous puissions l'ajouter.

Si vous suivez ce tutoriel chez vous

Si vous suivez ce tutoriel chez vous et non dans un [évènement Django Girls](#), vous pouvez passer directement au chapitre [Comment fonctionne l'Internet ?](#).

Les informations données ici sont couvertes dans le reste du tutoriel. Cette partie permet simplement de regrouper au même endroit tout ce qu'il est nécessaire d'installer avant de participer à un évènement. Les évènements Django Girls incluent une "soirée d'installation" qui permet de prendre un peu d'avance sur la journée de formation proprement dite.

Rien ne vous empêche de tout installer maintenant si vous le souhaitez. Cependant, si vous avez envie d'apprendre des choses avant d'installer plein de trucs sur votre ordinateur : passez ce chapitre et installez ce dont vous avez besoin au fil des chapitres.

Bonne chance !

Installation

Pendant l'atelier, vous allez apprendre à construire un blog. Afin d'être prête pour le jour J, vous devrez installer les éléments listés sur cette page.

Installer Python

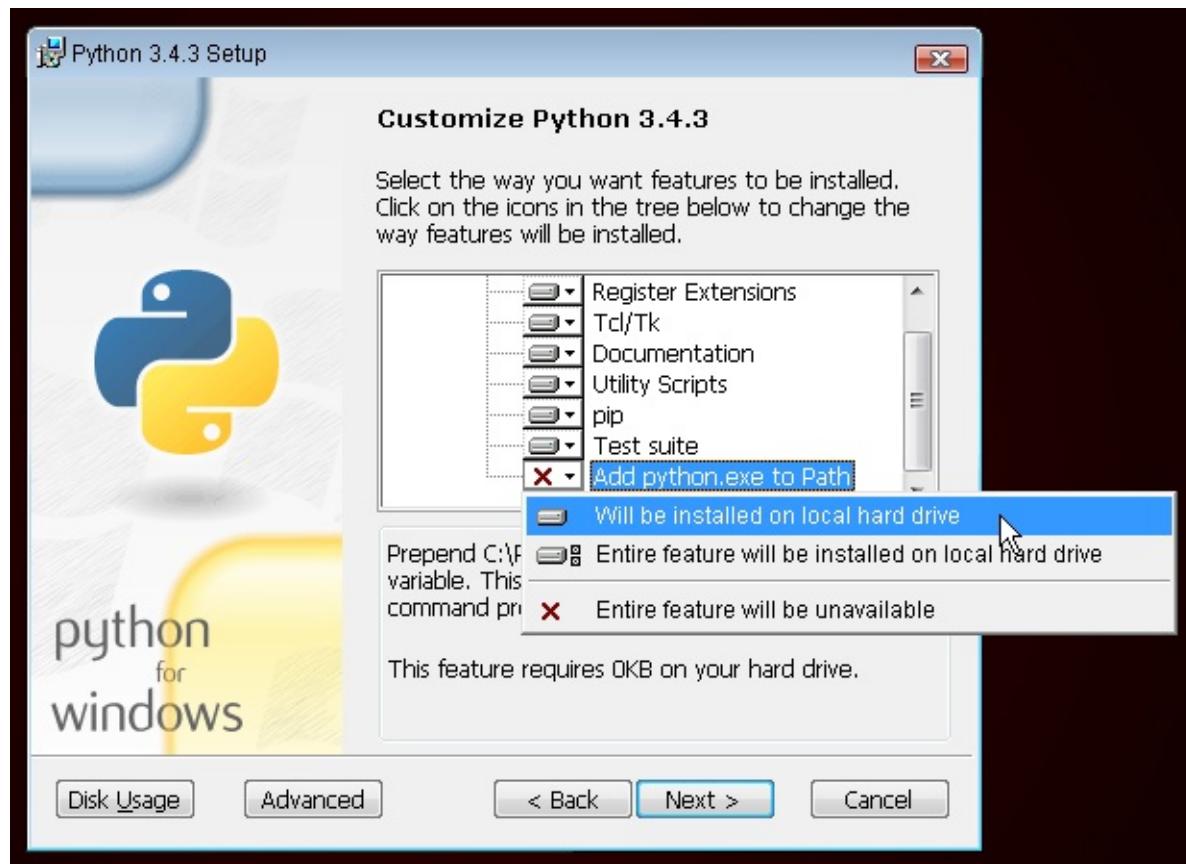
Note : ce sous-chapitre est en partie inspiré d'un autre tutoriel réalisé par les Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>)

Django est écrit en Python. Pour réaliser quelque chose en Django, il va nous falloir Python. Commençons par installer ce dernier ! Pour ce tutoriel, nous utilisons la version 3.4 de Python. Si vous avez une version antérieure, il va falloir la mettre à jour.

Windows

Vous pouvez télécharger Python pour Windows sur le site web <https://www.python.org/downloads/release/python-343/>. Après avoir téléchargé le fichier ***.msi**, lancez-le en double-cliquant sur son icône et suivez les instructions qui s'affichent à l'écran. Attention : il est important de se souvenir du chemin d'accès (le dossier) où vous avez installé Python. Vous en aurez besoin plus tard.

Une chose à laquelle vous devez faire attention : dans le second écran de l'installateur intitulé "Customize", assurez-vous de bien dérouler l'écran jusqu'en bas et de choisir l'option "Ajouter python.exe au chemin", comme sur l'image ci dessous :



Linux

Il est très probable que Python soit déjà installé sur votre machine. Afin de vérifier qu'il est bien installé (et surtout quelle version vous avez), ouvrez une console et tapez la commande suivante :

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

Si Python n'est pas installé ou que vous avez une version différente, vous pouvez l'installer en suivant les instructions suivantes :

Debian ou Ubuntu

Tapez cette commande dans votre terminal :

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

Fedorâ

Tapez cette commande dans votre terminal :

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

openSUSE

Tapez cette commande dans votre terminal :

```
$ sudo zypper install python3
```

OS X

Vous devez aller sur le site <https://www.python.org/downloads/release/python-343/> et télécharger l'installateur Python :

- Téléchargez le fichier *Mac OS X 64-bit/32-bit installer*,
- Double-cliquez sur le fichier *python-3.4.3-macosx10.6.pkg* pour lancer l'installateur.

Vérifiez que l'installation s'est bien déroulée en ouvrant votre *Terminal* et en lançant la commande `python3` :

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

Si vous avez des questions ou si quelque chose ne fonctionne pas et que vous ne savez pas quoi faire : demandez de l'aide à votre coach ! Il arrive parfois que les choses ne se déroulent pas comme prévu et il est alors préférable de demander à quelqu'un qui a plus d'expérience.

Mettre en place virtualenv et installer Django

Note : ce chapitre est en partie inspiré d'un autre tutoriel réalisé par les Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Ce chapitre est en partie inspiré du [tutoriel django-marcador](#) qui est sous licence Creative Commons Attribution-ShareAlike 4.0 International License. Le tutoriel django-marcador a été créé par Markus Zapke-Gründemann et al.

L'environnement virtuel

Avant d'installer Django, nous allons vous faire installer un outil extrêmement utile qui vous aidera à maintenir votre environnement de développement propre. Nous vous recommandons fortement de ne pas sauter cette étape, même si elle n'est pas indispensable. En commençant avec la meilleure configuration possible vous éviterez beaucoup de problèmes par la suite !

Donc, commençons par créer un **environnement virtuel de programmation** (ou *virtualenv*). Chaque projet aura sa propre configuration en Python/Django grâce à *virtualenv*. Ce qui veut dire que si vous modifiez un site web, ça n'affectera pas les autres sites sur lesquels vous travaillez. Plutôt cool, non ?

Tout ce dont vous avez besoin, c'est de trouver un dossier où vous voulez créer votre `virtualenv` ; vous pouvez choisir votre home par exemple. Sous Windows, le home ressemble à `c:\utilisateurs\Nom` (où `Nom` est votre login).

Dans ce tutoriel, nous allons utiliser un nouveau dossier `djangogirls` que vous allez créer dans votre dossier home :

```
mkdir djangogirls
cd djangogirls
```

Nous allons créer un *virtualenv* appelé `myvenv`. Pour cela, nous taperons une commande qui ressemblera à :

```
python3 -m venv myvenv
```

Windows

Afin de créer un nouveau `virtualenv`, vous avez besoin d'ouvrir votre console (nous en avons déjà parlé dans un chapitre précédent. Est-ce que vous vous en souvenez ?) et tapez `c:\Python34\python -m venv myvenv`. Ça ressemblera à ça :

```
C:\Utilisateurs\Nom\djangogirls> C:\Python34\python -m venv myvenv
```

C:\Python34\python doit être le nom du dossier où vous avez installé Python et `myvenv` doit être le nom de votre `virtualenv`. Vous pouvez choisir un autre nom mais attention : il doit être en minuscules, sans espaces et sans accents ou caractères spéciaux. C'est aussi une bonne idée de choisir un nom plutôt court, car vous aller souvent l'utiliser !

Linux et OS X

Pour créer un `virtualenv` sous Linux ou OS X, tapez simplement la commande `python3 -m venv myvenv`. Ça ressemblera à ça :

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

`myvenv` est le nom de votre `virtualenv`. Vous pouvez choisir un autre nom, mais veillez à n'utiliser que des minuscules et à n'insérer ni espaces, ni caractères spéciaux. C'est aussi une bonne idée de choisir un nom plutôt court, car vous aller souvent l'utiliser!

NOTE: initialiser un environnement virtuel sous Ubuntu 14.04 de cette manière donne l'erreur suivante :

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

Pour résoudre ce problème, utilisez plutôt la commande `virtualenv`.

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

Travailler avec `virtualenv`

Les commandes listées ci-dessus permettent de créer un dossier appelé `myvenv` (ou le nom que vous avez choisi) qui contient notre environnement virtuel. Pour faire simple, c'est un dossier composé lui-même d'autres dossiers et de fichiers.

Windows

Démarrez votre environnement virtuel en exécutant :

```
C:\Utilisateurs\Nom\djangogirls> myvenv\Scripts\activate
```

Linux et OS X

Démarrez votre environnement virtuel en exécutant :

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

N'oubliez pas de remplacer `myvenv` par le nom que vous avez choisi pour votre `virtualenv` (le cas échéant) !

NOTE : il arrive parfois que `source` ne soit pas disponible. Dans ce cas, vous pouvez essayer ceci :

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

Vous saurez que votre `virtualenv` est lancé quand le prompt de votre console ressemblera à ceci :

```
(myvenv) C:\Utilisateurs\Nom\djangogirls>
```

ou :

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

Vous remarquez que le préfixe `(myvenv)` est apparu !

Quand vous travaillez dans un environnement virtuel, la commande `python` fera automatiquement référence à la bonne version de Python. Vous pouvez donc utiliser `python` plutôt que `python3`.

Ok, nous avons installé toutes les dépendances dont nous avions besoin. Nous allons enfin pouvoir installer Django !

Installation de Django

Maintenant que vous avez lancé votre `virtualenv`, vous pouvez installer Django à l'aide de `pip`. Dans votre console, tapez `pip install django~=1.11.0`. Notez bien que nous utilisons un tilde suivi du signe égal : `~=`).

```
(myvenv) ~$ pip install django~=1.11.0
Downloading/unpacking django==1.11
Installing collected packages: django
Successfully installed django
Cleaning up...
```

Sous Windows :

Si jamais vous obtenez des erreurs lorsque vous utilisez pip sous Windows, vérifiez si votre chemin d'accès contient des espaces, des accents ou des caractères spéciaux (ex : `c:\Utilisateurs\Nom d'Utilisateur\djangogirls`). Si c'est le cas, changez de dossier et essayez d'en créer un nouveau en prenant en compte le fait qu'il ne doit donc avoir ni accents, ni espaces, ni caractères spéciaux (ex : `c:\djangogirls`). Après l'avoir déplacé, essayez de retaper la commande précédente.

Sous Linux :

Si vous obtenez une erreur lorsque vous utilisez pip sous Ubuntu 12.04, tapez la commande `python -m pip install -U --force-reinstall pip` pour réparer l'installation de pip dans votre virtualenv.

Et voilà ! Vous êtes (enfin) prêt(e) pour créer votre première application Django !

Installez un éditeur de code

Choisir un éditeur de texte parmi tous ceux qui sont disponibles est surtout une histoire de goûts personnels. La plupart des programmeurs Python utilisent des IDE (Environnements de développement intégrés) complexes mais très puissants, comme PyCharm par exemple. Ce n'est pas forcément le meilleur choix pour débuter : ceux que nous vous recommandons sont tout aussi puissants, mais beaucoup plus simples à utiliser.

Vous pouvez choisir l'un des éditeurs de la liste ci-dessous, mais n'hésitez pas à demander à votre coach l'éditeur qu'il·elle préfère.

Gedit

Gedit est un éditeur libre et gratuit disponible pour tous les systèmes d'exploitation.

[Télécharger](#)

Sublime Text 3

Sublime text est un éditeur très populaire : il est disponible gratuitement sous forme de version d'évaluation pour tout les systèmes d'exploitation. Il est facile à installer et à utiliser.

[Télécharger](#)

Atom

Atom est un éditeur très récent créé par [GitHub](#). Disponible pour tout les systèmes d'exploitation, il est libre, gratuit, facile à installer et à utiliser.

[Télécharger](#)

Pourquoi installer un éditeur de texte?

Vous vous demandez sûrement pourquoi nous vous faisons installer un éditeur spécialement créé pour écrire du code. Pourquoi ne pourrions nous pas simplement utiliser un éditeur de texte comme Word ou Notepad ?

La première raison est que votre code doit être écrit en **texte brut**. Le problème avec les applications comme Word ou Textedit, c'est qu'elles ne produisent pas du texte brut mais du texte enrichi (avec des polices et de la mise en page), basé sur un standard comme [RTF \(Rich Text Format\)](#).

La seconde raison est que les éditeurs de texte dédiés à la programmation contiennent de nombreuses fonctions très utiles. Ils peuvent colorer le texte en fonction du sens de celui-ci (coloration syntaxique) ou ajouter automatiquement un guillemet fermant à chaque fois que vous ouvrez un guillemet.

Vous allez bientôt vous rendre compte à quel point un logiciel dédié à la programmation peut être pratique ! Prenez un peu de temps pour trouver l'éditeur qui vous convient, car il deviendra rapidement l'un de vos outils préférés :)

Installer Git

Windows

Vous pouvez télécharger Git sur [git-scm.com](#). Vous pouvez cliquer sur "next" à toutes les étapes, sauf pour la cinquième, "Adjusting your PATH environment" : n'oubliez pas de choisir "Run Git and associated Unix tools from the Windows command-line", situé en bas de la liste des options disponibles. Les autres choix par défaut n'ont pas besoin d'être modifiés. L'option "Checkout Windows-style, commit Unix-style line endings" est parfaite: vous n'avez rien à changer sur cette page.

MacOS

Vous pouvez télécharger Git sur [git-scm.com](#). Pour le reste de l'installation, suivez simplement les instructions de l'installateur.

Linux

Git est probablement déjà installé mais, si ce n'est pas le cas, voici les instructions à suivre :

```
sudo apt install git
# ou
sudo yum install git
# ou
sudo zypper install git
```

Créer un compte GitHub

Allez sur [GitHub.com](#) et créez-vous un nouveau compte gratuitement.

Créer un compte PythonAnywhere

Si vous ne l'avez pas encore fait, n'oubliez pas de vous créer un compte "Beginner" sur PythonAnywhere.

- [www.pythonanywhere.com](#)

: Le nom d'utilisateur que vous allez choisir va déterminer l'adresse de votre blog de la manière suivante :
`votrenomutilisateur.pythonanywhere.com`. Si vous ne savez pas quoi prendre, nous vous conseillons de choisir votre surnom ou un nom proche du sujet de votre blog.

Commencer à lire

Félicitations, vous avez tout installé et êtes prête ! Si vous avez toujours du temps avant l'atelier, il peut être utile de commencer à lire les premiers chapitres :

- [Comment fonctionne l'Internet ?](#)
- [Introduction à la ligne de commande](#)
- [Introduction à Python](#)
- [Qu'est-ce que Django?](#)

Comment fonctionne l'Internet ?

Ce chapitre est inspiré par la présentation "How the Internet works" par Jessica McKellar (<http://web.mit.edu/jessstess/www/>).

Vous utilisez sûrement Internet tous les jours. Mais savez-vous ce qu'il se passe vraiment quand vous tapez une adresse comme [https://django.org](https://.djangoproject.org) dans votre navigateur et appuyez sur Entrée ?

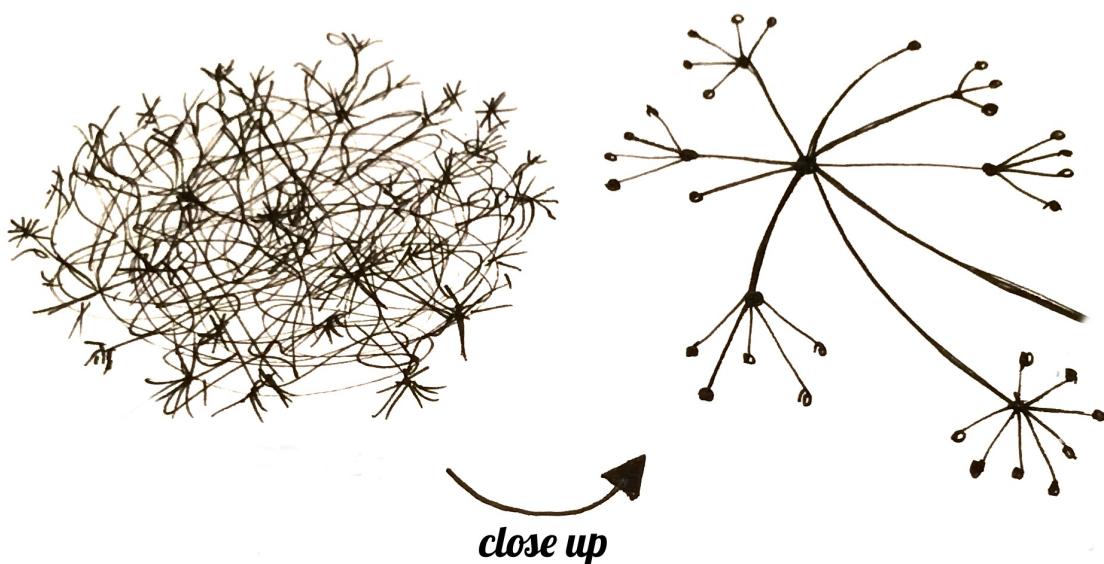
Avant tout, il faut savoir qu'un site web n'est rien de plus qu'un tas de fichiers sauvegardés sur un disque dur. Exactement comme vos vidéos, vos musiques ou vos photos. Cependant, les sites web ont quelque chose d'unique : ils contiennent du code informatique appelé HTML.

Si vous n'avez pas l'habitude de la programmation, il peut être difficile de comprendre HTML au début, mais vos navigateurs web (comme Chrome, Safari, Firefox, etc.) adorent ça. Les navigateurs web sont conçus pour comprendre ce code, pour suivre les instructions qu'il contient et présenter les fichiers de votre site web exactement comme vous voulez qu'ils soient présentés.

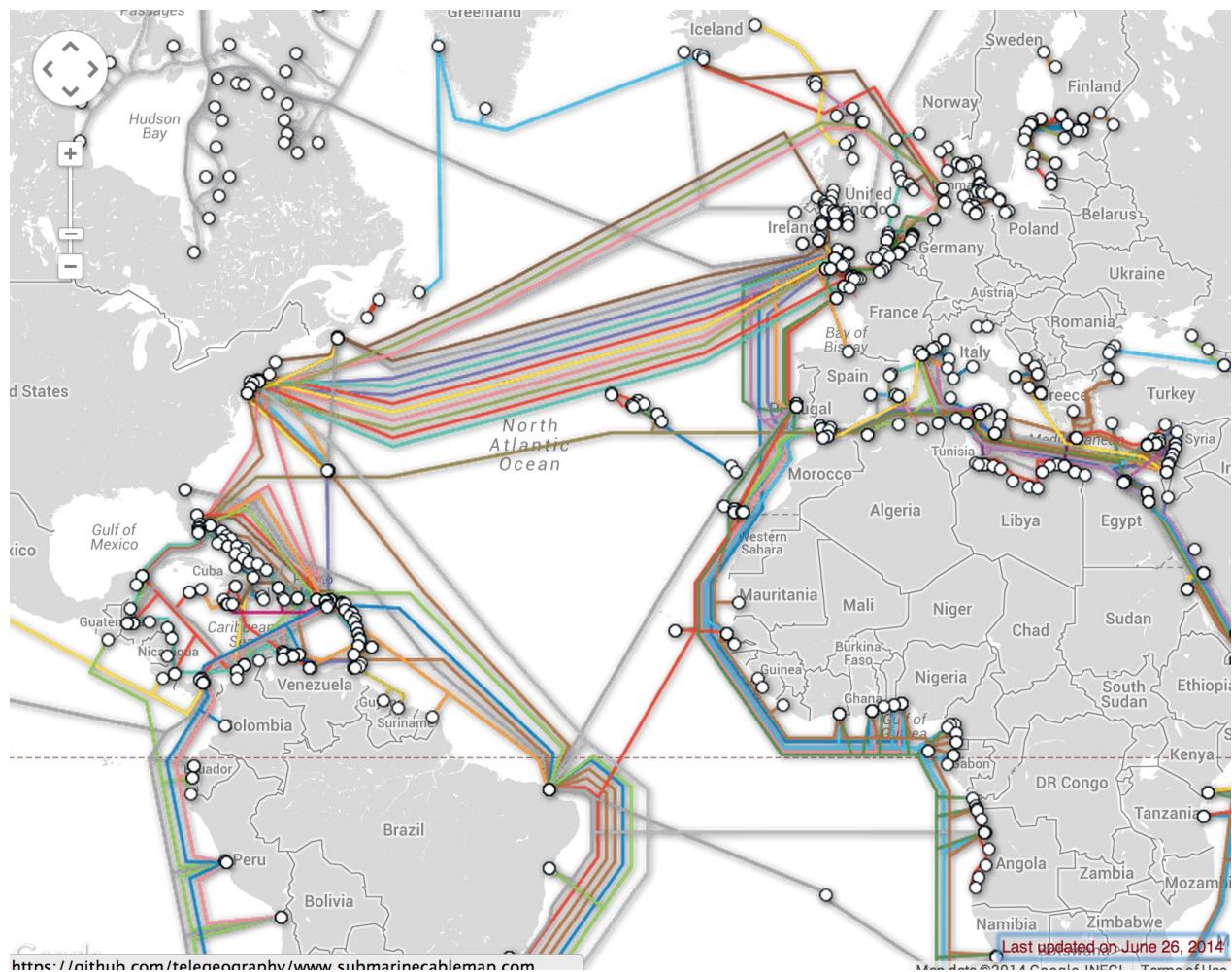
Comme pour n'importe quel autre fichier, il faut stocker les fichiers HTML sur un disque dur quelque part. Pour Internet, on utilise des ordinateurs spéciaux, très puissants, appelés *serveurs*. Ils n'ont pas d'écran, de clavier ou de souris, car leur rôle est de stocker des données, et de les servir. C'est pour ça qu'on les appelle des *serveurs* -- parce qu'ils sont là pour vous *servir* des données.

Bon, d'accord. Mais vous avez envie de savoir à quoi Internet ressemble, n'est-ce-pas ?

On va faire un dessin ! Internet ressemble à ça :

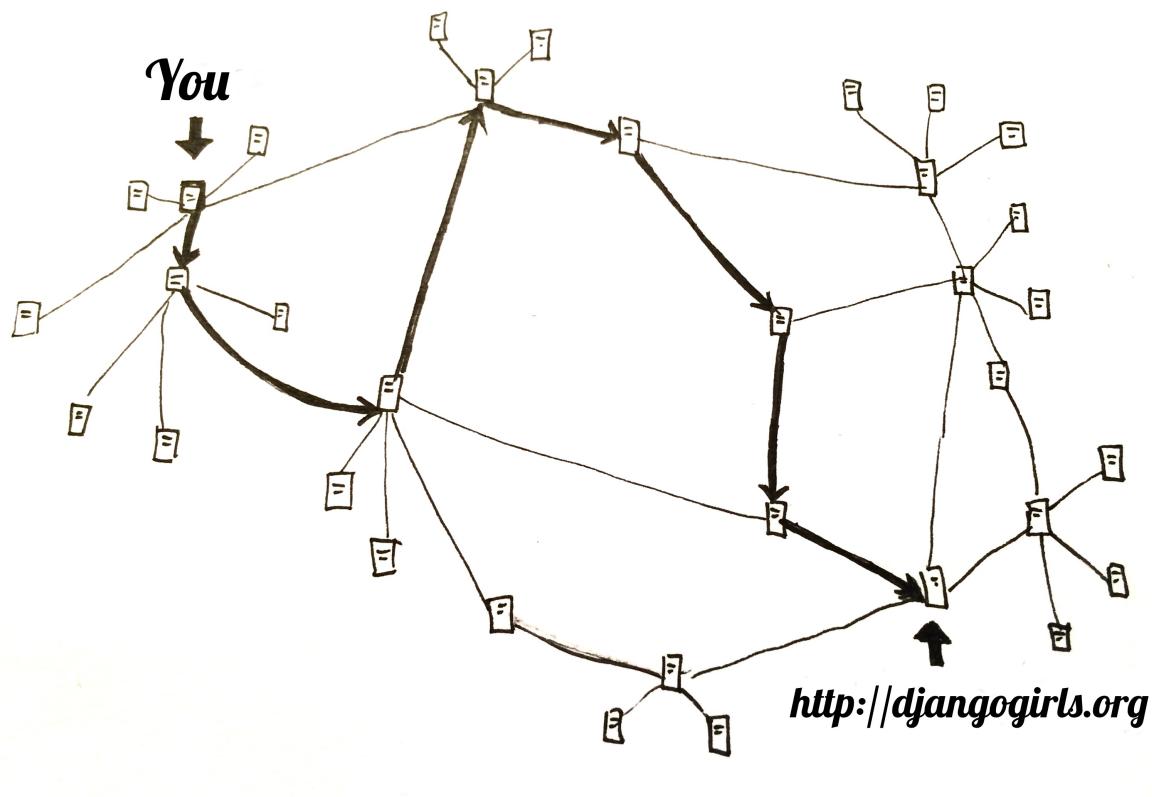


C'est le bazar, non ? En fait, c'est un réseau de machines connectées entre elles (les *serveurs* dont on parlait plus tôt). Des centaines de milliers de machines ! Des millions de kilomètres de câbles, partout dans le monde ! Vous pouvez aller voir une carte des câbles sous-marins (<http://submarinecablemap.com/>) pour voir à quel point le réseau est compliqué. Voici une capture d'écran du site :



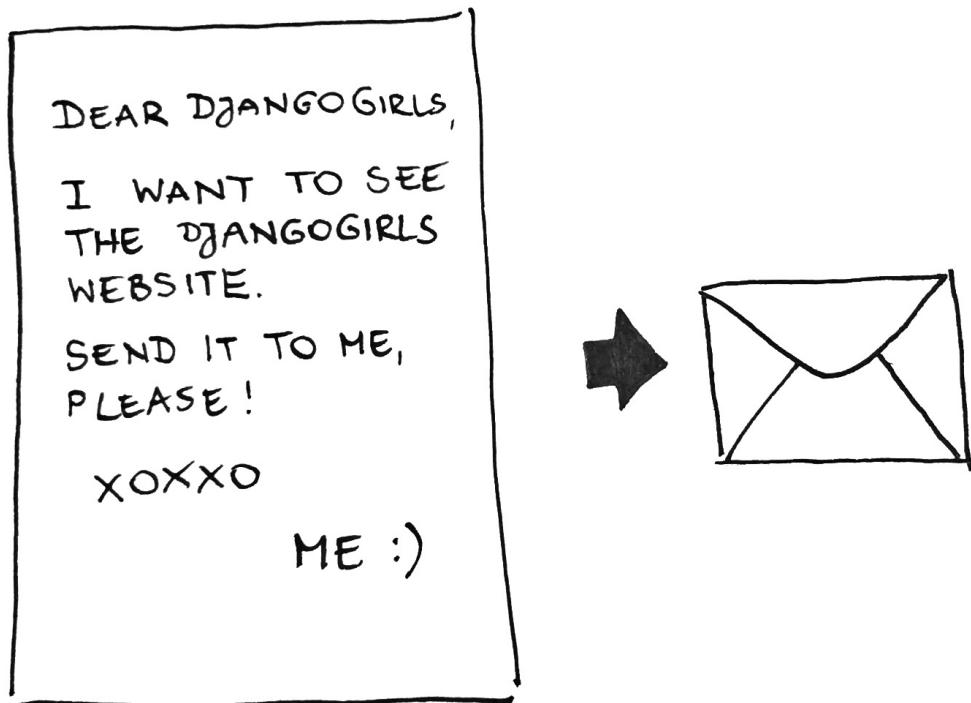
Fascinant, non ? Cependant, il n'est évidemment pas possible de tirer un câble entre chaque machine connectée à Internet. Du coup, pour atteindre une machine (par exemple, celle où <https://djangogirls.org> est sauvegardé), on doit faire passer une requête par plein d'autres machines.

Ça ressemble ça :



C'est un peu comme si, quand vous tapez <https://djangogirls.org>, vous envoyiez une lettre qui dit "Chères Django Girls, je voudrais voir le site djangogirls.org. Pouvez-vous me l'envoyer ?"

Votre lettre part vers le bureau de poste le plus proche. Ensuite, il file vers un autre, qui est plus proche de votre destinataire. Puis un autre, et encore un autre, jusqu'à sa destination. Une chose à retenir : si vous envoyez beaucoup de lettres (*data packets*) au même endroit, il se pourrait qu'elles transitent par des postes différentes (*routers*). Cela dépend de la manière dont elles sont distribuées à chaque bureau de poste.



Oui oui, c'est aussi simple que ça. Vous envoyez des messages et attendez une réponse. Alors, bien sûr, le papier et le crayon sont remplacés par des octets de données, mais l'idée est la même.

À la place des adresses postales (nom de rue, ville, code postal), nous utilisons des adresses IP. Votre ordinateur commence par demander au DNS (Domaine Name System) de traduire djangogirls.org en une adresse IP. Ça marche un peu comme un de ces vieux annuaires où l'on peut chercher le nom d'une personne et trouver son numéro de téléphone et son adresse.

Quand vous envoyez une lettre, elle a besoin de certaines choses pour transiter correctement, comme une adresse et un timbre. Vous devez aussi utiliser une langue que votre destinataire comprend. C'est la même chose pour les paquets de données que vous envoyez pour voir un site web. Vous utilisez un protocole appelé HTTP (Hypertext Transfer Protocol).

Donc, au final, pour avoir un site web il faut qu'il soit sur un *serveur* (c'est une machine). Lorsque le *serveur* reçoit une *requête* (dans une lettre), il envoie votre site Web (dans une autre lettre). Puisqu'on est dans un tutoriel sur Django, vous devez vous demander ce que Django fait. Quand vous envoyez une réponse, vous ne renvoyez pas toujours la même réponse à tout le monde. C'est bien mieux quand les lettres sont personnalisées, surtout quand elles s'adressent à quelqu'un qui vient de vous écrire, non ? Et bien Django vous aide à écrire les lettres personnalisées et intéressantes :).

Assez parlé, il est temps de commencer à créer des trucs !

Introduction à l'interface en ligne de commande

C'est un peu exaltant, non ? Dans quelques instants, vous allez écrire votre première ligne de code :)

Commençons par vous présenter un nouvel ami : la ligne de commande !

Les étapes suivantes vont vous montrer comment utiliser la fenêtre noire que tous les bidouilleurs·euses utilisent. Elle est un peu effrayante à première vue, mais en fait, c'est tout simplement un programme qui attend qu'on lui donne des commandes.

Note : Il existe deux mots pour parler de dossier : dossier ou répertoire. Il se peut que nous utilisions les deux dans le tutoriel mais, pas de panique : ils signifient la même chose.

Qu'est-ce qu'une ligne de commande ?

Cette fenêtre, qu'on appelle aussi **ligne de commande** ou **interface en ligne de commande**, est une application textuelle qui permet de voir et de manipuler des fichiers sur votre ordinateur. C'est un peu la même chose que l'Explorateur Windows ou Finder dans Mac, mais sans interface graphique. On l'appelle parfois aussi : *cmd*, *CLI*, *prompt*, *console* ou *terminal*.

Ouvrir l'interface en ligne de commande

Pour commencer à expérimenter, nous avons d'abord besoin d'ouvrir notre interface en ligne de commande.

Windows

Aller dans Menu démarrer → Tous les programmes → Accessoires → Invite de commandes.

Mac OS X

Applications → Utilitaires → Terminal.

Linux

Vous la trouverez probablement dans Applications → Accessoires → Terminal, mais ça dépend de votre système. Si elle n'est pas là, demandez à Google :)

Prompt

Vous devriez maintenant voir une fenêtre noire ou blanche qui attend vos commandes.

Si vous êtes sous Mac ou Linux, vous verrez probablement un `$`, comme ça :

```
$
```

Sur Windows, c'est un signe `>`, comme ça :

```
>
```

Chaque commande commence par ce signe, puis un espace. Mais vous n'avez pas besoin de le taper, votre ordinateur le fait pour vous :)

Petite remarque : il se peut que vous voyiez quelque chose comme `C:\Users\ola>` OU `olas-MacBookAir:~ola$` avant le signe de prompt. Pas de problème : c'est parfaitement normal. C'est juste parce que dans ce tutoriel, nous tentons de simplifier les choses autant que possible.

Votre première commande (YAY !)

Commençons par quelque chose de simple. Tapez la commande suivante :

```
$ whoami
```

ou

```
> whoami
```

Puis, appuyez sur la touche `entrée`. Voilà ce qui s'affiche chez moi :

```
$ whoami  
olasitarska
```

Comme vous pouvez le voir, l'ordinateur vient d'afficher votre nom d'utilisateur. Sympa, non ? ;)

Essayez de taper chaque commande sans copier-coller. Ça aide à les retenir !

Les bases

Les différents systèmes d'exploitation ont des commandes légèrement différentes, donc faites attention à suivre les instructions pour votre système d'exploitation. Allons-y !

Dossier courant

Ce serait pratique de savoir dans quel répertoire nous nous trouvons. Pour le savoir, tapez la commande suivante et appuyez sur `entrée` :

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

Si vous êtes sous Windows :

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

Vous verrez probablement quelque chose de similaire sur votre machine. Quand vous ouvrez une ligne de commande, vous démarrez habituellement dans le dossier personnel de votre utilisateur.

Remarque : "pwd" veut dire "print working directory" (afficher le dossier courant).

Lister les fichiers et les dossiers

Du coup, que pouvons-nous trouver dans ce dossier personnel ? Pour le savoir, essayons ceci :

```
$ ls
Applications
Bureau
Musique
Téléchargements
...
```

Windows :

```
> dir
Directory of C:\Users\olasitarska
05/08/2014 07:28 PM <DIR>    Applications
05/08/2014 07:28 PM <DIR>    Bureau
05/08/2014 07:28 PM <DIR>    Musique
05/08/2014 07:28 PM <DIR>    Téléchargements
...
```

Changer le dossier courant

Maintenant, essayons d'aller sur notre bureau :

```
$ cd Bureau
```

Windows :

```
> cd Bureau
```

Vérifions que ça a bien changé :

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

Windows :

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

Et voilà !

Pro tip : si vous tapez `cd B` puis que vous appuyez sur la touche `tabulation`, la ligne de commande va automatiquement compléter le reste du nom. Cela va vous permettre d'aller plus vite et d'éviter des fautes de frappe. Si plusieurs dossiers commencent par un « B », appuyez sur `tabulation` deux fois pour avoir une liste des options.

Créer un dossier

Que diriez-vous de créer un répertoire dédié aux exercices sur votre bureau ? Vous pouvez le faire de cette façon :

```
$ mkdir exercices
```

Windows :

```
> mkdir exercices
```

Cette petite commande crée un dossier nommé `exercices` sur votre bureau. Vous pouvez vérifier qu'il est bien là en regardant votre bureau, ou en lançant la commande `ls` ou `dir` ! Essayez donc :)

Pro tip : Si vous voulez éviter de taper les mêmes commandes plein de fois, essayez d'appuyer sur les touches flèche haut et flèche bas pour retrouver les dernières commandes que vous avez tapé.

Un peu d'exercice !

Petit défi pour vous : dans votre nouveau dossier `exercices`, créez un dossier appelé `test`. Pour ça, utilisez les commandes `cd` et `mkdir`.

Solutions :

```
$ cd exercices  
$ mkdir test  
$ ls  
test
```

Windows :

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

Félicitations ! :)

Nettoyage

Supprimons tout ce qu'on vient de faire, histoire d'éviter de laisser du bazar.

D'abord, revenons au Bureau :

```
$ cd ..
```

Windows :

```
> cd ..
```

Grâce à `..` et la commande `cd`, vous pouvez aller directement dans le dossier parent de votre répertoire courant (c'est à dire le dossier qui contient le dossier dans lequel vous étiez).

Vérifiez où vous êtes :

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

Windows :

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

Maintenant, il est temps de supprimer notre dossier `exercices` :

Attention : Supprimer des fichiers avec `del`, `rmdir` ou `rm` est irréversible, ce qui veut dire que *les fichiers supprimés sont perdus à jamais* ! Du coup, faites très attention avec cette commande.

```
$ rm -r exercices
```

Windows :

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

Et voilà. Pour être sûr que le dossier a bien été supprimé, vérifiez :

```
$ ls
```

Windows :

```
> dir
```

Sortir

C'est tout pour le moment ! Vous pouvez maintenant fermer la ligne de commande. Faisons-le à la manière des bidouilleurs·euses. :)

```
$ exit
```

Windows :

```
> exit
```

Cool, non ? :)

Résumé

Voici un résumé de quelques commandes utiles :

Commande (Windows)	Commande (Mac OS / Linux)	Description	Exemple
exit	exit	ferme la fenêtre	<code>exit</code>
cd	cd	change le dossier courant	<code>cd test</code>
dir	ls	liste des fichiers/dossiers	<code>dir</code>
copy	cp	copie un fichier	<code>copy c:\test\test.txt c:\windows\test.txt</code>
move	mv	déplace un fichier	<code>move c:\test\test.txt c:\windows\test.txt</code>
mkdir	mkdir	crée un nouveau dossier	<code>mkdir testdirectory</code>
del	rm	supprime un dossier/fichier	<code>del c:\test\test.txt</code>

Ce ne sont que quelques-unes des commandes que vous pouvez utiliser dans votre ligne de commande. Cette liste est suffisante pour réaliser ce tutoriel.

Si vous êtes curieuse, ss64.com contient une référence complète de toutes les commandes pour tous les systèmes d'exploitation.

Vous êtes prête ?

Nous allons plonger dans Python !

Commençons par Python

On est parti !

Tout d'abord, laissez-nous vous en dire un peu plus sur Python. Python est un langage de programmation très populaire qui peut être utilisé pour créer des sites web, des jeux, des logiciels scientifiques, des graphiques et bien d'autres choses encore.

Python a été créé à la fin des années 1980. L'objectif principal des créateurs du langage était de rendre ce langage de programmation lisible aussi bien par des humains que par des machines. Par conséquent, il a l'air beaucoup plus simple à lire que d'autres langages de programmation. Cependant, ne vous fiez pas à son apparence simplifiée de lecture et d'apprentissage : Python est un langage très puissant !

Installation de Python

: Si vous avez suivi la partie installation du tutoriel, vous n'avez pas besoin d'installer Python à nouveau. Vous pouvez sauter cette partie et passer au chapitre suivant !

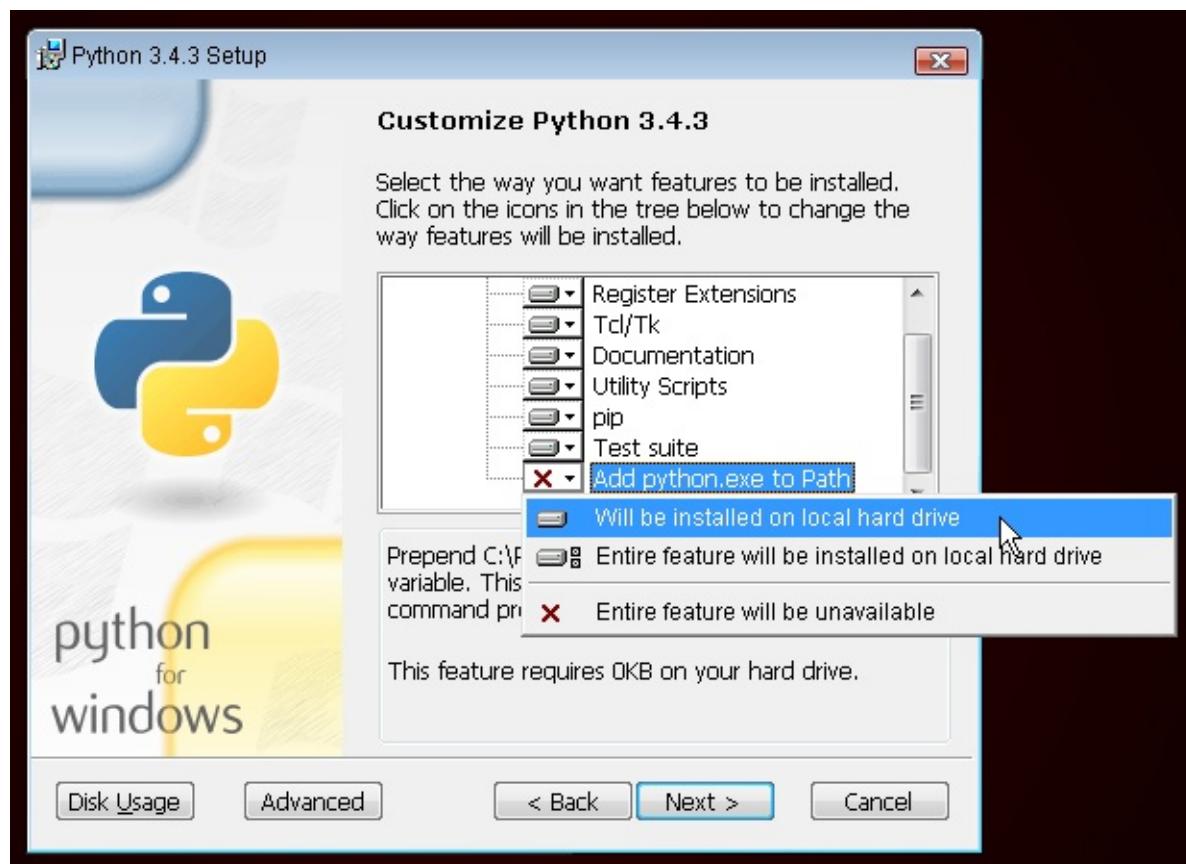
Note : ce sous-chapitre est en partie inspiré d'un autre tutoriel réalisé par les Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>)

Django est écrit en Python. Pour réaliser quelque chose en Django, il va nous falloir Python. Commençons par installer ce dernier ! Pour ce tutoriel, nous utilisons la version 3.4 de Python. Si vous avez une version antérieure, il va falloir la mettre à jour.

Windows

Vous pouvez télécharger Python pour Windows sur le site web <https://www.python.org/downloads/release/python-343/>. Après avoir téléchargé le fichier *.msi, lancez-le en double-cliquant sur son icône et suivez les instructions qui s'affichent à l'écran. Attention : il est important de se souvenir du chemin d'accès (le dossier) où vous avez installé Python. Vous en aurez besoin plus tard.

Une chose à laquelle vous devez faire attention : dans le second écran de l'installateur intitulé "Customize", assurez-vous de bien dérouler l'écran jusqu'en bas et de choisir l'option "Ajouter python.exe au chemin", comme sur l'image ci dessous :



Linux

Il est très probable que Python soit déjà installé sur votre machine. Afin de vérifier qu'il est bien installé (et surtout quelle version vous avez), ouvrez une console et tapez la commande suivante :

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

Si Python n'est pas installé ou que vous avez une version différente, vous pouvez l'installer en suivant les instructions suivantes :

Debian ou Ubuntu

Tapez cette commande dans votre terminal :

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

Fedorâ

Tapez cette commande dans votre terminal :

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

openSUSE

Tapez cette commande dans votre terminal :

```
$ sudo zypper install python3
```

OS X

Vous devez aller sur le site <https://www.python.org/downloads/release/python-343/> et télécharger l'installateur Python :

- Téléchargez le fichier *Mac OS X 64-bit/32-bit installer*,
- Double-cliquez sur le fichier *python-3.4.3-macosx10.6.pkg* pour lancer l'installateur.

Vérifiez que l'installation s'est bien déroulée en ouvrant votre *Terminal* et en lançant la commande `python3` :

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

Si vous avez des questions ou si quelque chose ne fonctionne pas et que vous ne savez pas quoi faire : demandez de l'aide à votre coach ! Il arrive parfois que les choses ne se déroulent pas comme prévu et il est alors préférable de demander à quelqu'un qui a plus d'expérience.

L'éditeur de texte

Vous allez bientôt écrire vos premières lignes de code : Il vous faut tout d'abord télécharger un éditeur de texte !

: Vous l'avez peut-être déjà fait dans le chapitre d'installation. Si c'est le cas, passez directement au prochain chapitre!

Choisir un éditeur de texte parmi tous ceux qui sont disponibles est surtout une histoire de goûts personnels. La plupart des programmeurs Python utilisent des IDE (Environnements de développement intégrés) complexes mais très puissants, comme Pycharm par exemple. Ce n'est pas forcément le meilleur choix pour débuter : ceux que nous vous recommandons sont tout aussi puissants, mais beaucoup plus simples à utiliser.

Vous pouvez choisir l'un des éditeurs de la liste ci-dessous, mais n'hésitez pas à demander à votre coach l'éditeur qu'il·elle préfère.

Gedit

Gedit est un éditeur libre et gratuit disponible pour tous les systèmes d'exploitation.

[Télécharger](#)

Sublime Text 3

Sublime Text est un éditeur très populaire : il est disponible gratuitement sous forme de version d'évaluation pour tous les systèmes d'exploitation. Il est facile à installer et à utiliser.

[Télécharger](#)

Atom

Atom est un éditeur très récent créé par [GitHub](#). Disponible pour tous les systèmes d'exploitation, il est libre, gratuit, facile à installer et à utiliser.

[Télécharger](#)

Pourquoi installer un éditeur de texte?

Vous vous demandez sûrement pourquoi nous vous faisons installer un éditeur spécialement créé pour écrire du code. Pourquoi ne pourrions-nous pas simplement utiliser un éditeur de texte comme Word ou Notepad ?

La première raison est que votre code doit être écrit en **texte brut**. Le problème avec les applications comme Word ou Textedit, c'est qu'elles ne produisent pas du texte brut mais du texte enrichi (avec des polices et de la mise en page), basé sur un standard comme [RTF \(Rich Text Format\)](#).

La seconde raison est que les éditeurs de texte dédiés à la programmation contiennent de nombreuses fonctions très utiles. Ils peuvent colorer le texte en fonction du sens de celui-ci (coloration syntaxique) ou ajouter automatiquement un guillemet fermant à chaque fois que vous ouvrez un guillemet.

Vous allez bientôt vous rendre compte à quel point un logiciel dédié à la programmation peut être pratique ! Prenez un peu de temps pour trouver l'éditeur qui vous convient, car il deviendra rapidement l'un de vos outils préférés :)

Introduction à Python

Note : ce chapitre est en partie inspiré d'un autre tutoriel réalisé par les Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Allons écrire du code !

Le prompt Python

Pour commencer à jouer avec Python, nous avons besoin d'ouvrir une *ligne de commande* sur votre ordinateur. Normalement, vous savez déjà comment le faire -- vous l'avez appris dans le chapitre [Introduction à la ligne de commande](#).

Dès que vous êtes prête, suivez les instructions suivantes.

Afin d'ouvrir une console Python, tapez `python` sous Windows ou `python3` sous Mac OS/Linux et appuyez sur `entrée`.

```
$ python3
Python 3.4.3 (...)

Type "help", "copyright", "credits" or "license" for more information.

>>>
```

Votre première commande Python !

Après avoir lancé la commande Python, votre prompt (ou invite de commandes) s'est changé en `>>>`. Cela signifie que maintenant, les seules commandes que nous pouvons taper sont dans le langage Python. Vous n'avez pas besoin de taper `>>>` - Python fait ça pour vous.

Quand vous voudrez sortir de la console Python, tapez `exit()` ou utilisez le raccourci `ctrl + z` pour Windows ou `ctrl + d` pour Mac/Linux. Après ça, vous ne verrez plus le `>>>`.

Pour le moment, nous ne voulons pas quitter la console Python car nous nous aimeraisons mieux la connaître. Démarrons avec quelque chose de vraiment simple. Par exemple, faisons un peu de math : tapez `2 + 3` et appuyez sur `entrée`.

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

Pas mal ! Vous voyez comment la réponse est sortie ? Python sait faire des maths ! Vous pouvez essayer d'autres commandes comme : `- 4 * 5 - 5 - 1 - 40 / 2`

Amusez-vous un peu avec ça, et revenez ici après :).

Comme vous pouvez le constater, Python est une très bonne calculette. Comme vous vous en doutez, il est aussi capable de faire autre chose ...

Chaînes de caractères (Strings)

Et si nous essayions avec votre nom ? Tapez votre prénom entre guillemets, comme cela :

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

Vous venez de créer votre première chaîne de caractères ! C'est une suite de caractères qui peut être traitée par un ordinateur. Une chaîne de caractères doit toujours commencer et terminer par le même caractère. Cela peut être un guillemet simple (') ou un guillemet double ("), ça n'a pas d'importance. Cela permet à Python de savoir que tout ce qui se trouve à l'intérieur de ces guillemets est une chaîne de caractères.

Il est possible d'assembler des chaînes de caractères comme ceci :

```
>>> "Salut " + "Ola"
'Salut Ola'
```

Vous pouvez aussi multiplier une chaîne de caractères par un nombre :

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

Si vous avez besoin de mettre une apostrophe dans votre chaîne de caractères, vous avez deux possibilités.

Vous pouvez utiliser des guillemets doubles :

```
>>> "J'aime la mousse au chocolat"
"J'aime la mousse au chocolat"
```

ou échapper l'apostrophe avec une barre oblique inversée (un backslash, \) :

```
>>> 'J\''aime la mousse au chocolat'
"J'aime la mousse au chocolat"
```

Pas mal, non ? Pour voir votre nom en majuscules, tapez juste :

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

Vous venez d'utiliser la **fonction** `upper` sur votre chaîne de caractères ! Une fonction (comme `upper()`) est un ensemble d'instructions que Python va effectuer sur un objet donné ("Ola") lorsque vous l'appellerez.

Si vous voulez savoir combien il y a de lettres dans votre nom, il y a une fonction pour ça !

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

Vous avez peut-être remarqué que parfois, on appelle la fonction avec . en la plaçant après la chaîne de caractères (comme "Ola".upper()) alors qu'à d'autres moment, on appelle d'abord la fonction puis la chaîne de caractères entre parenthèses ? Il s'avère que dans certains cas, les fonctions appartiennent à des objets (c'est le cas de `upper()`) et qu'elles ne peuvent être appliquées qu'à des chaînes de caractères. Dans ce cas, on appelle la fonction une **méthode**. D'autres fois, les fonctions n'appartiennent à rien de particulier et peuvent être utilisées sur différents types d'objets (c'est le cas de `len()`). C'est pour ça que nous passons "Ola" comme argument à la fonction `len`.

Résumé

OK, assez parlé de chaînes de caractères. Jusque-là, nous avons découvert :

- **le prompt** - taper des commandes (du code) dans le prompt Python donne des réponses dans Python
- **les nombres et les chaînes de caractères** - dans Python, les nombres sont utilisés pour faire des calculs, et les chaînes de caractères pour manipuler du texte
- **opérateurs** - comme + et * qui combinent des valeurs pour en obtenir de nouvelles
- **les fonctions** - comme `upper()` et `len()` qui effectuent des actions sur les objets.

Ce sont des bases présentes dans tous les langages de programmation que vous pouvez apprendre. Prête pour quelque chose de plus compliqué ? Allons-y !

Les erreurs

Essayons quelque chose de nouveau. Pouvons-nous obtenir la longueur d'un nombre de la même façon que celle de notre nom ? Tapez `len(304023)` et appuyez sur entrée :

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

Nous venons d'obtenir notre première erreur ! Elle nous dit que les objets de type "int" (integers, ce qui signifie nombre entier) n'ont pas de longueur. Que pouvons-nous faire, du coup ? Pourquoi ne pas essayer d'écrire notre nombre comme une chaîne de caractères ? Après tout, les chaînes de caractères ont bien une taille, non ?

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

Ça a marché ! Nous avons utilisé la fonction `str` à l'intérieur de la fonction `len`. La fonction `str()` convertit n'importe quoi en chaîne de caractères.

- La fonction `str` convertit des choses en **chaînes de caractères**
- La fonction `int` convertit des choses en **entiers**

Important : il est possible de convertir des nombres en texte, mais il n'est pas toujours possible de convertir du texte en nombres. Parce que, bon, ça vaudrait quoi `int('salut')` ?

Variables

Il existe un concept super important en programmation : les variables. Une variable, c'est juste un nom pour quelque chose que l'on aimerait utiliser plus tard. Les programmeurs-euses utilisent des variables pour stocker des données, rendre leur code plus lisible, et pour ne pas avoir à se rappeler de ce que sont les choses.

Disons que nous aimerais créer une variable appelée `name` :

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

Vous voyez ? C'est tout bête ! C'est simplement : `name` vaut Ola.

Vous avez peut-être remarqué que contrairement à tout à l'heure, le programme ne renvoie rien. Du coup, comment faire pour vérifier que la variable existe vraiment ? Tapez simplement `name` et appuyez sur entrée :

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

Youpi ! Votre première variable :) ! Vous pouvez toujours changer ce à quoi elle fait référence :

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

Vous pouvez aussi l'utiliser dans des fonctions :

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

Génial, non ? Et bien sûr, les variables peuvent être n'importe quoi, y compris des nombres ! Essayez ça :

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

Mais que se passe-t-il si nous utilisons le mauvais nom ? Essayez de deviner ! C'est parti !

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

Une erreur ! Comme vous pouvez le voir, Python a différents types d'erreurs, et celle-ci est une **NameError**. Python vous donne cette erreur quand vous essayez d'utiliser une variable qui n'a pas encore été définie. Si vous rencontrez cette erreur par la suite, vérifiez dans votre code que vous n'avez pas fait une faute de frappe dans une variable.

Jouez un peu avec les variables et essayez de voir ce que vous pouvez faire !

La fonction print

Essayez ça :

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

Quand vous tapez `name`, l'interpréteur Python répond avec la *représentation* de la chaîne de caractères associée à la variable "name", c'est à dire les lettres M-a-r-i-a, entourées par des guillemets simples. Quand vous dites `print(name)`, Python va "imprimer" le contenu de la variable sur l'écran, sans les guillemets, ce qui est plus sympa.

Comme nous le verrons plus tard, `print()` est aussi utile lorsque l'on veut afficher des choses depuis l'intérieur de fonctions ou des choses sur plusieurs lignes.

Les listes

En plus des chaînes de caractères et des entiers, Python possède tout un tas d'autres types d'objets. Nous allons maintenant vous présenter un type appelé **listes**. Les listes sont exactement ce que vous pensez qu'elles sont : des objets qui sont des listes d'autres objets :)

Allez-y, créez une liste :

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

Oui, cette liste est vide. Pas très utile, non ? Créons maintenant une liste de numéros de loterie. Nous ne voulons pas nous répéter tout le temps, donc mettons-la dans une variable :

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

Voilà, nous avons une liste ! Qu'est ce que nous pourrions en faire ? Commençons par voir combien de numéros de loterie il y a dans cette liste. Une idée de la fonction pour faire ça ? Vous la connaissez déjà !

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

Hé oui ! `len()` peut aussi vous donner le nombre d'objets dans une liste. Pratique, non ? Peut-être qu'on peut aussi la trier :

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

Ça ne renvoie rien : cette fonction a juste changé l'ordre dans lequel les nombres apparaissent dans la liste. Affichons-la encore pour voir ce qu'il s'est passé :

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

Comme vous pouvez le voir, les nombres de la liste sont maintenant triés du plus petit au plus grand. Bravo !

Pouvons-nous inverser cet ordre ? Essayons !

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

Facile, non ? Si vous voulez ajouter quelque chose à la liste, vous pouvez le faire en tapant cette commande :

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

Si vous ne souhaitez afficher que le premier nombre, vous pouvez le faire en utilisant des **indices**. Un indice est un nombre qui dit où l'élément apparaît dans la liste. Les programmeurs·euses préfèrent compter à partir de 0 : le premier objet dans notre liste a donc pour indice 0, le suivant 1 et ainsi de suite. Essayez ça :

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

Comme vous pouvez le voir, nous pouvons accéder à différents objets dans la liste en utilisant le nom de la liste suivi de l'indice de l'objet entre crochets.

Pour supprimer un objet de votre liste, vous aurez besoin de son **indice** ainsi que de la commande `pop()`. Essayons l'exemple suivant : supprimez le premier numéro de votre liste.

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

Ça marche à merveille !

Jouons encore un peu avec les indices ! Essayez-en des nouveaux : 6, 7, 1000, -1, -6 ou -1000. Est-ce que vous arrivez à prévoir le résultat avant de taper la commande ? Est-ce que ces résultats vous paraissent logiques ?

Vous pouvez trouver une liste complète des méthodes disponibles pour les listes dans ce chapitre de la documentation de Python : <https://docs.python.org/3/tutorial/datastructures.html>

Dictionnaires

Un dictionnaire est un peu comme une liste. Cependant, nous utilisons des clefs plutôt que des indices pour accéder aux valeurs. Une clef peut être n'importe quelle chaîne de caractère ou n'importe quel nombre. La syntaxe pour définir un dictionnaire vide est la suivante :

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

C'est comme ça que l'on crée un dictionnaire vide. Hourra !

Maintenant, essayez d'écrire la commande suivante (et essayez aussi de changer le contenu) :

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

Avec cette commande, vous venez de créer une variable nommée `participant` avec trois paires clef-valeur :

- La clef `name` pointe vers la valeur 'Ola' (un objet chaîne de caractères),
- `country` pointe vers 'Poland' (une autre chaîne de caractères),
- et `favorite_numbers` pointe vers [7, 42, 92] (une liste contenant trois nombres).

Vous pouvez vérifier le contenu de chaque clef avec cette syntaxe :

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

Vous voyez, c'est un peu comme une liste; Cependant, vous n'avez pas besoin de vous souvenir de l'indice, juste de son nom.

Que se passe-t-il lorsque nous demandons à Python la valeur correspondant à une clef qui n'existe pas ? Pouvez-vous le deviner ? Essayons voir !

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

Oh, une autre erreur ! Celle-ci est une **KeyError**. Python nous donne un coup de main et nous dit que la clef 'age' n'existe pas dans le dictionnaire.

Vous vous demandez peut-être quand est-ce qu'il faut utiliser un dictionnaire ou une liste ? C'est une bonne question. Réfléchissez-y un instant avant de regarder la réponse à la ligne suivante.

- Vous avez besoin d'une suite ordonnée d'éléments ? Utilisez une liste.
- Vous avez besoin d'associer des valeurs à des clefs, de manière à pouvoir les retrouver efficacement (par clef) par la suite ? Utilisez un dictionnaire.

Comme les listes, les dictionnaires sont *mutables*, ce qui signifie qu'ils peuvent être modifiés après leur création. Vous pouvez ajouter de nouvelles paires clé/valeur au dictionnaire après sa création, comme ceci :

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

Comme pour les listes, la fonction `len()` permet d'obtenir le nombre de paires clef-valeur du dictionnaire. Essayez et tapez la commande suivante :

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

J'espère que c'est compréhensible pour l'instant :) Prête pour s'amuser un peu plus avec les dictionnaires ? Passez à la ligne suivante pour voir des trucs géniaux.

Vous pouvez utiliser la commande `pop()` pour supprimer un élément du dictionnaire. Par exemple, si vous voulez supprimer l'entrée correspondant à la clé « `favorite_numbers` », tapez la commande suivante :

```
>>> participant.pop('favorite_numbers')
>>> participant
{'country': 'Poland', 'favorite_language': 'Python', 'name': 'Ola'}
```

Comme vous pouvez le voir dans votre console, la paire clef-valeur correspondant à "favorite_numbers" a été supprimée.

De même, vous pouvez changer la valeur associée à une clef déjà créée dans le dictionnaire. Tapez ceci :

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

Voilà, la valeur de la clé "country" a été modifiée de "Poland" à "Germany" . :) Ce n'est pas cool ça ? Yep ! Un autre truc génial d'appris !

Résumé

C'est super ! Vous savez plein de choses sur la programmation maintenant. Dans cette partie, vous avez appris :

- **les erreurs** - vous savez maintenant comment lire et comprendre les erreurs qui apparaissent quand Python ne comprend pas l'une de vos commandes
- **les variables** - des noms pour les objets qui vous permettent de coder plus facilement et de rendre votre code plus lisible
- **les listes** - des listes d'objets stockés dans un ordre particulier
- **les dictionnaires** - des objets stockés sous forme de paires clef-valeur

On continue ? :)

Comparer des choses

Comparer des choses est très important en programmation. Quelle serait la chose la plus facile à comparer ? Les nombres, bien sûr ! Voyons voir comment ça marche :

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

Nous avons donné à Python des nombres à comparer. Comme vous pouvez le voir, Python peut comparer des nombres, mais aussi des résultats de méthodes. Pas mal, non ?

Vous vous demandez probablement pourquoi nous avons mis deux signes `==` côté à côté pour savoir si deux nombres étaient égaux ? On utilise déjà `=` pour assigner des valeurs aux variables. Du coup, il faut toujours, oui **toujours**, mettre deux `==` si vous voulez savoir si deux choses sont égales. Nous pouvons aussi dire que certaines choses ne sont pas égales à d'autres Pour cela, nous utilisons le symbole `!=`, comme illustré dans l'exemple ci-dessus.

Donnons encore un peu de boulot à Python :

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

> et `<` sont faciles, mais qu'est ce que `>=` et `<=` veulent dire ? Ils se lisent comment ça :

- `x > y` veut dire : `x` est plus grand que `y`
- `x < y` signifie: `x` est inférieure à `y`
- `x <= y` signifie: `x` est inférieur ou égal à `y`
- `x >= y` veut dire : `x` est supérieur ou égal à `y`

Super ! Un dernier ? Essayez ça :

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

Vous pouvez donner à Python autant de nombres à comparer que vous le souhaitez et il vous donnera une réponse. Plutôt malin, non ?

- **and** - si vous utilisez l'opérateur `and` (et), les deux comparaisons doivent être True (vraies) pour que la commande toute entière soit True
- **or** - si vous utilisez l'opérateur `or` (ou), il suffit qu'une des deux comparaisons soit True (vraie) pour que la commande toute entière soit True

Vous connaissez l'expression "on ne compare pas les choux et les carottes" ? Essayons l'équivalent en Python :

```
>>> 1 > 'django'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'gt' not supported between instances of 'int' and 'str'
```

Comme vous le voyez, Python n'est pas capable de comparer un nombre (`int`) et une chaîne de caractères (`str`). À la place, il nous montre une **TypeError** et nous dit que les deux types ne peuvent pas être comparés.

Booléen

Au passage, vous venez de découvrir un nouveau type d'objets en Python. On l'appelle **Booléen**. C'est probablement le type le plus simple qui existe.

Il n'y a que deux objets Booléens : - True (vrai) - False (faux)

Pour que Python comprenne qu'il s'agit d'un Booléen, il faut toujours l'écrire True (première lettre en majuscule, les autres en minuscule). **true**, **TRUE**, **tRUE** ne marchent pas -- seul **True** est correct. (Et c'est aussi vrai pour **False**.)

Les Booléens aussi peuvent être des variables ! regardez :

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

Vous pouvez aussi faire ça :

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

Entraînez-vous et amusez-vous avec les Booléens en essayant de lancer les commandes suivantes :

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

Bravo ! Les Booléens sont l'une des fonctionnalités les plus cools en programmation et vous venez juste d'apprendre comment les utiliser !

Sauvegardez tout ça !

Pour l'instant, nous avons écrit tout notre code Python directement dans l'interpréteur, ce qui nous limite à une ligne à la fois. Les programmes normaux sont sauvegardés dans des fichiers et sont exécutés par **l'interpréteur** ou le **compilateur** de notre langage de programmation. Jusque-là, c'était ligne par ligne dans **l'interpréteur**. Nous allons avoir besoin de bien plus qu'une ligne de code par la suite alors, vous allez rapidement avoir besoin de :

- Quitter l'interpréteur Python
- Ouvrir l'éditeur de code de notre choix
- Sauvegarder du code dans un nouveau fichier Python
- Le lancer !

Pour quitter l'interpréteur Python que nous sommes en train d'utiliser, il suffit de taper la fonction `exit()` :

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

Cela vous ramènera dans la ligne de commande de votre système d'exploitation.

Tout à l'heure, dans la section [L'éditeur de texte](#), nous avons choisi un éditeur de texte. Ouvrez-le et écrivez le code suivant dans un nouveau fichier :

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

Note : Vous avez probablement constaté que votre code se pare de multiples couleurs : ça fait partie des choses qui rendent la programmation dans un éditeur de texte bien plus agréable. Votre console Python n'avait pas cette fonctionnalité : tout était donc de la même couleur. Dans votre éditeur de texte, vous devriez voir que la fonction `print` possède différentes couleurs. C'est ce qu'on appelle « la coloration syntaxique ». C'est une fonctionnalité très utile lorsque l'on programme. La couleur des choses va vous permettre de détecter des problèmes : une chaîne de caractères non fermée, une faute dans un mot (ce sera par exemple le cas dans la fonction `def` que vous verrez un peu plus bas). C'est ce genre de fonctionnalités qui font que vous aimerez rapidement programmer avec un éditeur de code :)

Vous avez maintenant pas mal de connaissances en Python : n'hésitez pas à écrire du code avec ce que vous avez appris aujourd'hui !

Sauvegardons maintenant le fichier en lui donnant un nom descriptif. On n'a qu'à l'appeler **python_intro.py** et le sauvegarder sur le bureau. Vous pouvez donner le nom que vous souhaitez à ce fichier mais il est important qu'il se termine par **.py**. L'extension **.py** permet de signaler à votre système d'exploitation que ce fichier est un **fichier exécutable Python** et que Python peut le lancer.

Une fois que le fichier est sauvegardé, vous pouvez le lancer ! En utilisant les compétences que vous avez apprises dans la section sur la ligne de commande, utilisez le terminal pour **changer le dossier courant** vers le bureau.

Sur Mac, ça ressemblera à :

```
$ cd ~/Desktop
```

Sous Linux, comme ça (il se peut que le mot Bureau soit dans une autre langue) :

```
$ cd ~/Desktop
```

Et sous Windows, ce sera comme ça :

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

Si vous êtes bloquée, n'hésitez pas à appeler à l'aide.

Maintenant, utilisez Python pour exécuter le code contenu dans votre fichier :

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

Super ! Vous venez de lancer votre premier programme python à partir d'un fichier. Cool non ?

Et maintenant, passons à un autre outil essentiel de la programmation :

If...elif...else

Quand on code, il y a plein choses qui ne doivent être exécutées que dans certaines conditions. Pour cela, Python possède ce qu'on appelle l'instruction **if** (si).

Remplacez le code dans votre fichier **python_intro.py** avec ceci :

```
if 3 > 2:
```

Si nous sauvegardons ce fichier et que nous le lançons, nous obtiendrons l'erreur suivante :

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

Python s'attend à ce que nous lui donnions des instructions sur ce qu'il faut exécuter lorsque la condition `3 > 2` est vraie (ou plutôt, `True`). Essayons de lui faire afficher "Ça marche !". Remplacez le code dans **python_intro.py** par ceci :

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

Avez-vous remarqué que nous avions décalé la ligne suivante de quatre espaces ? C'est ce que l'on appelle indenter. Nous avons besoin d'indenter pour que Python sache quel code exécuter si le résultat est vrai. Un seul espace suffirait, mais à peu près tous·tes les programmeurs·euses Python pensent que 4 espaces sont plus clairs. Une seule `tab` (tabulation)

compte également comme 4 espaces.

Sauvegardez le fichier et relancez le :

```
$ python3 python_intro.py
Ça marche !
```

Et que se passe-t-il si une condition n'est pas vraie ?

Dans les exemples précédents, le code était exécuté quand la condition était vraie. Cependant, Python possède aussi des instructions `elif` (sinon si) et `else` (sinon) :

```
if 5 > 2:
    print('5 est effectivement plus grand que 2')
else:
    print("5 n'est pas plus grand que 2")
```

Lorsque vous exécuterez le code, ceci s'affichera :

```
$ python3 python_intro.py
5 est effectivement plus grand que 2
```

Et si 2 était plus grand que 5, la seconde commande serait exécutée. Facile, non ? Voyons comment `elif` fonctionne :

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

Exécutons le code :

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

Que s'est-il passé ? `elif` vous permet d'ajouter d'autres conditions à exécuter si les précédentes échouent.

Vous pouvez ajouter autant de `elif` que vous le souhaitez après le premier `if`. Voici un exemple :

```
volume = 57
if volume < 20:
    print("C'est plutôt calme.")
elif 20 <= volume < 40:
    print("Une jolie musique de fond.")
elif 40 <= volume < 60:
    print("Parfait, je peux entendre tous les détails du morceau.")
elif 60 <= volume < 80:
    print("Parfait pour faire la fête !")
elif 80 <= volume < 100:
    print("Un peu trop fort !")
else:
    print("Au secours ! Mes oreilles ! :(")
```

Python va tester les différentes conditions puis il affichera ceci :

```
$ python3 python_intro.py
Parfait, je peux entendre tous les détails du morceau.
```

Résumé

Avec ces trois derniers exercices, vous avez appris :

- **Comment comparer des choses** - en Python, vous pouvez comparer des choses avec `>`, `>=`, `==`, `<=`, `<` et avec les opérateurs `and`, `or`
- **Booléen** - un type d'objet qui n'a que deux valeurs possibles : `True` et `False`
- **Comment sauvegarder des fichiers** - stocker votre code dans des fichiers pour pouvoir écrire des programmes plus longs.
- **if...elif...else** - des instructions que vous permettent de n'exécuter du code que dans certaines conditions.

Il est temps d'attaquer la dernière partie de ce chapitre !

Vos propres fonctions !

Vous vous souvenez des fonctions comme `len()` que vous pouvez exécuter en Python ? Et bien, bonne nouvelle : vous allez apprendre à écrire vos propres fonctions !

Une fonction est un ensemble d'instructions que Python va exécuter. Chaque fonction en Python commence par le mot-clé `def`. On lui donne un nom, et elle peut avoir des paramètres. Commençons par quelque chose de facile. Remplacer le code de `python_intro.py` par ceci :

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

hi()
```

Voilà, notre première fonction est prête !

Vous vous demandez peut-être pourquoi nous avons écrit le nom de la fonction à la fin du fichier. C'est parce que Python lit le fichier et l'exécute du haut vers le bas. Donc pour pouvoir utiliser notre fonction, nous devons la réécrire en bas.

Launchons notre code pour voir ce qui se passe :

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

C'était facile ! Construisons maintenant notre première fonction avec des paramètres. Dans l'exemple précédent, nous avions une fonction qui disait "Hi there!" à la personne qui la lançait. Faisons une fonction identique, mais ajoutons un nom cette fois :

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

Comme vous le voyez, nous avons donné à notre fonction un paramètre appelé `name` :

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

Rappelez-vous : la fonction `print` est indentée de quatre espaces dans le bloc `if`, car elle est exécutée uniquement quand la condition est satisfaite. Voyons comment ça marche :

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

Oups, une erreur. Heureusement, Python nous donne un message d'erreur assez utile. Il nous dit que la fonction `hi()` (celle que nous avons définie) a besoin d'un argument (que nous avons appelé `name`). Nous avons oublier de passer cet argument lorsque nous avons appelé notre fonction. Corrigeons la dernière ligne du fichier :

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

Et exécutez votre code à nouveau :

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

Et que se passe-t-il quand on change de nom ?

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

Exécutez votre code à nouveau :

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

Maintenant, que pensez-vous qu'il se passera lorsque nous écrirons un autre nom (ni Ola, ni Sonja) ? Faites un essai et regardez si vous avez raison. Ceci devrait s'afficher :

```
Hi anonymous!
```

Super, non ? Avec ça, vous n'avez pas besoin de vous répéter lorsque vous voulez changer le nom de la personne à saluer. C'est pour cette raison que nous avons besoin de fonctions : vous ne voulez pas avoir à répéter votre code !

Faisons maintenant quelque chose de plus malin : comme vous le savez, il existe plus de deux prénoms. Cependant, ce serait un peu pénible de devoir écrire une condition pour chacun d'entre eux, n'est-ce pas ?

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

hi("Rachel")
```

Exécutons à nouveau notre code :

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

Félicitations ! Vous venez juste d'apprendre à écrire des fonctions ! :)

Les boucles

C'est déjà la dernière partie. C'était rapide, non ? :)

Les programmeurs·euses n'aiment pas devoir se répéter. L'essence de la programmation est d'automatiser les choses : nous aimerions pouvoir saluer automatiquement chaque personne. Pour cela, nous allons utiliser des boucles.

Vous vous souvenez des listes ? Faisons une liste de Django Girls :

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

Nous voulons saluer chacune d'entre elles par son nom. Nous avons déjà la fonction `hi` pour faire ça, utilisons donc une boucle :

```
for name in girls:
```

L'instruction `for` se comporte un peu comme `if`. Le code qui suit doit donc être indenté de quatre espaces.

Voilà le code complet à mettre dans votre fichier :

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

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

Exécutez votre code :

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

Comme vous le voyez, tout ce que nous avons mis dans un `for` avec une indentation est répété pour chaque élément de la liste `girls`.

Vous pouvez aussi utiliser `for` sur des nombres grâce à la fonction `range` :

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

Ce qui affiche :

```
1
2
3
4
5
```

`range` est une fonction qui crée une liste de nombres qui se suivent (c'est vous qui définissez l'intervalle à l'aide de paramètres).

Vous pouvez remarquer que le second de ces nombres n'est pas inclus dans la liste que Python nous donne (ce qui signifie que `range(1, 6)` compte de 1 à 5, mais n'inclut pas 6). C'est lié au fait que "range" est à moitié ouvert. Cela signifie qu'il inclut la première valeur mais pas la dernière.

Résumé

Et voilà ! **Vous êtes géniale !** Ce chapitre était un peu compliqué et vous devriez être fière de vous ! En tout cas, nous sommes super fières de vous !

N'hésitez pas à prendre une pause : étirez-vous, marchez un peu ou reposez-vous les yeux. Une fois que vous avez un peu rechargé vos batteries, vous pouvez attaquer le chapitre suivant :)



Qu'est-ce que Django?

Django (/dʒæŋgoʊʃ/jang-goh) est un framework web gratuit et libre écrit en Python. Un framework web est un ensemble de composants qui vous aide à développer des sites web plus rapidement et plus facilement.

Lorsque vous créez un site web, vous avez souvent besoin de la même chose : une manière de gérer l'authentification de vos utilisateurs (créer un compte, se connecter, se déconnecter), une partie dédiée à la gestion de votre site, des formulaires, une manière de mettre en ligne des fichiers, etc.

La bonne nouvelle, c'est que d'autres gens se sont aussi rendus compte de ce problème et ont décidé de s'allier avec des développeurs pour le résoudre. Ensemble, ces personnes ont créé différents frameworks, dont Django, pour fournir un set de composants de base qui peuvent être utilisés lors de la création d'un site web.

Les frameworks existent pour vous éviter de réinventer la roue à chaque fois. Ils vous aident aussi à alléger la charge de travail liée à la création d'un site web.

Pourquoi est-ce que vous auriez besoin d'un framework ?

Pour comprendre ce à quoi peut bien servir Django, nous avons besoin de nous intéresser aux multiples rôles des serveurs. Par exemple, la première chose qu'a besoin de savoir un serveur, c'est que vous aimerez qu'il vous affiche une page web.

Imaginez une boîte aux lettres (un port) dont l'arrivée de lettres (une requête) serait surveillée. C'est le travail qu'effectue le serveur. Le serveur web lit la lettre qu'il a reçue et en réponse, retourne une page web. Généralement, lorsque vous voulez envoyer quelque chose, vous avez besoin de contenu. Django est quelque chose qui va vous aider à créer ce contenu.

Que se passe-t-il quand quelqu'un demande un site web à votre serveur ?

Quand une requête arrive sur un serveur web, elle est transmise à Django dont le premier travail va être d'essayer de comprendre ce qui est exactement demandé. Il s'occupe tout d'abord de l'adresse de la page web et essaye de comprendre ce qu'il doit en faire. Ce travail est effectué par l'**urlresolver** de Django (l'adresse d'une page web est appelée URL, Uniform Resource Locator, ce qui nous aide à comprendre le nom *urlresolver*). Comme il n'est pas très malin, il prend une liste de modèles existants et essaye de trouver celui qui correspond à notre URL. Django lit sa liste de modèles du haut vers le bas et si jamais quelque chose correspond, il envoie la requête à la fonction associée (qui s'appelle une *vue* (*view*)).

Afin d'y voir un peu plus clair, imaginez une postière transportant une lettre. Elle descend la rue et vérifie à chaque maison si le numéro de celle-ci correspond à celui de la lettre. Si jamais les deux numéros correspondent, elle met la lettre dans la boîte aux lettres de cette maison. C'est à peu près comme cela que fonctionne l'urlresolver !

C'est dans la fonction *vue* que les choses intéressantes se passent : cela nous permet de jeter un œil dans la base de données pour obtenir des informations. Par exemple, peut-être que l'utilisateur vient de demander de changer quelque chose dans ces données ? Ce serait comme une lettre dont le contenu serait : "Merci de changer la description de mon emploi actuel". La *vue* va tout d'abord vérifier que l'utilisateur est bien autorisé à effectuer ce changement puis elle corrigera la description de l'emploi. Enfin, elle retournera un message de type : "C'est bon, j'ai terminé ! ". La *vue* créera une réponse et c'est Django qui se chargera de la transmettre au navigateur de l'utilisateur.

Bien sûr, ceci n'est qu'une description très simplifiée du processus. Vous n'avez pas besoin de connaître tous les détails techniques pour le moment : cette vue d'ensemble suffira largement.

Maintenant, nous aurions la possibilité de vous assommer avec des détails complexes sur comment tout cela fonctionne. À la place, nous allons plutôt commencer à construire quelque chose ! Nous vous donnerons toutes les informations importantes au fur et à mesure que vous progresserez. Ce sera plus sympa, non ?

Installation de Django

Si vous avez déjà suivi les étapes d'Installation, vous n'avez rien d'autre à faire - vous pouvez aller directement au chapitre suivant !

Note : ce chapitre est en partie inspiré d'un autre tutoriel réalisé par les Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Ce chapitre est en partie inspiré du [tutoriel django-marcador](#) qui est sous licence Creative Commons Attribution-ShareAlike 4.0 International License. Le tutoriel django-marcador a été créé par Markus Zapke-Gründemann et al.

L'environnement virtuel

Avant d'installer Django, nous allons vous faire installer un outil extrêmement utile qui vous aidera à maintenir votre environnement de développement propre. Nous vous recommandons fortement de ne pas sauter cette étape, même si elle n'est pas indispensable. En commençant avec la meilleure configuration possible vous éviterez beaucoup de problèmes par la suite !

Donc, commençons par créer un **environnement virtuel de programmation** (ou *virtualenv*). Chaque projet aura sa propre configuration en Python/Django grâce à virtualenv. Ce qui veut dire que si vous modifiez un site web, ça n'affectera pas les autres sites sur lesquels vous travaillez. Plutôt cool, non ?

Tout ce dont vous avez besoin, c'est de trouver un dossier où vous voulez créer votre `virtualenv` ; vous pouvez choisir votre home par exemple. Sous Windows, le home ressemble à `C:\Utilisateurs\Nom` (où `Nom` est votre login).

Dans ce tutoriel, nous allons utiliser un nouveau dossier `djangogirls` que vous allez créer dans votre dossier home :

```
mkdir djangogirls
cd djangogirls
```

Nous allons créer un `virtualenv` appelé `myvenv`. Pour cela, nous taperons une commande qui ressemblera à :

```
python3 -m venv myvenv
```

Windows

Afin de créer un nouveau `virtualenv`, vous avez besoin d'ouvrir votre console (nous en avons déjà parlé dans un chapitre précédent. Est-ce que vous vous en souvenez ?) et tapez `C:\Python34\python -m venv myvenv`. Ça ressemblera à ça :

```
C:\Utilisateurs\Nom\djangogirls> C:\Python34\python -m venv myvenv
```

`C:\Python34\python` doit être le nom du dossier où vous avez installé Python et `myvenv` doit être le nom de votre `virtualenv`. Vous pouvez choisir un autre nom mais attention : il doit être en minuscules, sans espaces et sans accents ou caractères spéciaux. C'est aussi une bonne idée de choisir un nom plutôt court, car vous allez souvent l'utiliser !

Linux et OS X

Pour créer un `virtualenv` sous Linux ou OS X, tapez simplement la commande `python3 -m venv myvenv`. Ça ressemblera à ça :

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

`myvenv` est le nom de votre `virtualenv`. Vous pouvez choisir un autre nom, mais veillez à n'utiliser que des minuscules et à n'insérer ni espaces, ni caractères spéciaux. C'est aussi une bonne idée de choisir un nom plutôt court, car vous aller souvent l'utiliser!

NOTE: initialiser un environnement virtuel sous Ubuntu 14.04 de cette manière donne l'erreur suivante :

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

Pour résoudre ce problème, utilisez plutôt la commande `virtualenv`.

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

Travailler avec `virtualenv`

Les commandes listées ci-dessus permettent de créer un dossier appelé `myvenv` (ou le nom que vous avez choisi) qui contient notre environnement virtuel. Pour faire simple, c'est un dossier composé lui-même d'autres dossiers et de fichiers.

Windows

Démarrez votre environnement virtuel en exécutant :

```
C:\Utilisateurs\Nom\djangogirls> myenv\Scripts\activate
```

Linux et OS X

Démarrez votre environnement virtuel en exécutant :

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

N'oubliez pas de remplacer `myenv` par le nom que vous avez choisi pour votre `virtualenv` (le cas échéant) !

NOTE : il arrive parfois que `source` ne soit pas disponible. Dans ce cas, vous pouvez essayer ceci :

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

Vous saurez que votre `virtualenv` est lancé quand le prompt de votre console ressemblera à ceci :

```
(myenv) C:\Utilisateurs\Nom\djangogirls>
```

ou :

```
(myenv) ~/djangogirls$
```

Vous remarquez que le préfixe `(myenv)` est apparu !

Quand vous travaillez dans un environnement virtuel, la commande `python` fera automatiquement référence à la bonne version de Python. Vous pouvez donc utiliser `python` plutôt que `python3`.

Ok, nous avons installé toutes les dépendances dont nous avions besoin. Nous allons enfin pouvoir installer Django !

Installation de Django

Maintenant que vous avez lancé votre `virtualenv`, vous pouvez installer Django à l'aide de `pip`. Dans votre console, tapez `pip install django~=1.11.0`. Notez bien que nous utilisons un tilde suivi du signe égal : `~=`).

```
(myvenv) ~$ pip install django~=1.11.0
Downloading/unpacking django==1.11
Installing collected packages: django
Successfully installed django
Cleaning up...
```

Sous Windows :

Si jamais vous obtenez des erreurs lorsque vous utilisez pip sous Windows, vérifiez si votre chemin d'accès contient des espaces, des accents ou des caractères spéciaux (ex : `c:\Utilisateurs\Nom d'Utilisateur\djangogirls`). Si c'est le cas, changez de dossier et essayez d'en créer un nouveau en prenant en compte le fait qu'il ne doit donc avoir ni accents, ni espaces, ni caractères spéciaux (ex : `c:\djangogirls`). Après l'avoir déplacé, essayez de retaper la commande précédente.

Sous Linux :

Si vous obtenez une erreur lorsque vous utilisez pip sous Ubuntu 12.04, tapez la commande `python -m pip install -U --force-reinstall pip` pour réparer l'installation de pip dans votre `virtualenv`.

Et voilà ! Vous êtes (enfin) prête pour créer votre première application Django !

Votre premier projet Django !

Note : ce chapitre est en partie inspiré d'un autre tutoriel réalisé par les Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Des morceaux de ce chapitre sont inspirés du [tutoriel django-marcador](#), disponible sous licence Creative Commons Attribution-ShareAlike 4.0 International. Le tutoriel django-marcador a été créé par Markus Zapke-Gründemann et al.

Nous allons créer un petit blog !

La première étape consiste à démarrer un nouveau projet Django. En gros, cela veut dire que nous allons lancer quelques scripts fournis par Django qui vont créer un squelette de projet Django. Il s'agit de fichiers et de dossiers que nous utiliserons par la suite.

Il existe certains fichiers et dossiers dont les noms sont extrêmement importants pour Django. Il ne faut pas renommer les fichiers que nous sommes sur le point de créer. Ce n'est pas non plus une bonne idée de les déplacer. Django a besoin de maintenir une certaine structure pour retrouver les éléments importants.

N'oubliez pas de tout exécuter dans votre virtualenv. Si vous ne voyez pas le préfixe `(myvenv)` dans votre console, vous avez besoin d'activer votre virtualenv. Nous vous avons expliqué comment faire ça dans le chapitre [Installation de Django](#), dans la partie [Travailler avec virtualenv](#). Tapez `myvenv\Scripts\activate` dans votre console Windows ou `source myvenv/bin/activate` dans celle de Mac OS ou Linux afin d'activer votre virtualenv.

Retournons à la création de notre premier projet. Tapez la commande suivante dans votre console MacOS ou Linux.

N'oubliez pas le point `.` à la fin :

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

Pour les utilisatrices de Windows, tapez la commande suivante. **N'oubliez pas le point `.` à la fin :**

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

Le point `.` est très important : c'est lui qui permet de dire au script d'installer Django dans votre répertoire courant (le point `.` est une référence abrégée à celui-ci).

Note : lorsque vous tapez les commandes précédentes dans votre console, vous ne devez recopier que la partie qui commence par `django-admin` OU `django-admin.py`. Les `(myvenv) ~/djangogirls$` et `(myvenv) C:\Users\Name\djangogirls>` du tutoriel sont là pour vous rappeler que ces commandes doivent être tapées dans votre console.

`django-admin.py` est un script qui crée les dossiers et fichiers nécessaires pour vous. Vous devriez maintenant avoir une structure de dossier qui ressemble à celle-ci:

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

`manage.py` est un script qui aide à gérer le site. Il permet notamment de lancer un serveur web sur notre ordinateur sans rien installer d'autre.

Le fichier `settings.py` contient la configuration de votre site web.

Vous vous souvenez de l'histoire du postier qui livre des lettres ? `urls.py` contient une liste de patterns d'urls utilisés par `urlresolver`.

Ignorons les autres fichiers pour l'instant, nous n'allons pas avoir besoin d'y toucher. La seule chose à retenir est qu'il ne faut pas les supprimer par accident !

Changer la configuration

Apportons quelques changements à `mysite/settings.py`. Ouvrez le fichier avec l'éditeur de code que vous avez installé tout à l'heure.

Ça serait sympa d'avoir l'heure correcte sur notre site Web. Allez sur [wikipedia timezones list](#) et copiez le fuseau horaire qui correspond le mieux à l'endroit où vous vous trouvez (TZ). (par exemple: `Europe/Paris`)

Dans `settings.py`, recherchez la ligne qui contient le `TIME_ZONE` et modifiez-la pour choisir votre propre fuseau horaire :

```
TIME_ZONE = 'Europe/Paris'
```

En remplaçant "Europe/Paris" par la valeur appropriée

Nous allons aussi devoir ajouter un chemin d'accès pour les fichiers statiques (nous en apprendrons plus sur les fichiers statiques et CSS plus tard dans le tutoriel). Allez jusqu'à la *fin* du fichier et juste en dessous de la ligne `STATIC_URL`, ajoutez-en une nouvelle avec `STATIC_ROOT` :

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

Quand `DEBUG` est à `True` et `ALLOWED_HOSTS` est vide, les ordinateurs qui sont autorisés à servir notre site web sont ceux de la liste `['localhost', '127.0.0.1', '[::1]']`. Plus loin dans ce tutorial, nous allons déployer le site web sur PythonAnywhere, donc nous allons anticiper un peu et ajouter le nom de domaine correspondant :

```
ALLOWED_HOSTS = ['127.0.0.1', '<your_username>.pythonanywhere.com']
```

Configuration de la base de données

Il existe tout un tas de systèmes de gestion de bases de données qu'il est possible d'utiliser pour stocker les données de votre site. Nous allons utiliser celui par défaut : `sqlite3`.

Il est déjà configuré dans cette partie de votre fichier `mysite/settings.py` :

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

Pour créer la base de donnée de notre blog, il faut lancer la commande suivante dans la console : `python manage.py migrate` (vous avez besoin d'être dans le dossier `djangogirls` qui contient le fichier `manage.py`). Si tout se passe bien, vous devriez voir quelque chose comme ça:

```
(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
```

Et voilà ! Il ne reste plus qu'à lancer le serveur et voir si notre site web fonctionne !

Pour cela, vous avez besoin d'être dans le dossier qui contient le fichier `manage.py` (le dossier `djangogirls`). Dans votre console, vous pouvez lancer le serveur en tapant `python manage.py runserver` :

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

Si vous utilisez Windows et que vous obtenez l'erreur `UnicodeDecodeError`, tapez plutôt cette commande :

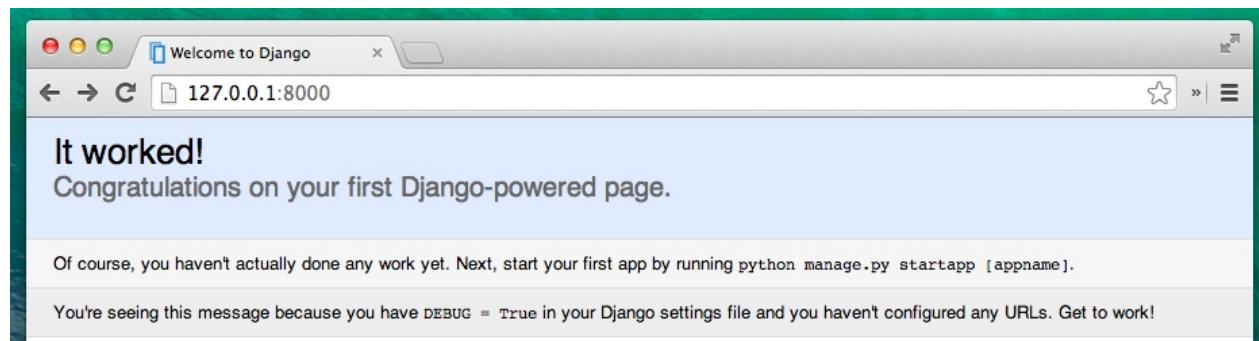
```
(myvenv) ~/djangogirls$ python manage.py runserver 0:8000
```

Ensuite, il ne nous reste plus qu'à vérifier que votre site fonctionne. Pour cela, ouvrez votre navigateur (Firefox, Chrome, Safari, Internet Explorer, ou n'importe quel autre), et entrez l'adresse suivante :

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

Tant qu'il sera lancé, le serveur web va monopoliser votre console. Pour pouvoir taper de nouvelles commandes pendant que le serveur tourne, ouvrez une nouvelle console et activez à nouveau votre virtualenv. Pour arrêter votre serveur web, retournez dans la console où il se trouve et appuyez sur CTRL+C : maintenez les boutons Control et C enfouis en même temps. Sous Windows, vous devrez peut-être appuyer sur CTRL+Arrêt défil.

Bravo ! Vous venez de créer votre premier site web, et de le lancer avec un serveur web ! C'est génial, non ?



Prête pour la suite ? Il est temps de créer du contenu !

Les modèles dans Django

Maintenant, nous aimerais créer quelque chose qui permet de stocker les articles de notre blog. Mais avant de pouvoir faire ça, nous allons tout d'abord devoir vous parler d'un truc qui s'appelle les `objets`.

Les objets

Il existe un concept en programmation qu'on appelle la `programmation orientée objets`. L'idée, c'est de modéliser les choses et de définir comment elles interagissent entre elles plutôt que de tout voir comme une séquence d'instructions.

Du coup, c'est quoi un objet ? C'est une collection de propriétés et d'actions. Ça a l'air bizarre dit comme ça. Un exemple devrait vous permettre d'y voir un peu plus clair.

Si on veut modéliser un chat, nous allons créer un objet `Chat` qui a quelques propriétés comme `couleur`, `age`, `humeur` (bonne humeur, mauvaise humeur, fatigué ;)). Il peut aussi avoir un `propriétaire` (un objet `Personne`), mais ce n'est pas obligatoire : cette propriété pourrait être vide dans le cas d'un chat sauvage.

Ensuite, nous pouvons donner des actions au `Chat` : `ronronner`, `gratter` OU `manger`. (Dans ce dernier cas, on donne au chat un objet `NourriturePourChat`, qui peut lui aussi avoir ses propres propriétés, comme le `goût`).

```
Chat
-----
couleur
age
humeur
propriétaire
ronronner()
gratter()
nourrir(nourriture_pour_chat)

NourriturePourChat
-----
gout
```

L'idée qu'il faut retenir, c'est que l'on décrit les choses du monde réel avec des propriétés (appelées `propriétés des objets`) et des actions (appelées `méthodes`).

Du coup, comment modéliser les articles de blog ? C'est bien gentil les chats, mais ce qui nous intéresse, ça reste de faire un blog !

Pour ça, il faut répondre à la question : qu'est-ce qu'un article de blog ? Quelles propriétés devrait-il avoir ?

Pour commencer, notre blog post doit avoir du texte : il a bien du contenu et un titre, n'est-ce pas ? Et puis, ce serait bien de savoir aussi qui l'a écrit. On a donc besoin d'un auteur. Enfin, on aimerait aussi savoir quand l'article a été écrit et publié.

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

Quel genre d'actions pourrions-nous faire sur un article de blog ? Un bon début serait d'avoir une `méthode` qui permet de publier le post.

On va donc avoir besoin d'une méthode `publish`.

Voilà, nous avons une idée de ce que nous avons besoin. Allons modéliser tout ça dans Django!

Les modèles dans Django

Maintenant que nous savons ce qu'est un objet, nous allons pouvoir créer un modèle Django pour notre post de blog.

Un modèle Django est un type particulier d'objet : il est sauvegardé dans la `database`. Une base de données est une collection de données. C'est à cet endroit que l'on stocke toutes les informations au sujet des utilisateurs, des blog posts, etc. Pour stocker nos données, nous allons utiliser une base de données SQLite. C'est la base de données par défaut dans Django. Elle sera largement suffisante pour ce que nous voulons faire.

Pour vous aider à visualiser ce qu'est une base de données, pensez à un tableur avec des colonnes (champs) et des lignes (données).

Créer une application

Pour éviter le désordre, nous allons créer une application séparée à l'intérieur de notre projet. Prenez l'habitude de bien tout organiser dès le début. Afin de créer une application, nous avons besoin d'exécuter la commande suivante dans notre console (prenez garde à bien être dans le dossier `djangogirls` où se trouve le fichier `manage.py`) :

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

Vous pouvez voir qu'un nouveau dossier `blog` a été créé et qu'il contient différents fichiers. Vos dossiers et fichiers liés à votre projet doivent maintenant être organisés selon cette structure :

```
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
```

Après avoir créé une nouvelle application, vous devez dire à Django de l'utiliser. Pour cela, nous allons éditer le fichier `mysite/settings.py`. Trouvez la section `INSTALLED_APPS` et ajoutez `'blog'`, juste avant `]`. La section doit maintenant ressembler à ceci :

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

Créer un modèle de blog post

Le fichier `blog/models.py` permet de définir les objets que nous appelons des `modèles`. C'est à cet endroit que nous allons définir ce qu'est un blog post. Pour éviter tout problème (les caractères accentués par exemple!), nous allons garder les termes en anglais.

Ouvrez le fichier `blog/models.py`, supprimez tout ce qui s'y trouve et copiez-y le morceau de code suivant :

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

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

Vérifiez que vous avez bien utilisé deux tirets bas (`_`) autour de `str`. C'est une convention fréquemment utilisée en Python qui porte même un petit nom en anglais : "dunder", pour "double-underscore".

Ce gros morceau de code a l'air effrayant mais, ne vous inquiétez pas : nous allons vous expliquer ce que signifie chacune de ces lignes!

Toutes les lignes qui commencent par `from` ou `import` sont des lignes qui permettent d'importer des morceaux d'autres fichiers. Concrètement, au lieu de recopier ou de copier-coller la même chose dans différents fichiers, nous pouvons tout simplement faire référence à certains morceaux d'autres fichiers à l'aide de `from ... import ...`.

`class Post(models.Model):` - C'est cette ligne qui permet de définir notre modèle. C'est un `object`.

- Le mot clef spécial `class` permet d'indiquer que nous sommes en train de définir un objet.
- `Post` est le nom de notre modèle. Vous pouvez lui donner un autre nom mais vous ne pouvez pas utiliser de caractères spéciaux ou accentués ni insérer des espaces. Le nom d'une classe commence toujours par une majuscule.
- `models.Model` signifie que Post est un modèle Django. Comme ça, Django sait qu'il doit l'enregistrer dans la base de données.

Maintenant, nous allons pouvoir définir les propriétés dont nous parlions au début de ce chapitre : `title (titre)`, `text (texte)`, `created_date (date de création)`, `published_date (date de publication)` et `author (auteur)`. Pour cela, nous allons avoir besoin de définir le type de chaque champ (Est-ce que c'est du texte? Un nombre? Une date? Une relation à un autre objet, un utilisateur par exemple?).

- `models.CharField` - Cela nous permet de définir un champ texte avec un nombre limité de caractères.
- `models.TextField` - Cela nous permet de définir un champ texte sans limite de caractères. Parfait pour le contenu d'un blog post !
- `models.DateTimeField` - Définit que le champ en question est une date ou une heure.
- `models.ForeignKey` - C'est un lien vers un autre modèle.

Malheureusement, nous n'avons pas le temps de vous expliquer tous les bouts de code que nous allons manipuler dans ce tutoriel. Si vous voulez en savoir un peu plus sur les différents champs disponibles dans les modèles ou que vous aimerez définir quelque chose qui n'est pas listé dans les exemples ci-dessus, n'hésitez pas à consulter la documentation de Django (<https://docs.djangoproject.com/fr/1.11/ref/models/fields/#field-types>).

Et sinon, c'est quoi `def publish(self):` ? Il s'agit de notre méthode `publish` dont nous parlions tout à l'heure. `def` signifie que nous créons une fonction/méthode qui porte le nom `publish`. Vous pouvez changer le nom de la méthode si vous le souhaitez. N'oubliez pas les règles de nommage et pensez à utiliser des minuscules et des tirets bas à la place des espaces. Par exemple, une méthode qui calcule le prix moyen d'un produit pourrait s'appeler `calcul_prix_moyen`.

Les méthodes renvoient (`return`) souvent quelque chose. C'est le cas de la méthode `__str__`. Dans notre tutoriel, lorsque nous appellerons la méthode `__str__()`, nous allons obtenir du texte (**string**) avec un titre de Post.

Si quelque chose ne vous paraît pas clair au sujet des modèles, n'hésitez pas à demander à votre coach ! Cela peut être compliqué à comprendre la première fois, surtout lorsque l'on apprend les objets et les fonctions en même temps. Gardez espoir ! Avec le temps, tout cela vous paraîtra de moins en moins magique et de plus en plus évident !

Créer des tables pour votre modèle dans votre base de données

La dernière étape pour cette section est d'ajouter notre nouveau modèle à notre base de données. Tout d'abord, nous devons signaler à Django que nous venons de créer notre modèle. Tapez `python manage.py makemigrations blog` dans votre console. Le résultat devrait ressembler à ça :

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

Django vient de nous préparer un fichier de migration que nous allons pouvoir appliquer dès maintenant à notre base de données. Pour cela, tapez `python manage.py migrate blog`. Normalement, vous devrez voir ceci s'afficher dans votre console :

```
(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
```

Youpi ! Notre modèle Post est maintenant intégré à la base de données. Ce serait cool de voir à quoi il ressemble réellement ! Pour ça, il va falloir attaquer la section suivante ! Au boulot ;)!

L'interface d'administration de Django

Pour ajouter, éditer et supprimer les posts que nous venons de modéliser, nous allons utiliser l'interface d'administration de Django.

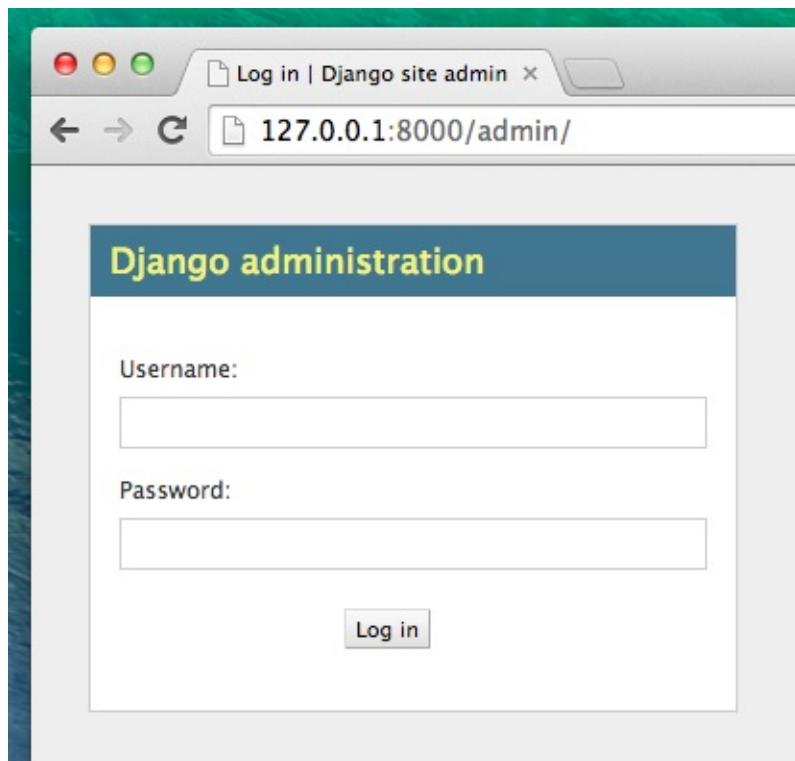
Ouvrons le fichier `blog/admin.py` et remplaçons son contenu par ceci :

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

admin.site.register(Post)
```

Comme vous pouvez le voir, nous importons le modèle « Post » que nous avons écrit dans le chapitre précédent. Afin que notre modèle soit visible dans l'interface d'administration, nous avons besoin d'enregistrer notre modèle à l'aide de `admin.site.register(Post)`.

Voilà, il est temps de jeter un œil à notre modèle Post. N'oubliez pas d'exécuter `python manage.py runserver` dans votre console afin de lancer le serveur web. Dans votre navigateur, entrez l'adresse <http://127.0.0.1:8000/admin/>. Si tout va bien, vous verrez une page comme celle-ci :



Afin de vous connecter, vous allez devoir créer un *superuser*, c'est à dire un utilisateur qui contrôlera l'intégralité du site. Retournez à votre ligne de commande : tapez `python manage.py createsuperuser` puis appuyez sur entrée. Tapez votre nom d'utilisateur (en minuscules, sans espace), votre email et votre mot de passe. Vous ne voyez pas le mot de passe que vous êtes en train de taper ? C'est tout à fait normal, ne vous inquiétez pas ! Tapez simplement votre mot de passe et appuyez sur entrée pour continuer. La sortie que vous allez obtenir doit ressembler à celle-ci. Notez que le nom d'utilisateur et l'adresse email doivent correspondre aux informations que vous venez d'entrer.

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

Retournez dans votre navigateur et connectez-vous en tant que superutilisateur grâce à l'utilisateur que vous venez de créer. Vous devriez accéder à l'interface d'administration de Django.

The screenshot shows the Django administration interface. At the top, it says "Site administration | Django" and the URL "127.0.0.1:8000/admin/". Below that is the "Django administration" header. The main area is titled "Site administration" and shows a list of models: "Auth", "Groups", "Users", "Blog", and "Posts". For each model, there are "Add" and "Change" buttons. The "Blog" section is currently selected.

Allez voir vos posts et expérimenter un peu avec. Ajoutez 5 ou 6 posts. Ne vous occupez pas du contenu - vous pouvez juste copier du texte de ce tutoriel pour aller plus vite :).

Sur vos 5 posts, faites en sorte qu'au moins 2 ou 3 posts possèdent une date de publication. Prenez garde à ne pas en donner à tous vos posts car nous allons avoir besoin de ces deux différents cas plus tard.

The screenshot shows the "Add post" form in the Django administration interface. The title is "Add post". The form fields are: "Author" (set to "olasitarska"), "Title" (text "Cras justo odio, dapibus ac facilisis in, eu"), "Text" (text block containing placeholder text), "Created date" (date "2014-07-07" and time "23:07:34"), and "Published date" (date "2014-07-07" and time "23:07:34"). At the bottom, there are buttons for "Save and add another", "Save and continue editing", and "Save".

Si jamais vous voulez en savoir un peu plus sur l'interface d'administration de Django, vous pouvez lire la documentation qui lui est associée : <https://docs.djangoproject.com/fr/1.11/ref/contrib/admin/>

Il est probablement temps d'aller recharger vos batteries avec un café, un thé ou un truc à grignoter. Vous venez de créer votre modèle Django : vous méritez bien une petite pause !

Déployer !

Le chapitre suivant peut-être un peu difficile à comprendre. Accrochez-vous et allez jusqu'au bout : le déploiement fait partie intégrale du processus de développement d'un site internet. Ce chapitre a été placé au milieu du tutoriel afin de permettre à votre coach de vous aider dans cette étape un peu compliquée qu'est la mise en ligne de votre site. Si jamais vous manquez de temps à la fin de la journée, ne vous inquiétez pas ! Une fois ce chapitre terminé, vous serez en mesure de finir le tutoriel chez vous :)

Jusqu'à présent, votre site web n'était seulement disponible que sur votre ordinateur. Maintenant, vous allez apprendre à le déployer ! Déployer signifie mettre en ligne votre site pour que d'autres personnes puissent enfin voir votre app :).

Comme vous l'avez appris, un site web a besoin d'être installé sur un serveur. Il existe de très nombreux fournisseurs de serveurs sur Internet. Nous allons en utiliser un qui dispose d'un système de déploiement relativement simple : [PythonAnywhere](#). PythonAnywhere est gratuit pour les petites applications qui n'ont pas beaucoup de visiteurs : cela correspond parfaitement à ce dont nous avons besoin pour le moment.

Nous allons aussi utiliser les services [GitHub](#), ce qui nous permettra d'héberger notre code en ligne. Il existe d'autres entreprises qui proposent des services similaires. Cependant, presque tous·tes les développeurs·ses possèdent aujourd'hui un compte GitHub et, dans quelques instants, vous aussi !

GitHub va nous servir d'intermédiaire pour envoyer et récupérer notre code sur PythonAnywhere.

Git

Git est un "gestionnaire de version" utilisé par de nombreux·ses développeurs·ses. Ce logiciel permet de garder une trace des modifications apportées à chaque fichier afin que vous puissiez facilement revenir en arrière ou à une version spécifique. Cette fonction est similaire au "suivi des modifications" de Microsoft Word, mais en beaucoup plus puissant.

Installer Git

Si vous avez suivi la partie "installation" du tutoriel, vous n'avez pas besoin d'installer Git à nouveau. Vous pouvez passer directement à la prochaine section et commencer à créer votre dépôt Git.

Windows

Vous pouvez télécharger Git sur [git-scm.com](#). Vous pouvez cliquer sur "next" à toutes les étapes, sauf pour la cinquième, "Adjusting your PATH environment" : n'oubliez pas de choisir "Run Git and associated Unix tools from the Windows command-line", situé en bas de la liste des options disponibles. Les autres choix par défaut n'ont pas besoin d'être modifiés. L'option "Checkout Windows-style, commit Unix-style line endings" est parfaite: vous n'avez rien à changer sur cette page.

MacOS

Vous pouvez télécharger Git sur [git-scm.com](#). Pour le reste de l'installation, suivez simplement les instructions de l'installateur.

Linux

Git est probablement déjà installé mais, si ce n'est pas le cas, voici les instructions à suivre :

```
sudo apt install git
# ou
sudo yum install git
# ou
sudo zypper install git
```

Démarrer un dépôt Git

Git conserve toutes les modifications apportées à un ensemble de fichiers dans un "repository" (ou "dépôt"). Nous allons devoir en créer un pour notre projet. Ouvrez votre terminal et allez dans le répertoire `djangogirls`. Ensuite, tapez les commandes suivantes :

: n'oubliez pas de vérifier dans quel répertoire vous vous trouvez avant d'initialiser votre dépôt. Pour cela tapez la commande `pwd` (OSX/Linux) ou `cd` (Windows). Vous devriez vous trouver dans le dossier `djangogirls`.

```
$ git init
Initialise un dépôt Git vide à l'emplacement ~/djangogirls/.git/
$ git config --global user.name "Votre nom"
$ git config --global user.email you@example.com
```

L'initialisation d'un dépôt git ne se fait qu'une fois par projet. De même, vous n'aurez plus jamais à ré-entrer votre nom d'utilisateur ou votre email.

Git va surveiller et conserver les modifications concernant l'ensemble des fichiers et dossiers présents dans ce répertoire, à l'exception de certains fichiers que nous aimerions exclure. Pour cela, nous allons créer un fichier appelé `.gitignore` dans le répertoire principal du projet. Ouvrez votre éditeur et créez un nouveau fichier en copiant le contenu suivant :

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

Enregistrez ce fichier `.gitignore` dans votre répertoire principal "djangogirls".

Attention: le point au début du nom du fichier est important ! Vous pouvez parfois rencontrer des difficultés à créer ce fichier. Par exemple, Mac ne vous laisse pas enregistrer un fichier qui commence par un point dans Finder. Pour contourner ce problème, utilisez la fonction "enregistrer sous" de votre éditeur : ça marche à tous les coups!

Note L'un des fichiers spécifiés dans `.gitignore` est `db.sqlite3`. Ce fichier est votre base de donnée locale, où tous vos posts de blog sont stockés. Ce fichier n'est pas ajouté à la base de données parce que votre site internet sur PythonAnywhere utilise une base de donnée différente. Cette base de donnée peut être en SQLite, comme celle créée localement sur votre ordinateur mais en général, une base de données appelée MySQL est utilisée parce qu'elle peut gérer beaucoup plus de visiteurs qu'une base de données SQLite. Dans tous les cas, ignorer votre base de données SQLite pour la copie sur GitHub signifie que tous les posts que vous avez créé jusqu'à maintenant vont rester sur votre machine locale et ne seront accessible que depuis cette machine. Vous allez devoir les ajouter à nouveau sur votre site internet en production. Considérez votre base de données locale comme une aire de jeu où vous pouvez essayer des choses différentes sans vous soucier de supprimer un de vos vrais post de blog.

Avant de taper la commande `git add` ou lorsque vous ne vous souvenez plus des changements que vous avez effectué dans votre projet, pensez à taper la commande `git status`. Cela permet surtout d'éviter les mauvaises surprises, comme l'ajout ou l'envoi d'un mauvais fichier. La commande `git status` permet d'obtenir des informations sur tous les fichiers non-suivis/modifiés/mis-à-jour, l'état de la branche, et bien plus encore. Voici ce qui se passe lorsque vous tapez cette commande :

```
$ 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)
```

Pour le moment, nous n'avons fait que regarder l'état de notre branche. Pour enregistrer nos changements, nous allons devoir taper les commandes suivantes :

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

Publier votre code sur GitHub

Allez sur GitHub.com et inscrivez-vous gratuitement (si vous possédez déjà un compte, c'est très bien!)

Ensuite, créez un nouveau dépôt en lui donnant le nom "my-first-blog". Pensez à laisser les options par défaut. Dans notre cas, il n'est pas nécessaire de cocher la case "initialise with a README". Vous pouvez aussi laisser l'option .gitignore vide car nous avons déjà créé ce fichier précédemment. Enfin, comme nous n'avons pas besoin d'une licence pour notre application, laissez le champ License à None.

Owner: hjwp / Repository name: my-first-blog

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

Description (optional):

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

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

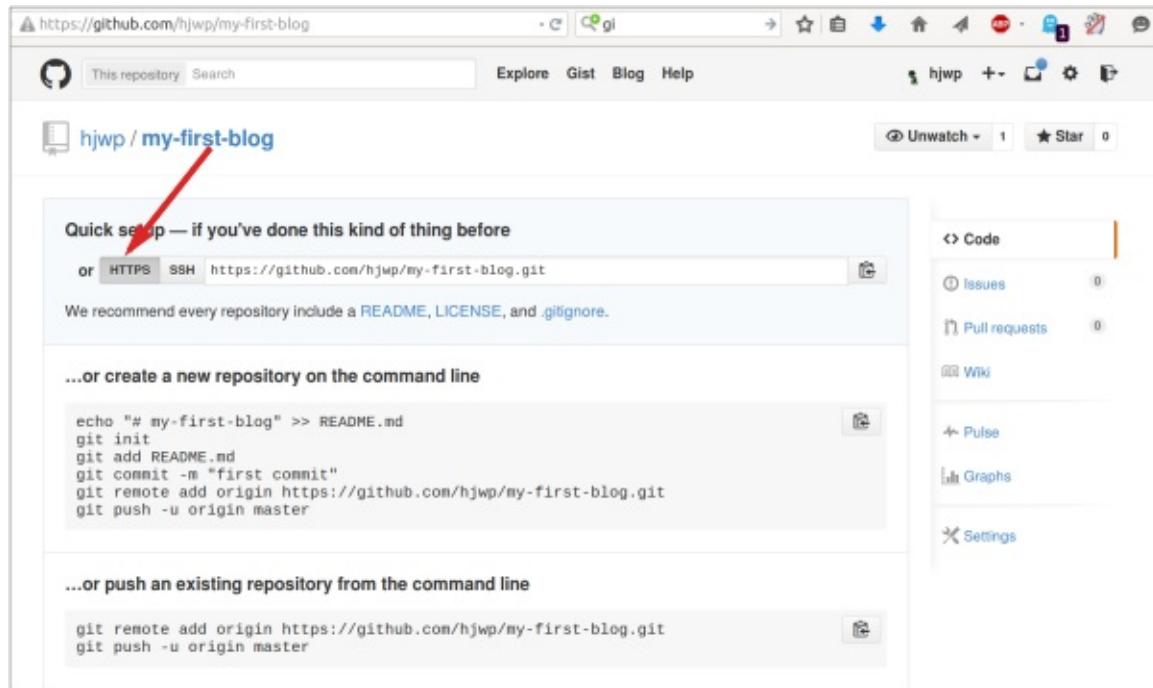
Initialize this repository with a README: This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add .gitignore: [None](#) Add a license: [None](#)

[Create repository](#)

: dans le cadre de ce tutoriel, le nom `my-first-blog` est très important. Cependant, vous êtes libre de le changer mais, attention : à chaque fois que ce nom apparaitra dans le tutoriel, vous allez devoir le substituer avec le nom que vous avez choisi. Il est probablement plus simple de garder le nom `my-first-blog` pour cette fois.

La page suivante vous donne l'URL qui va vous permettre de cloner votre dépôt. Choisissez la version « HTTPS » et copiez l'URL car nous allons rapidement en avoir besoin :



Nous avons maintenant besoin de relier nos deux dépôts : celui sur notre ordinateur et celui sur GitHub. On utilise l'expression "hook" en anglais pour cette décrire cette étape.

Tapez les instructions suivantes dans votre console (remplacez `<votre-nom-d'utilisateur-github>` avec le nom d'utilisateur de votre compte GitHub et sans les chevrons) :

```
$ git remote add origin https://github.com/<votre-nom-d'utilisateur-github>/my-first-blog.git  
$ git push -u origin master
```

Entrez votre nom d'utilisateur et mot de passe GitHub. Vous devriez voir quelque chose comme ceci :

```
Username for 'https://github.com': votre-nom  
Password for 'https://votre-nom@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/votre-nom/my-first-blog.git  
 * [new branch]      master -> master  
Branch master set up to track remote branch master from origin.
```

Votre code est maintenant sur GitHub. Allez jeter un coup d'œil ! Votre code est maintenant au même endroit que d'autres projets super cool : [Django](#), le tutoriel [Django Girls](#) et les nombreux autres projets libres qui sont hébergés sur GitHub :)

Mettre votre blog en ligne avec PythonAnywhere

Vous avez peut être déjà créé un compte PythonAnywhere au cours de la phase d'installation - si c'est le cas, inutile de le refaire.

Si vous ne l'avez pas encore fait, n'oubliez pas de vous créer un compte "Beginner" sur PythonAnywhere.

- www.pythonanywhere.com

: Le nom d'utilisateur que vous allez choisir va déterminer l'adresse de votre blog de la manière suivante : `votrenomdutilisateur.pythonanywhere.com`. Si vous ne savez pas quoi prendre, nous vous conseillons de choisir votre surnom ou un nom proche du sujet de votre blog.

Récupérer votre code sur PythonAnywhere

Une fois enregistré sur PythonAnywhere, vous serez automatiquement redirigée sur votre écran d'accueil où se trouve la liste des consoles. Cliquez sur le lien "Bash" dans la partie "start a new console". C'est la version PythonAnywhere des consoles que vous avez sur votre ordinateur.

: PythonAnywhere utilise Linux. Si vous êtes sous Windows, la console sera un peu différente de celle de votre ordinateur.

Importons notre code depuis GitHub vers PythonAnywhere en créant un "clone" de notre dépôt. Tapez la commande suivante dans la console de PythonAnywhere (n'oubliez pas d'utiliser votre nom d'utilisateur GitHub à la place de `<your-github-username>`):

```
$ git clone https://github.com/<votre-nom-d'utilisateur-github>/my-first-blog.git
```

Cette commande va permettre d'effectuer une copie de votre code vers PythonAnywhere. La commande `tree my-first-blog` permet d'afficher un aperçu de ce qui se trouve maintenant sur votre serveur :

```
$ 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
```

Créer un virtualenv sur PythonAnywhere

Tout comme sur votre ordinateur, vous allez devoir créer un environnement virtuel et installer Django sur PythonAnywhere. L'opération est identique, à une différence près pour les utilisatrices de Windows : il s'agit ici d'une console Linux. Pas de panique, c'est très simple ! Ouvrez la console Bash de PythonAnywhere et tapez les commandes suivantes :

```
$ 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~=1.11.0
Collecting django
[...]
Successfully installed django-1.11
```

: L'étape `pip install` peut prendre quelques minutes. Patience, patience ! Cependant, si cela prend plus de 5 minutes, c'est que quelque chose ne va pas. N'hésitez pas à solliciter votre coach.

Créer une base de données sur PythonAnywhere

Tout comme l'environnement virtuel, la base de données n'est pas partagée entre le serveur et votre ordinateur. Cela signifie, entre autre, que vous n'aurez plus forcément les mêmes utilisateurs ni les mêmes posts sur votre ordinateur et sur PythonAnywhere.

Pour créer une base de données sur PythonAnywhere, nous allons taper les mêmes commandes que sur notre ordinateur: d'abord `migrate`, puis `createsuperuser`:

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

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

Faire de votre blog une application web

Maintenant, notre code est sur PythonAnywhere, notre virtualenv est prêt et la base de données est initialisée. Nous sommes prêts à le publier comme une application web !

Retourner sur la page d'accueil de PythonAnywhere en cliquant sur le logo en haut à gauche. Ensuite, cliquez sur l'onglet **Web et Add a new web app**.

Après avoir confirmé votre nom de domaine, choisissez **manual configuration** dans la boîte de dialogue (NB : ne choisissez *pas* l'option "Django"). Enfin, choisissez **Python 3.4** et cliquez sur "next" pour fermer l'assistant de configuration.

: Faites bien attention à sélectionner l'option configuration manuelle ("Manual configuration") et non l'option "Django". N'oubliez pas une chose : vous êtes bien trop cool pour prendre l'option Django qui est fourni par défaut ;-)

Configurer le virtualenv

Une fois l'assistant fermé, vous serez automatiquement conduite sur la page de configuration dédiée à votre application web. Dès que vous aurez besoin de modifier quelque chose concernant votre appli, c'est là que vous devrez aller.

The screenshot shows the PythonAnywhere web interface. At the top, there's a navigation bar with tabs: Consoles, Files, Web (which is selected), Schedule, and Databases. Below the navigation bar, it says "Configuration for edith.pythonanywhere.com". There's a button to "Add a new web app". Under the "Actions" section, there are two buttons: "Reload edith.pythonanywhere.com" (green) and "Delete edith.pythonanywhere.com" (red). A red arrow points to the "Virtualenv:" section. In the "Code" section, there's information about the source code, WSGI configuration file, and Python version. The WSGI configuration file is listed as "/var/www/edith_pythonanywhere_com_wsgi.py". A red arrow also points to the "Virtualenv:" section, which contains a text input field with the value "/home/edith/my-first-blog/myvenv" and a blue checkmark button next to it.

Dans la section "Virtualenv", cliquez sur le texte en rouge qui indique "Enter the path to a virtualenv" (entrer le chemin d'accès de votre environnement virtuel), et entrez ceci : `/home/<your-username>/my-first-blog/myvenv/`. Cliquez sur la boîte bleue avec la case à cocher pour sauvegarder le chemin d'accès.

: N'oubliez pas de mettre votre nom d'utilisateur. Ne vous inquiétez pas : si vous faites une erreur, PythonAnywhere vous le signalera.

Configurer le fichier WSGI

Django utilise le protocole "WSGI" qui est un standard pour servir des sites web qui utilisent Python. Ce protocole est supporté par PythonAnywhere. Afin que PythonAnywhere détecte notre blog Django, nous allons éditer le fichier de configuration WSGI.

Dans l'onglet web, vous allez trouver une section code : cliquez sur le lien "WSGI configuration file" : votre fichier de configuration devrait s'intituler `/var/www/<your-username>_pythonanywhere_com_wsgi.py`. Une fois que vous avez cliqué dessus, vous serez automatiquement redirigée dans un éditeur de texte.

Supprimer le contenu du fichier et le remplacer par ce qui suit :

```
import os
import sys

path = '/home/<your-username>/my-first-blog' # use your own username here
if path not in sys.path:
    sys.path.append(path)

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

from django.core.wsgi import get_wsgi_application
from django.contrib.staticfiles.handlers import StaticFilesHandler
application = StaticFilesHandler(get_wsgi_application())
```

: N'oubliez pas de remplacer `<your-username>` par votre nom d'utilisateur **Note** : A la ligne 3, on s'assure que PythonAnywhere saura trouver notre application. Il est très important que ce chemin d'accès soit correct, et plus particulièrement qu'il n'y ait pas d'espaces en plus. Dans le cas contraire "ImportError" s'affichera dans le log d'erreur.

Le but de ce fichier est de permettre à PythonAnywhere de savoir où votre application web se situe et de connaître le nom des fichiers de configuration de Django.

`StaticFileHandler` sert à gérer notre CSS. Cette étape est réalisée automatiquement en exécutant la commande `runserver`. Nous aborderons un peu plus en détails les fichiers statiques quand nous éditerons le CSS de notre site.

Cliquez sur **Save** puis, retournez dans l'onglet **Web**.

Et voilà, c'est fini ! Vous n'avez plus qu'à cliquer sur le gros bouton vert **Reload** et vous devriez voir votre application en ligne. Le lien vers votre site est situé en haut de l'onglet web de PythonAnywhere.

Conseils en cas de bug

Si vous constatez une erreur lorsque vous essayez de visiter votre site web, les **logs d'erreurs** devraient vous permettre de comprendre ce qui ne marche pas. Vous trouverez un lien vers ces fichiers dans l'onglet **Web** de PythonAnywhere. Regardez s'il y a des messages d'erreurs ; les plus récents seront en bas du fichier. Les bugs les plus fréquents que vous pouvez rencontrer sont les suivants :

- Oublier une étape lors du passage dans la console. Vous devriez avoir fait toutes les étapes suivantes : créer un environnement virtuel, l'activer, installer Django et enfin créer la base de données.
- Se tromper dans le chemin d'accès à l'environnement virtuel : si c'est le cas, vous trouverez un petit message d'erreur en rouge dans l'onglet "web", section `virtualenv`.
- Se tromper lors de la création du fichier de configuration WSGI : pensez à vérifier si le chemin d'accès que vous avez entré est bien celui de "my-first-blog" ?
- Se tromper de version de Python : votre environnement virtuel et votre application web doivent toutes les deux être sous Python 3.4.
- Il y a quelques [astuces générales de débogage sur le wiki PythonAnywhere](#).

Et n'oubliez pas, votre coach est là pour vous aider !

Votre site est en ligne !

La page qui s'affiche devrait être la même que celle sur votre ordinateur : "Welcome to Django". Vous pouvez essayer d'accéder à l'interface d'administration en ajoutant `/admin/` à la fin de l'URL. Normalement, une page de login devrait s'afficher. Une fois connectée en utilisant votre nom d'utilisateur et votre mot de passe, vous devriez pouvoir ajouter des nouveaux posts sur le serveur.

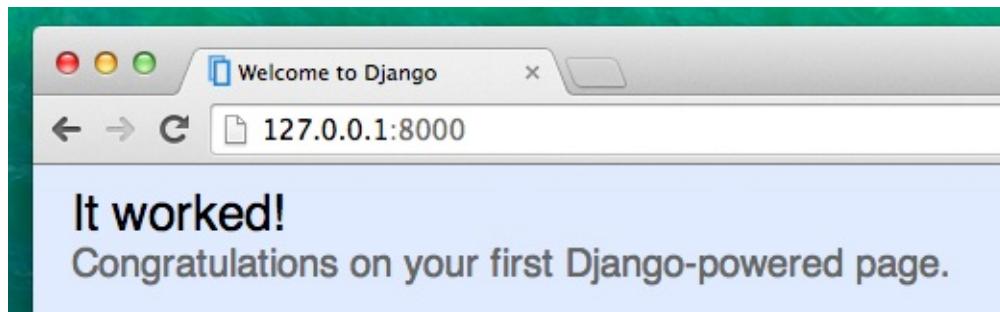
Félicitations ! Le déploiement est l'une des parties les plus épineuses du développement web et il faut souvent plusieurs jours avant d'obtenir quelque chose de fonctionnel. Mais vous avez réussi sans trop d'encombre à mettre votre site en ligne : parfait <3

Les urls Django

Nous sommes sur le point de construire notre première page web : la page d'accueil de notre blog ! Avant de passer à la partie code, apprenons-en un peu plus sur les urls Django.

Qu'est-ce qu'une URL ?

Une URL est simplement une adresse web. Vous pouvez voir une URL à chaque fois que vous visitez un site web: l'URL se trouve dans la barre d'adresse (hé oui! `127.0.0.1:8000` est aussi une URL ! `https://djangogirls.org` est aussi une URL) :



Chaque page internet a besoin de sa propre URL. Cela permet à votre application de savoir ce qu'elle doit afficher à un utilisateur lorsqu'il entre une URL. Dans Django, nous utilisons un outil appelé `URLconf` (configuration des URLs) : c'est un ensemble de patterns que Django va essayer de faire correspondre avec l'URL reçue afin d'afficher la vue correspondante.

Comment les URLs fonctionnent-elles dans Django ?

Ouvrons le fichier `mysite/urls.py` dans notre éditeur de code et regardons à quoi il ressemble :

```
"""mysite URL Configuration

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

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

Comme vous pouvez le voir, Django nous a déjà préparé une partie du travail.

Les lignes encadrées par trois guillemets (`"""` ou `'''`) sont appelées docstrings ; nous pouvons les ajouter au début de nos fichiers, de nos classes ou de nos méthodes pour décrire ce qu'elles font. Ces lignes ne seront donc pas exécutées par Python.

Comme vous pouvez le voir, l'adresse de l'interface d'administration est déjà en place :

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

Cela signifie que pour chaque URL qui commence par `admin/`, Django affichera la vue correspondante. Dans cet exemple, vous pouvez constater que toutes les URLs liées à l'interface d'administration sont contenues dans une seule ligne : en plus d'être pratique, cela rend notre fichier beaucoup plus propre et lisible.

Regex

Vous vous demandez sûrement comment Django arrive à faire correspondre les URLs aux vues correspondantes ? Bon, on respire un grand coup car ça va être un peu complexe. Django utilise des `regex` ("expressions régulières"). Les regex ont beaucoup (vraiment beaucoup !) de règles qui permettent de donner une description de la chaîne de caractères que l'on recherche (pattern). Étant donné que les regex sont un sujet avancé, nous ne rentrerons pas en détail dans leur fonctionnement.

Si vous avez quand-même envie de comprendre comment nous avons créé nos patterns, vous pouvez lire ce qui va suivre. Dans notre exemple, nous allons utiliser un petit sous ensemble des règles disponibles pour écrire des patterns :

```
^ -> le début du texte
$ -> la fin du texte
\d -> un chiffre
+ -> indique que l'expression précédente doit se répéter au moins une fois
() -> capture une partie du pattern
```

Tout ce qui ne fait pas partie de ces règles et qui est présent dans la description de ce que l'on cherche sera interprété de manière littérale.

Maintenant, imaginez que vous avez un site web qui a comme adresse : `http://www.mysite.com/post/12345/ . 12345` désigne le numéro de votre post.

Ce serait vraiment pénible de devoir écrire une vue différente pour chaque post que nous aimerais rédiger. Nous allons créer un pattern qui correspond à cette URL et qui nous permettra aussi d'extraire le numéro de post : `^post/(\d+)/$`. Décomposons-la morceau par morceau pour comprendre ce que nous faisons :

- `^ post /` indique à Django d'attraper toutes les url qui commencent par `post/` (juste après `^`)
- `(\d+)` signifie qu'il y aura un nombre (un ou plusieurs chiffres) que nous souhaitons capturer et extraire
- `/` dit à Django que le caractère `/` doit suivre le nombre
- `$` marque la fin de l'URL, ce qui signifie que seules les chaînes de caractères se terminant par `/` correspondront au pattern

Votre première URL Django !

Bon, il est temps de créer votre première URL ! Nous voulons que "`http://127.0.0.1:8000/`" soit la page d'accueil de notre blog et qu'elle nous montre la liste des articles du blog.

Nous aimerais aussi garder notre fichier `mysite/urls.py` propre. Pour cela, nous allons importer les URLs de notre application `blog` dans notre fichier principal `mysite/urls.py`.

On y va : ajoutez une ligne qui va nous permettre d'importer `blog.urls` dans notre URL principale (`''`). Notez que nous utilisons ici la fonction `include` et qu'il est donc nécessaire de l'ajouter à l'import de la première ligne de notre fichier.

Votre fichier `mysite/urls.py` devrait maintenant ressembler à ceci:

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

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

Django va maintenant rediriger tout ce qui arrive sur "`http://127.0.0.1:8000/`" vers `blog.urls` puis regardera dans ce fichier pour y trouver la suite des instructions à suivre.

En Python, les expressions régulières commencent toujours par `r` au début de la chaîne de caractères. Cela permet d'indiquer à Python que ce qui va suivre inclut des caractères qu'il ne doit pas interpréter en tant que code Python mais en tant qu'expression régulière.

blog.urls

Créez un nouveau fichier vide `blog/urls.py`. OK ! Ajoutez maintenant ces deux premières lignes :

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

Nous venons d'importer les méthodes de Django dont nous avons besoin ainsi que toutes les `vues` liées à notre application `blog`. Cependant, nous n'avons pas encore créé de vues ! Pas de problème : nous y viendrons dans une minute

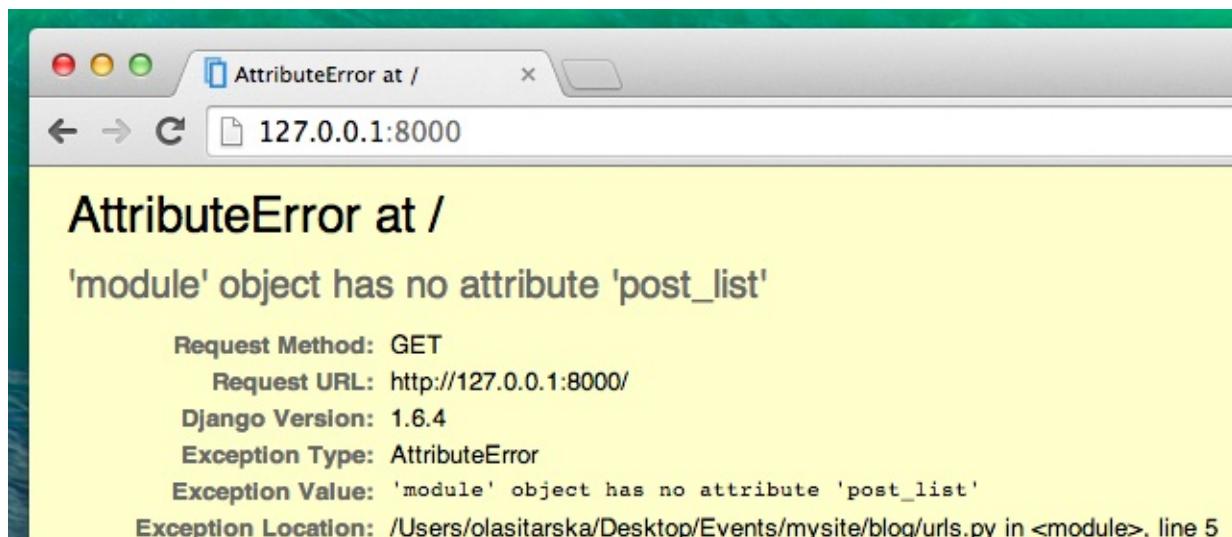
Après ça, nous pouvons ajouter notre premier pattern d'URL:

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

Comme vous pouvez le voir, nous assignons une `vue` appelée `post_list` à l'URL `^$`. Décomposons cette expression régulière : `^` pour début suivie de `$` pour fin. Si nous mettons ces deux symboles ensemble, cela donne l'impression que nous sommes à la recherche d'une chaîne de caractères (string) vide. Ça tombe bien car c'est exactement ce que nous voulons ! En effet, l'URL resolver de Django ne considère pas '<http://127.0.0.1:8000/>' comme faisant partie de l'URL. Ce pattern va donc indiquer à Django d'afficher la vue `views.post_list` à un utilisateur de votre site web qui se rendrait à l'adresse "<http://127.0.0.1:8000/>".

La dernière partie, `name='post_list'`, est le nom de l'URL qui sera utilisée afin d'identifier la vue. Ce nom peut être le même que celui de la vue ou quelque chose de complètement différent. Plus tard dans ce tutoriel, nous allons utiliser les noms que nous avons donné à nos URLs. Il est donc important de donner un nom unique à chaque URL que nous créons. Pour vous faciliter la tâche, essayez de trouver des noms d'URLs simple à retenir.

Est-ce que tout fonctionne toujours ? Ouvrez votre navigateur à l'adresse <http://127.0.0.1:8000/> pour vérifier.



"It works" a disparu ! Ne vous en faites pas : ce que vous voyez est juste une page d'erreur. N'ayez pas peur des pages d'erreur, elles sont en fait très utiles :

Sur cette page, vous pouvez lire le message **no attribute 'post_list'** (il manque un attribut "post_list"). Est-ce que *post_list* vous rappelle quelque chose ? Yep, c'est le nom que nous avons donné à notre vue ! Cela signifie que nous avons posé les fondations mais, que nous n'avons pas encore créé notre *vue*. Pas de problème, on y vient :).

Si vous voulez en savoir plus au sujet de la configuration des URLs dans Django, vous pouvez aller consulter la documentation officielle du framework : <https://docs.djangoproject.com/fr/1.11/topics/http/urls/>

Créons nos vues Django!

Il est enfin temps de se débarrasser du bug que nous avons créé dans le chapitre précédent :)

C'est dans la `vue` que nous allons ranger toute la partie "logique" de notre application. C'est elle qui va se charger d'aller chercher les informations liées à notre `modèle` que nous venons de créer et de les passer à un `template`. Nous allons créer ce template dans le chapitre suivant. Concrètement, les vues ne sont que des méthodes Python un peu plus élaborées que celles que nous avons manipulées dans la partie **Introduction à Python**.

Les vues sont placées dans le fichier `views.py`. Nous allons créer nos `vues` dans le fichier `blog/views.py`.

blog/views.py

Ok, allons-y ! Ouvrons ce fichier pour voir ce qu'il contient :

```
from django.shortcuts import render

# Create your views here.
```

Il n'y pas encore grand chose dans ce fichier.

Les lignes qui commencent par `#` permettent de commenter notre code : ces lignes ne seront donc pas exécutées par Python. Pratique, non ?

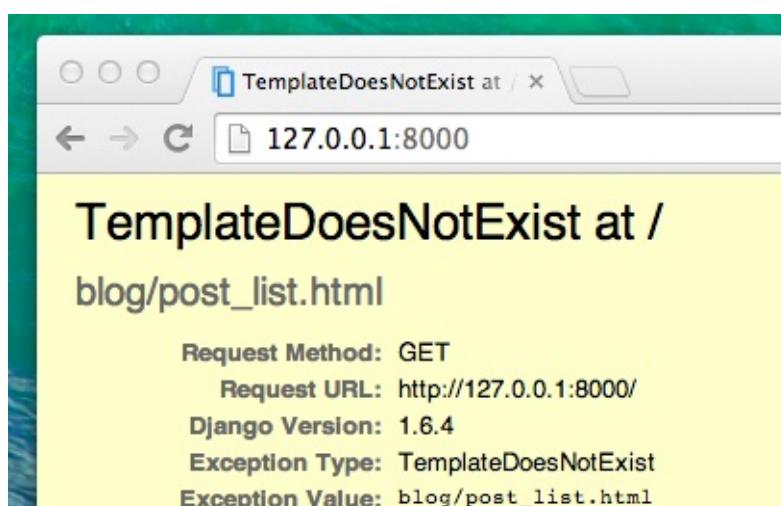
La vue la plus simple que l'on peut créer ressemble à ceci :

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

Comme vous pouvez le voir, nous avons créé une méthode (`def`) appelée `post_list` qui prend une `request` (requête) et `return` (retourne) une méthode `render` qui va permettre d'assembler tout ça selon notre template `blog/post_list.html`.

Sauvegardez votre fichier et allez à l'adresse <http://127.0.0.1:8000/> pour voir ce qui s'affiche maintenant.

Une autre erreur ! Voyons ce qu'elle nous dit :



Celle-là est plutôt simple : `TemplateDoesNotExist`. Corrigeons ça en créant un template dans la section suivante !

Pour en apprendre un peu plus sur les vues dans Django, consultez la documentation officielle :
<https://docs.djangoproject.com/fr/1.11/topics/http/views/>

Introduction au HTML

Vous vous demandez sûrement ce qu'est un template.

Un template est un fichier que vous pouvez réutiliser afin de présenter des informations différentes sous un seul et même format. Par exemple, vous pourriez avoir envie d'utiliser un template pour écrire une lettre : bien que son contenu varie ou qu'elle puisse être adressée à des personnes différentes, sa forme reste la même.

Le format d'un template Django est décrit grâce à un langage qui s'appelle HTML (c'est le même HTML que celui dont nous parlions dans le chapitre un, **Comment fonctionne l'Internet**).

Qu'est-ce que le HTML ?

HTML est un code simple qui est interprété par votre navigateur (Chrome, Firefox ou Safari) et qui permet d'afficher une page web à l'utilisateur.

L'abréviation HTML signifie « HyperText Markup Language ». **HyperText** signifie que c'est un type de texte qui supporte les hyperliens entre les pages. **Markup** signifie que nous avons pris un document et que nous avons balisé le code pour signifier (ici, au navigateur) comment il faut interpréter la page. Le code HTML est construit à l'aide de **balises**, chacune commençant par `<` et finissant par `>`. Ces balises représentent des **éléments** markup.

Votre premier template !

Créer un template signifie créer un fichier template. Et oui, encore des fichiers ! Vous aviez déjà probablement remarqué que tout tourne autour des fichiers.

Les templates sont sauvegardés dans le dossier `blog/templates/blog`. Tout d'abord, créons un dossier appelé `templates` à l'intérieur du dossier de notre blog. Ensuite, créez un autre dossier appelé `blog` à l'intérieur de votre dossier `templates` :

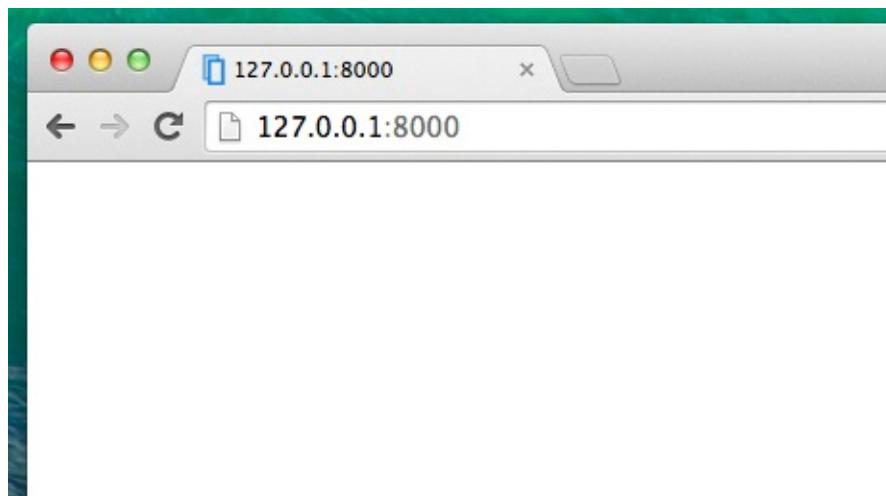
```
blog
└── templates
    └── blog
```

Vous pourriez vous demander pourquoi nous avons besoin de deux dossiers portant tous les deux le nom `blog`. Comme vous le découvrirez plus tard, c'est une simple convention de nommage qui va nous faciliter la vie quand les choses vont commencer à devenir compliquées.

Et maintenant, créez un fichier `post_list.html` (laisser le vide pour le moment) dans le dossier `blog/templates/blog`.

Allons regarder à quoi ressemble notre site maintenant : <http://127.0.0.1:8000/>

Si vous avez une erreur `TemplateDoesNotExist`, essayez de redémarrer votre serveur. Prenez votre ligne de commande et arrêtez votre serveur en appuyant simultanément sur `Ctrl+C` (les touches `Control` et `C` de votre clavier). Vous pouvez le relancer en tapant la commande `python manage.py runserver`.

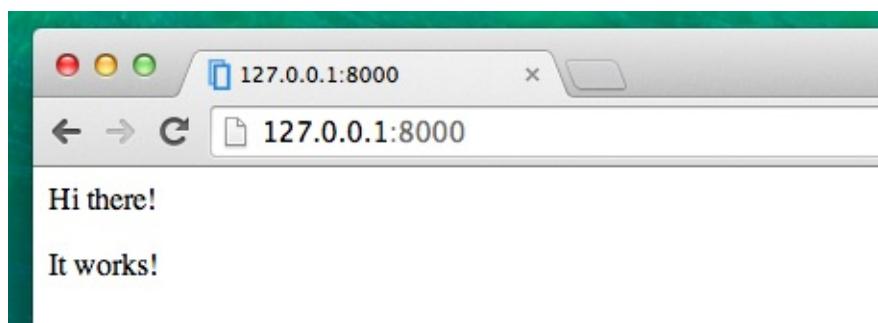


Et voilà, il n'y a plus d'erreurs ! Bravo :) Cependant, notre site ne peut rien faire d'autre pour le moment qu'afficher une page blanche. La faute à notre template que nous avons laissé vide. Allons corriger ça.

Ajoutez ce qui suit à votre fichier template :

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

Alors, à quoi ressemble notre site web maintenant ? Allons le découvrir : <http://127.0.0.1:8000/>



Ça marche ! Bon boulot :)

- La balise la plus élémentaire, `<html>`, figure toujours au début de n'importe quelle page web tandis que `</html>` est toujours située à la fin. Comme vous pouvez le constater, l'intégralité du contenu de notre page web est située entre la balise de départ, `<html>`, et la balise fermante, `</html>`.
- `<p>` est la balise pour les éléments de type paragraphe. `</p>` permet de fermer chaque paragraphe.

Head & body

Chaque page HTML est divisée en deux éléments : **head** (entête) et **body** (corps).

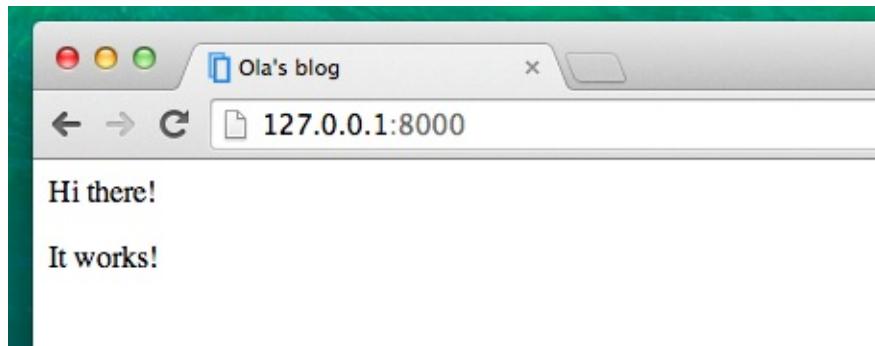
- **head** est un élément qui contient des informations sur le document : son contenu ne s'affichera pas à l'écran.
- **body** est un élément qui contient tout le reste. Son contenu s'affichera à l'écran et constituera notre page web.

Nous utilisons `<head>` pour transmettre la configuration de la page au navigateur tandis que `<body>` l'informe sur le contenu de la page.

Par exemple, vous pouvez donner un titre à votre site en utilisant l'élément **titre** dans le `<head>` :

```
<html>
  <head>
    <title>Le Blog d'Ola</title>
  </head>
  <body>
    <p>Hi there!</p>
    <p>It works!</p>
  </body>
</html>
```

Sauvegardez votre fichier et actualisez la page.



Vous avez vu comment le navigateur a compris que « Le Blog d'Ola » est le titre de votre page ? Il a interprété `<title>Le blog d'ola</title>` et a placé ce texte dans la barre de titre de votre navigateur (c'est ce titre qui va être aussi utilisé lorsque vous créez un marque-page, etc.).

Vous avez aussi probablement remarqué que chaque balise ouvrante possède sa *balise fermante*, composée d'un `/`, est qu'elles *encadrent* les différents éléments. Cela signifie que vous ne pouvez pas fermer une balise si celles imbriquées à l'intérieur de celle-ci n'ont pas été fermées.

Pensez à lorsque vous mettez des choses à l'intérieur de boîtes. Vous avez une grosse boîte, `<html></html>` ; à l'intérieur de celle-ci, on trouve une plus petite boîte, `<body></body>`, qui contient elle-même d'autres petites boîtes, `<p></p>`.

Essayez de vous rappeler cet exemple lorsque vous utilisez les balises *fermant*es et que vous avez des éléments *imbriqués*. Si vous ne suivez pas ces règles, votre navigateur risque de ne pas être capable d'interpréter votre code correctement et risque de mal afficher votre page web.

Personnaliser votre template

Et si nous en profitons pour nous amuser un peu ? Essayons de personnaliser notre template ! Voici quelques balises que vous pouvez utiliser :

- `<h1>Titre 1</h1>` - pour vos titres les plus importants
- `<h2>Titre 2</h2>` - pour les sous-titres
- `<h3>Titre 3</h3>` ... et ainsi de suite jusqu'à `<h6>`
- `texte` permet de mettre l'accent sur une partie du texte
- `texte` permet de mettre encore plus l'accent sur une partie de texte
- `
` permet d'insérer un saut de ligne (vous ne pouvez rien mettre à l'intérieur d'un élément br)
- `link` permet de créer un lien
- `premier itemsecond item` permet de créer des listes, comme celle que nous sommes en train de faire !
- `<div></div>` permet de créer une section au sein de la page

Voici un exemple de template utilisant plusieurs balises :

```

<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="">Mon 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>

```

Nous avons créé trois sections à l'aide de `div`.

- Le premier `div` contient le titre de notre blog - c'est à la fois un titre et un lien
- Les deux autres `div` contiennent nos posts avec leur date de publication, un titre de post `h2` qui est cliquable ainsi que deux `p` (paragraphe) de texte : un pour la date et l'autre pour notre post.

Ce qui nous donne :



Yaaay ! Pour l'instant, notre template nous permet seulement d'afficher les **mêmes informations** alors que nous disions précédemment qu'il doit nous permettre d'afficher des informations **differentes** utilisant le **même format**.

Ce qu'on aimerait pouvoir maintenant, c'est afficher les posts que nous avons créés précédemment dans l'interface d'administration de Django. Penchons-nous là dessus.

Une dernière chose : déployer !

Ne serait-il pas génial de pouvoir voir tout ces changements en ligne ? Hop, déployons à nouveau !

Committer et pusher votre code sur GitHub

Tout d'abord, allons voir quels sont les fichiers qui ont changé depuis notre dernier déploiement (lancez ces commandes dans votre console locale et non celle de PythonAnywhere) :

```
$ git status
```

Assurez-vous que vous êtes bien dans le dossier `djangogirls`. Voici la commande qui permet de dire à `git` d'inclure tout les changements qui ont eu lieu dans ce dossier :

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

`--all` (traduction de "tout") signifie que `git` va aussi analyser si vous avez supprimé des fichiers (par défaut, il ne s'intéresse qu'aux nouveaux fichiers ou à ceux modifiés). Essayez de vous rappeler du chapitre 3 : `.` permet de désigner le dossier courant.

Avant que nous puissions uploader nos fichiers, regardons ce que `git` à l'intention de faire (tous les fichiers que `git` va uploader vont apparaître en vert) :

```
$ git status
```

On y est presque : nous devons maintenant lui dire de sauvegarder ces changements dans son historique. Nous allons y ajouter un "message de commit" qui nous permettra de décrire ce qui a été changé. Vous pouvez mettre ce que vous voulez dans un message de commit. Généralement, il est préférable de mettre quelque chose d'utile qui vous permettra de vous souvenir plus tard de ce que vous avez fait.

```
$ git commit -m "Modification du HTML du site"
```

N'oubliez pas d'utiliser de doubles guillemets autour de votre message de commit.

Une fois que nous avons fait cela, nous pouvons mettre en ligne (pusher) nos modifications sur GitHub :

```
git push
```

Puller les modifications sur PythonAnywhere et recharger son appli web

- Allez sur la page des [consoles de PythonAnywhere](#). Retournez dans votre **console Bash** ou ouvrez-en une nouvelle puis tapez la commande suivante :

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

Voilà ! Votre code modifié est téléchargé. Si vous voulez vérifier ce que vous venez de récupérer, vous pouvez aller jeter un coup d'œil dans l'onglet **Files** de PythonAnywhere.

- Pour finir, n'oubliez pas de recharger votre application web : onglet [web](#) puis cliquez sur le bouton **Reload**.

Retournez sur votre site en cliquant sur l'adresse en haut de la page : normalement, vous devriez voir la dernière version. Si ce n'est pas le cas, ce n'est pas grave : n'hésitez pas à demander de l'aide à votre coach :)

L'ORM Django et les QuerySets

Dans ce chapitre, nous allons apprendre comment Django se connecte à la base de données et comment il y enregistre des choses. On respire un grand coup et on y va !

Qu'est-ce qu'un QuerySet ?

Un QuerySet est, par essence, une liste d'objets d'un modèle donné. C'est ce qui vous permet de lire, trier et organiser, des données présentes dans une base de données.

Il est plus simple d'apprendre avec un exemple. Et si nous nous intéressions à celui-ci ?

Le shell Django

Ouvrez la console de votre ordinateur (et non celle de PythonAnywhere) et tapez la commande suivante :

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

Ceci devrait maintenant s'afficher dans votre console :

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

Vous êtes maintenant dans la console interactive de Django. C'est comme celle de Python, mais avec toute la magie qu'apporte Django :). Du coup, les commandes Python sont aussi utilisables dans cette console.

Lister tous les objets

Essayons tout d'abord d'afficher tous nos posts. Vous pouvez le faire à l'aide de cette commande :

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

Ooops ! Voilà que ça nous renvoie une erreur qui nous dit qu'il n'existe pas de Post. En effet, nous avons oublié de commencer par un "import" !

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

Rien de compliqué : nous importons le modèle `Post` depuis notre `blog.models`. Essayons à nouveau la commande précédente :

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

Cela nous permet d'obtenir une liste des posts que nous avons créé tout à l'heure ! Rappelez-vous : nous avions créé ces posts à l'aide de l'interface d'administration de Django. Cependant, nous aimerais maintenant créer de nouveaux posts à l'aide de Python : comment allons-nous nous y prendre ?

Créer des objets

Voici comment créer un nouveau objet Post dans la base de données :

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

Cependant, il nous manque un petit quelque chose : `me`. Nous avons besoin de lui passer une instance du modèle `User` en guise d'auteur (author). Comment faire ?

Tout d'abord, il nous faut importer le modèle User :

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

Avons-nous des utilisateurs dans notre base de données ? Voyons voir :

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

C'est le superutilisateur que nous avions créé tout à l'heure ! Essayons maintenant d'obtenir une instance de l'utilisateur :

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

Comme vous pouvez le voir, nous obtenons (`get`) un utilisateur (`User`) avec comme nom d'utilisateur (`username`) 'ola'. Cool ! Bien sûr, vous pouvez utiliser votre nom si vous le souhaitez.

Nous allons enfin pouvoir créer notre post :

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

Youpi ! Et si on vérifiait quand même si ça a marché ?

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

Et voilà : un post de plus dans la liste !

Ajouter plus de posts

Amusez-vous à ajouter d'autres posts pour vous entraîner un peu. Essayez d'ajouter 2-3 posts en plus puis passez à la partie suivante.

Filtrer les objets

L'intérêt des QuerySets, c'est que l'on peut les filtrer. Disons que nous aimerais retrouver tous les posts écrits par l'utilisateur Ola. Pour cela, nous allons utiliser `filter` à la place de `all` dans `Post.objects.all()`. Les parenthèses vont nous servir à préciser quelles sont les conditions auxquelles un post de blog doit se conformer pour être retenu par notre QuerySet. Dans notre exemple, `author` est égal à `me`. La manière de le dire en Django c'est : `author=me`. Maintenant, votre bout de code doit ressembler à ceci :

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

Et si nous voulions chercher les posts qui contiennent uniquement le mot "titre" dans le champs `title` ?

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

Il y a deux tirets bas (`_`) entre `title` et `contains`. L'ORM de Django utilise cette syntaxe afin de séparer les noms de champ ("title") et les opérations ou les filtres ("contains"). Si vous n'utilisez qu'un seul tiret bas, vous allez obtenir une erreur du type : "FieldError: Cannot resolve keyword title_contains".

Comment obtenir une liste de tous les posts publiés ? Cela se fait facilement en filtrant tous les posts qui ont une date de publication, `published_date`, dans le passé :

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

Malheureusement, le post que nous avons créé dans la console Python n'est pas encore publié. Allons corriger ce problème ! Dans un premier temps, nous aimerais obtenir une instance du post que nous voulons publier :

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

Ensuite, publions-le grâce à notre méthode `publish` !

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

Maintenant, essayez d'obtenir à nouveau la liste des posts publiés. Pour cela, appuyez trois fois sur la flèche du haut et appuyez sur `entrée` :

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

Classer les objets

Les QuerySets permettent aussi de trier la liste des objets. Essayons de les trier par le champ `created_date` :

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

On peut aussi inverser l'ordre de tri en ajouter `-` au début :

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

Chainer les QuerySets

Vous pouvez aussi combiner les QuerySets and les **chainant** les unes aux autres :

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

C'est un outil très puissant qui va vous permettre d'écrire des requêtes complexes.

Génial ! Vous êtes maintenant prête à passer à l'étape suivante ! Pour fermer le shell, tapez ceci:

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

Données dynamiques dans les templates

Nous avons différents morceaux en place : le modèle `Post` qui est défini dans le fichier `models.py`, la vue `post_list` dans `views.py` et nous venons de créer notre template. Mais comment allons-nous faire pour faire apparaître nos posts dans notre template HTML ? Car au final, n'est-ce pas le but que nous souhaiterions atteindre ? Nous aimerais prendre du contenu, en l'occurrence notre modèle sauvegardé dans notre base de données, et réussir à joliment l'afficher dans notre template.

C'est à ça que servent les *vues* : connecter les modèles et les templates. Dans notre vue `post_list`, nous allons avoir besoin de prendre les modèles dont nous avons besoin et de les passer au template. Concrètement, c'est dans la *vue* que nous allons décider ce qui va s'afficher (modèle) dans un template.

Ok, et sinon, on fait comment ?

Nous allons avoir besoin d'ouvrir le fichier `blog/views.py`. Pour l'instant, la vue `post_list` ressemble à ceci :

```
from django.shortcuts import render

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

Est-ce que vous vous souvenez de comment rajouter des morceaux de code écrits dans d'autres fichiers ? Nous en avons parlé dans un chapitre précédent. Nous allons devoir importer notre modèle qui est défini dans le fichier `models.py`. Pour cela, nous allons ajouter la ligne `from .models import Post` de la façon suivante :

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

Le point après `from` signifie le *dossier courant* ou *l'application courante*. Comme `views.py` et `models.py` sont dans le même dossier, nous pouvons tout simplement utiliser `.` et le nom du fichier, sans le `.py`. Ensuite, nous importons le modèle (`Post`).

Ok, et après ? Afin de pouvoir aller chercher les véritables posts de blog de notre modèle `Post`, nous avons besoin de quelque chose qui s'appelle un `QuerySet`.

QuerySet

Normalement, ce mot doit vous évoquer quelque chose. Nous en avons un peu parlé dans la section [Django ORM \(QuerySets\)](#).

Maintenant, nous allons nous intéresser à une liste de blog posts qui sont publiés et classés par date de publication (`published_date`). Ça tombe bien, on a déjà fait ça dans la section sur les QuerySets !

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

Il ne nous reste plus qu'à mettre cette ligne de code à l'intérieur de notre fichier `blog/views.py`, dans la fonction `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', {})
```

Veuillez noter que nous créons une *variable* pour notre QuerySet : `posts`. Considérez que c'est le nom de notre QuerySet. À partir de maintenant, nous allons pouvoir faire référence à notre QuerySet en utilisant ce nom.

Notez aussi que notre code utilise la fonction `timezone.now()` que nous allons aussi devoir importer de `timezone`.

Il nous manque encore un petit quelque chose : passer notre QuerySet `posts` à notre template. Nous nous intéresserons plus particulièrement à celui-ci dans la section suivante.

Dans la fonction `render`, nous avons déjà un paramètre `request`, qui désigne tout ce que nous recevons d'un utilisateur par l'intermédiaire d'Internet, et un fichier template appelé `'blog/post_list.html'`. Le dernier paramètre, qui ressemble à `{}`, va nous permettre de glisser des instructions que notre template va suivre. Nous avons besoin par exemple de lui donner des noms : nous allons rester sur `'posts'` pour le moment :). Ça va ressembler à ça : `{'posts': posts}`. La partie située avant `:` est une chaîne de caractères : vous devez donc l'entourer de guillemets `''`.

Au final, notre fichier `blog/views.py` doit ressembler à ceci maintenant :

```
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})
```

Et voilà, c'est bon ! Nous allons retourner du côté de notre template pour que notre QuerySet puisse s'afficher correctement !

Si vous voulez en savoir plus sur les QuerySets, n'hésitez pas à consulter la documentation officielle du framework : <https://docs.djangoproject.com/fr/1.11/ref/models/querysets/>

Templates Django

Il est temps d'afficher des données ! Pour nous aider, Django fournit des balises de gabarit (**template tags**) qui sont intégrées au framework. Pour le reste du tutoriel, nous utiliserons plutôt le mot template, bien plus répandu que sa traduction "gabarit".

Qu'est-ce que c'est que des balises de template ?

En HTML, vous ne pouvez pas mettre directement du code Python car les navigateurs seraient incapables de le comprendre. Les navigateurs ne connaissent que le HTML. Nous vous avons signalé précédemment que HTML est du genre statique, alors que Python est bien plus dynamique.

Les **Balises de template Django** nous permettent de transférer des choses ressemblant à du Python dans du HTML afin de nous permettre de construire des sites web plus rapidement et facilement. Cool, non ?

Template d'affichage de la liste des posts

Dans le chapitre précédent, nous avons donné à notre template une liste de posts à l'aide de la variable `posts`. Nous allons maintenant les afficher en HTML.

Afin d'afficher une variable dans un template Django, nous utiliserons des doubles accolades avec le nom de la variable à l'intérieur. Ça ressemble à ceci :

```
{{ posts }}
```

Essayez de faire la même chose avec votre template `blog/templates/blog/post_list.html`. Remplacez tout ce qui se trouve entre la seconde balise `<div>` jusqu'au troisième `</div>` avec la ligne `{{ posts }}` . Sauvegardez votre fichier et rafraîchissez votre page pour voir le résultat :



Comme vous pouvez le voir, tout ce que nous avons, c'est ceci :

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

Cela signifie que Django l'interprète comme une liste d'objets. Essayez de vous rappeler comment afficher des listes en Python. Si vous avez un trou de mémoire, allez voir dans le chapitre **Introduction à Python**. Vous avez trouvé ? Avec des boucles ! Dans un template Django, vous pouvez les écrire de la façon suivante :

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

Essayez ceci dans votre template.

The screenshot shows a web browser window titled "Django Girls blog" with the URL "127.0.0.1:8000". The page displays the title "Django Girls Blog" and two posts: "My second post" and "My first post".

Ça marche ! Cependant, nous aimerais plutôt les afficher à la manière des posts statiques, comme lorsque nous les avions créés dans le chapitre **Introduction au HTML**. Vous pouvez mixer HTML et balises de template. Notre `body` ressemble maintenant à ceci :

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

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

Tout ce qui se situe entre `{% for %}` et `{% endfor %}` va être répété pour chaque objet présent dans la liste. Rafraîchissez votre page :

The screenshot shows a web browser window titled "Django Girls blog" with the URL "127.0.0.1:8000". The page displays the title "Django Girls Blog" and two posts: "published: June 30, 2014, 10:58 p.m." and "My second post". Below "My second post" is a large block of placeholder text: "Vestibulum id ligula porta felis euismod semper. Cum sociis natoque penatibus et m: justo odio, dapibus ac facilisis in, egestas eget quam. Lorem ipsum dolor sit amet, co consectetur ac, vestibulum at eros. Sed posuere consectetur est at lobortis. Cum socii mattis consectetur purus sit amet fermentum." Further down, another post is partially visible: "published: June 29, 2014, noon" and "My first post".

Avez-vous remarqué que nous utilisons une notation légèrement différente cette fois ({{ post.title }} or {{ post.text }}) ? Nous accédons aux données associées à chaque champ défini dans notre modèle `Post` . De même, les barres verticales | nous permettent de rediriger le texte des posts à travers un filtre qui convertit automatiquement les fins de lignes en paragraphes.

Encore une chose !

Maintenant, ça serait bien de voir si votre site Web fonctionne toujours sur Internet. Nous allons essayer de le re-déployer sur PythonAnywhere. Voici un récapitulatif des étapes...

- En premier lieu, envoyez votre code sur GitHub (push)

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

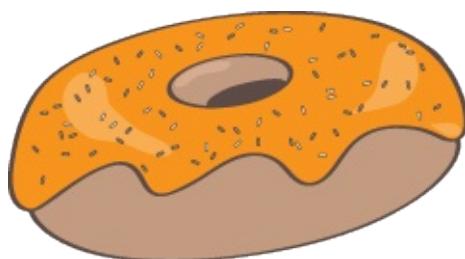
- Ensuite, reconnectez-vous à [PythonAnywhere](#) et allez sur la "**Bash console**" (ou démarrez-en une nouvelle), et lancez les commandes suivantes :

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

- Finalement, allez sur l'onglet **Web** et cliquez sur **Reload** sur votre application web. Votre site mis-à-jour devrait être en ligne !

Félicitations ! Maintenant, pourquoi ne pas essayer d'ajouter un nouveau post à l'aide de l'interface d'administration ? N'oubliez pas d'ajouter une date de publication ! Ensuite, rafraîchissez votre page et regardez si votre post apparaît.

Ça a marché ? Nous sommes super fières de vous ! Éloignez vous un peu de votre clavier maintenant : vous avez mérité de faire une pause. :)



CSS - Rendez votre site joli !

Soyons honnêtes : notre blog est plutôt moche, non ? Un peu de CSS devrait nous permettre d'arranger ça !

Qu'est-ce que le CSS ?

Les feuilles de style, ou Cascading Style Sheets (CSS), sont un langage informatique utilisé pour décrire l'apparence et le formatage d'un document écrit en langage markup (ex : HTML). Pour faire simple : des produits cosmétiques pour pages web ;)

Je suppose que vous n'avez pas particulièrement envie de partir de rien et de devoir tout construire vous-même. Pour éviter cela, nous allons une nouvelle fois utiliser différentes ressources créées et mises à disposition gratuitement sur Internet par d'autres développeurs·ses. Réinventer à chaque fois la roue n'est pas vraiment fun, en plus, c'est absolument inutile.

Utilisons Bootstrap !

Bootstrap est l'un des frameworks HTML et CSS les plus populaires. Il est utilisé pour créer de très beaux sites web : <https://getbootstrap.com/>

Il a été créé par d'anciens programmeurs·ses de chez Twitter et est maintenant développé par des bénévoles aux quatre coins du monde.

Installer Bootstrap

Pour installer Bootstrap, vous avez besoin d'ajouter ceci dans le `<head>` de votre fichier `.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">
```

En faisant ceci, vous n'ajoutez aucun nouveau fichier à votre projet : vous reliez simplement des fichiers hébergés sur Internet à votre projet. Essayez maintenant de rafraîchir votre page. Et voilà !



C'est déjà un peu mieux !

Les fichiers statiques dans Django

Enfin, allons jeter un coup d'œil à ces **fichiers statiques** dont nous n'arrêtons pas de vous parler. Votre CSS et vos images sont des fichiers statiques et non dynamiques. Cela signifie que leur contenu ne dépend pas du contexte de la requête et qu'il sera donc identique pour chaque utilisateur.

Où ranger les fichiers statiques dans Django ?

Comme vous l'avez probablement remarqué lorsque nous avons exécuté la commande `collectstatic` sur le serveur, Django sait déjà où trouver les fichiers statiques pour la partie "admin". Maintenant, il ne nous reste plus qu'à ajouter les fichiers statiques liés à notre app `blog`.

Pour cela, nous allons créer un dossier appelé `static` à l'intérieur de notre blog app :

```
djangogirls
└── blog
    ├── migrations
    ├── static
    └── templates
    └── mysite
```

Django est capable de détecter automatiquement tout les dossiers appelés "static" dans l'arborescence de votre app. Il sera ainsi capable d'utiliser les fichiers présents à l'intérieur de ces dossiers comme des fichiers statiques.

Votre première CSS !

Nous allons créer un fichier CSS afin de personnaliser notre page web. Créez un nouveau dossier appelé `css` à l'intérieur de votre dossier `static`. Ensuite, créez un nouveau fichier appelé `blog.css` à l'intérieur du dossier `css`. Vous êtes prête ?

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

Et c'est parti pour un peu de CSS ! Ouvrez le fichier `static/css/blog.css` dans votre éditeur de texte.

Nous n'allons pas trop nous attarder sur les CSS aujourd'hui. Nous vous invitons, une fois rentrée chez vous, à vous plonger dans d'autres tutoriels de CSS. Vous verrez, c'est assez simple à comprendre ! Vous pouvez par exemple consulter le cours [Codecademy HTML & CSS course](#) qui est une excellente ressource et qui vous permettra d'en apprendre plus sur la personnalisation de site web à l'aide de CSS.

Que pourrions-nous faire rapidement ? Pourquoi ne pas changer la couleur de notre entête ? Pour indiquer la couleur que nous souhaitons utiliser, nous devons utiliser un code particulier. Ces codes commencent par `#` et sont suivis de 6 lettres (A-F) et chiffres (0-9). Afin de trouver le code associé à la couleur de votre choix, vous pouvez consulter le site <http://www.colorpicker.com/>. Vous pouvez aussi utiliser des [couleurs prédefinies](#), comme `red` ou `green`.

Ajoutez le code suivant dans votre fichier `blog/static/css/blog.css` :

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

`h1 a` est un sélecteur CSS. Cela signifie que nous appliquons ce style pour chaque élément `a` présent à l'intérieur d'un élément `h1` (ce qui est le cas de `<h1>link</h1>`). Dans notre exemple précédent, nous indiquons notre souhait de changer la couleur du texte en `#FCA205`, c'est à dire en orange. Bien évidemment, vous êtes libre de choisir

n'importe quelle couleur !

Un fichier CSS permet de déterminer le style des éléments présents dans un fichier HTML. Les différents éléments sont identifiés par leur nom (`a`, `h1`, `body`), l'attribut `class` ou l'attribut `id`. `Class` et `id` sont des noms que vous choisissez vous-même. Les classes définissent des groupes d'éléments tandis que les ids désignent des éléments précis. Par exemple, l'élément suivant peut être identifié par CSS à la fois par son nom `a`, sa classe `external_link`, ou son identifiant `link_to_wiki_page` :

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

Nous vous conseillons d'en apprendre un peu plus sur les sélecteurs CSS sur [w3schools](#).

Afin que nos modifications fonctionnent, nous devons aussi signaler à notre template HTML que nous utilisons des CSS. Ouvrez le fichier `blog/templates/blog/post_list.html` et ajoutez cette ligne au tout début de celui-ci :

```
{% load static %}
```

Hop, nous chargeons les fichiers statiques :). Pour l'ajout de code suivant, gardez en tête que le navigateur lit vos fichiers dans l'ordre dans lequel ils lui sont donnés : en le plaçant à l'endroit que nous vous indiquons, vous allez pouvoir remplacer du code provenant des fichiers Bootstrap par le vôtre. Donc, entre le `<head>` et le `</head>` et après les liens vers les fichiers CSS de Bootstrap, ajoutez ceci :

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

Nous venons simplement de dire à notre template où nous avons rangé notre fichier CSS.

Maintenant, votre fichier doit ressembler à ceci :

```
{% load static %}
<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>
        <h2><a href="{{ post.title }}>{{ post.title }}</a></h2>
        <p>{{ post.text|linebreaksbr }}</p>
      </div>
    {% endfor %}
  </body>
</html>
```

Ok, on sauvegarde et on rafraîchit la page !

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

Bravo ! Peut-être que nous pourrions un peu aérer notre page web en augmentant la marge du côté gauche ? Essayons pour voir !

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

Ajoutez ceci à votre fichier CSS, sauvegardez-le et voyons le résultat !

The screenshot shows the same browser window after applying the CSS rule. The left margin of the main content area has been increased, creating a visual gap between the left edge of the page and the start of the blog posts.

Et si nous changions aussi la police de caractères de notre entête ? Collez ceci dans la partie `<head>` de votre fichier `blog/templates/blog/post_list.html` :

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

Cette ligne nous permet d'importer la police de caractères *Lobster* depuis Google Fonts (<https://www.google.com/fonts>).

Maintenant, ajoutons `font-family: 'Lobster';` à l'intérieur du bloc déclaratif `h1 a` dans le fichier CSS `blog/static/css/blog.css`. Le bloc déclaratif est le code situé à l'intérieur des accolades `{` et `}`. N'oubliez pas ensuite de rafraîchir la page.

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



Super !

Comme nous l'avions mentionné précédemment, il existe une notion de classe dans CSS. En gros, cela vous permet de donner un nom à un morceau de code HTML auquel vous souhaitez appliquer un style particulier sans que cela ne concerne le reste du code. C'est particulièrement pratique lorsque vous avez deux divs qui font quelque chose de différent (par exemple, votre entête et votre post) et que vous ne voulez pas qu'ils soient identiques.

Allons donner des noms à certaines parties de notre code html. Ajouter la classe `page-header` à votre `div` qui contient votre entête. Votre fichier doit ressembler maintenant à ceci :

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

Maintenant, ajoutez la classe `post` à votre `div` contenant votre blog post.

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

Nous allons maintenant ajouter des blocs déclaratifs à différents sélecteurs. Les sélecteurs qui commencent par `.` sont reliés aux classes. Le net regorge de bons tutoriels sur CSS qui vous permettront de comprendre le code que nous allons rajouter à notre fichier. Pour l'instant, copier-coller le code qui suit dans votre fichier `blog/static/css/blog.css` :

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

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

.content {
    margin-left: 40px;
}

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

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

.save {
    float: right;
}

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

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

.post {
    margin-bottom: 70px;
}

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

Nous allons maintenant nous intéresser au code concernant les posts. Il va falloir remplacer le code suivant :

```
{% 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 %}
```

Ce code se trouve dans le fichier `blog/templates/blog/post_list.html`. Il doit être remplacé par le code suivant :

```
<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>
```

Sauvegardez les fichiers modifiés et rafraîchissez votre site web.



Woohoo ! C'est pas mal, non ? Le code que nous avons collé n'est pas bien compliqué à comprendre et vous devriez pouvoir en saisir l'essentiel rien qu'en le lisant (ce n'est pas grave si ce n'est pas le cas !).

N'ayez pas peur et jouez un peu avec la CSS : essayez par exemple d'en changer des morceaux. Vous avez cassé quelque chose ? Pas de problème : vous pouvez toujours annuler vos modifications !

Voilà pour la partie CSS. Nous vous encourageons vivement à suivre le tutoriel gratuit de [Code Academy](#) : considérez ce tutoriel comme un petit travail à faire une fois rentrée chez vous. Vous connaîtrez ainsi tout ce qu'il y a à savoir pour rendre son site bien plus joli !

Prête pour le chapitre suivant ? :)

Héritage de template

Django vous réserve encore bien des surprises : une assez géniale est l'**héritage de template**. Qu'est ce que ça signifie ? C'est une fonctionnalité qui vous permet de réutiliser certains morceaux de HTML dans différentes pages de votre site web.

Concrètement, cela permet d'éviter de vous répéter dans chaque fichier lorsque vous voulez utiliser la même information ou mise en page. Ainsi, lorsque vous voudrez changer quelque chose, vous n'aurez à le faire qu'une seule fois !

Créer un template de base

Un template de base est le template le plus simple que vous pouvez faire hériter à chaque page de votre site web.

Créons le fichier `base.html` dans le dossier `blog/templates/blog/` :

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

Ensuite, ouvrez ce fichier `base.html` et collez-y tout ce qui se trouve dans le fichier `post_list.html`. Ça devrait ressembler à ça :

```
{% load static %}
<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.get_absolute_url }}>{{ post.title }}</a></h1>
              <p>{{ post.text|linebreaksbr }}</p>
            </div>
          {% endfor %}
        </div>
      </div>
    </div>
  </body>
</html>
```

Puis, dans le fichier `base.html`, remplacez tout ce qui se trouve dans `<body>` (de `<body>` à `</body>`) par ceci :

```
<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>
```

Nous venons concrètement de remplacer tout ce qui se trouve entre `{% for post in posts %}{% endfor %}` par :

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

Qu'est-ce que cela signifie ? Vous venez simplement de créer un `block` : c'est une balise de template qui vous permet d'insérer le HTML de ce block dans d'autres templates qui héritent de `base.html`. Nous vous expliquerons comment faire dans un instant.

Maintenant, sauvegardez votre fichier puis ouvrez à nouveau `blog/templates/blog/post_list.html`. Supprimez tout ce qui n'est pas dans body. Supprimez aussi `<div class="page-header"></div>`. Votre fichier doit maintenant ressembler à ça :

```
{% 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 %}
```

Maintenant, ajoutez cette ligne au début du fichier :

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

Cela signifie que nous sommes en train d'étendre le modèle du template `base.html` dans `post_list.html`. Une dernière chose à faire : déplacez tout le contenu du fichier dans la partie située entre `{% block content %}` et `{% endblock %}`. Attention à ne pas déplacer la ligne que nous venons juste d'insérer. Comme ceci :

```
{% 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 %}
```

Et voilà ! Vérifiez que votre site fonctionne toujours correctement :)

Si jamais vous rencontrez une erreur de type `TemplateDoesNotExist` qui signale que le fichier `blog/base.html` n'existe pas et que `runserver` tourne dans votre console, tuez-le (en appuyant sur les touches Ctrl+C en même temps) et relancez votre server à l'aide de la commande `python manage.py runserver`.

Finaliser votre application

Nous avons déjà franchi toutes les étapes nécessaires à la création de notre site web : nous savons maintenant comment écrire un modèle, une URL, une vue et un template. Nous avons même réussi à rendre notre site web plus joli !

C'est le moment de pratiquer tout ce que vous avez appris aujourd'hui !

Tout d'abord, il faudrait que notre blog possède une page qui permet d'afficher un post, n'est-ce pas ?

Nous avons déjà un modèle `Post`, nous n'avons donc pas besoin de retourner éditer `models.py`.

Créer un lien dans un template

Nous allons tout d'abord ajouter un lien à l'intérieur du fichier `blog/templates/blog/post_list.html`. Pour le moment, ce fichier doit ressembler à ceci :

```
{% 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 %}
```

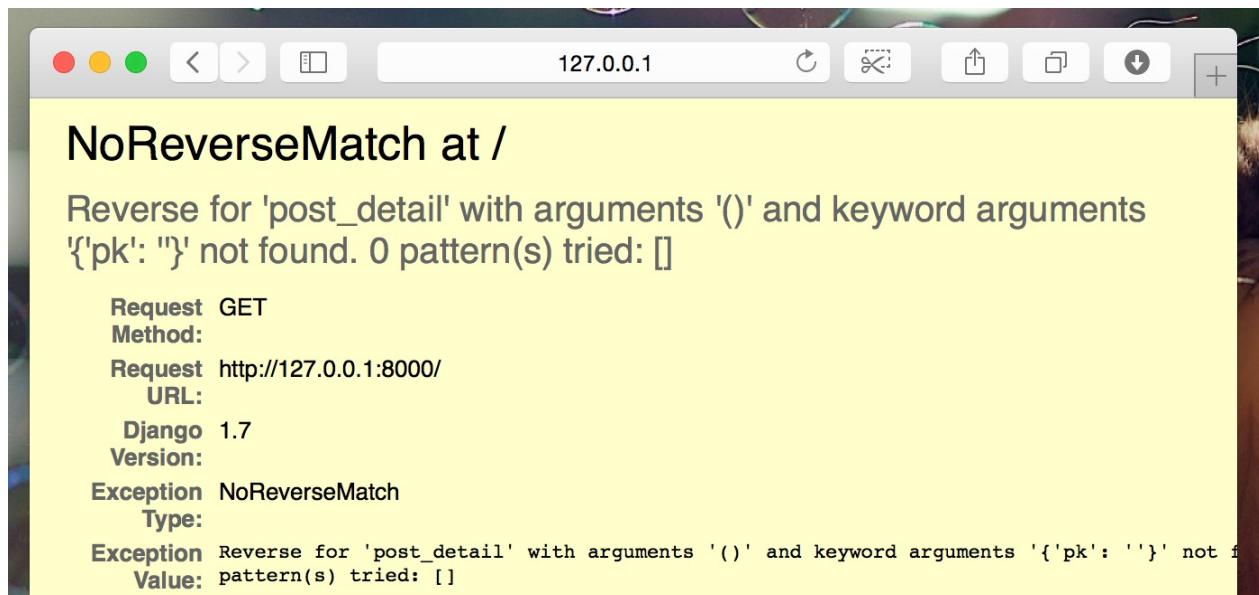
Nous aimerais pouvoir cliquer sur le titre du post et arriver sur une page avec le contenu de celui-ci. Pour cela, changeons `<h1>{{ post.title }}</h1>` pour qu'il pointe vers la page de contenu du post :

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

C'est le moment parfait pour expliquer ce mystérieux `{% url 'post_detail' pk=post.pk %}`. Vous vous souvenez peut-être que la notation `{% %}` nous permet d'utiliser les balises de template Django. Cette fois, nous allons utiliser des balises qui vont s'occuper de créer des URLs à notre place !

`blog.views.post_detail` est le chemin d'accès vers la vue `post_detail` que nous aimerais créer. Attention : `blog` désigne notre application (le dossier `blog`) et `views` le fichier `views.py`. Enfin, `post_detail` est le nom de notre vue.

Si nous essayons d'aller à <http://127.0.0.1:8000/>, nous allons rencontrer une erreur : nous n'avons pas d'URL ou de vue pour `post_detail`. L'erreur ressemble à ceci :



Créer une URL vers le contenu d'un post

Allons créer notre URL dans le fichier `urls.py` pour notre vue `post_detail` !

Nous aimions que le contenu de notre premier post s'affiche à cette URL : <http://127.0.0.1:8000/post/1/>

Allons créer une URL dans le fichier `blog/urls.py` qui pointera Django vers une vue appelée `post_detail`. Cela nous permettra d'afficher l'intégralité d'un post de blog. Ajoutez la ligne `url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail, name='post_detail'),` dans le fichier `blog/urls.py`. Votre fichier devrait maintenant ressembler à ceci :

```
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'),
]
```

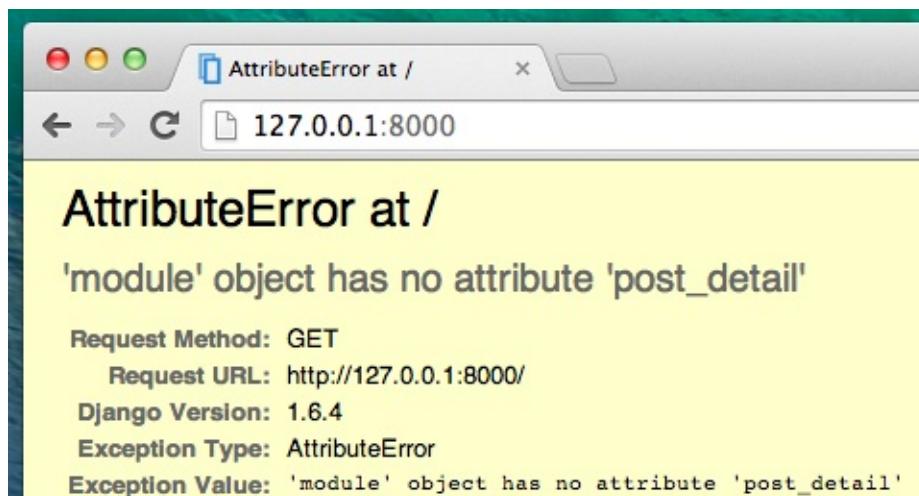
`^post/(?P<pk>[0-9]+)/$` a l'air plutôt effrayant mais, ne vous inquiétez pas, décortiquons-le ensemble :

- Il commence par `^`, qui désigne le "début"
- `post/` signifie seulement qu'après le début, l'URL doit contenir le mot `post` et `/`. Jusque-là, tout va bien.
- `(?P<pk>[0-9]+)` : ok, là, on s'accroche :). Cela signifie que Django va prendre tout ce que vous placez là et le transférer à une vue sous la forme d'une variable appelée `pk`. `[0-9]` nous dit aussi que nous ne voulons que des nombres (tout ce qui est entre 0 et 9 inclus) et non des lettres. `+` signifie qu'il faut, au minimum, un chiffre à cet endroit. Du coup, quelque chose comme `http://127.0.0.1:8000/post//` n'est pas valide tandis que `http://127.0.0.1:8000/post/1234567890/` l'est complètement!
- `/` - nous avons encore besoin d'un `/`
- `$` - "la fin!"

Concrètement, cela signifie que si vous entrez `http://127.0.0.1:8000/post/5/` dans votre barre d'adresse, Django va comprendre que vous cherchez à atteindre une vue appelée `post_detail` et qu'il doit communiquer l'information que `pk` est égal `5` dans cette vue.

`pk` est un raccourci pour `primary key`. Ce nom est très souvent utilisé dans les projets Django. Cependant, vous pouvez appeler cette variable comme bon vous semble, toujours dans la limite des règles suivantes : pas d'accents, pas de caractères spéciaux, des minuscules et des `_` à la place des espaces. Par exemple, à la place de `(?P<pk>[0-9]+)`, nous pourrions utiliser la variable `post_id`, ce qui donnerait : `(?P<post_id>[0-9]+)`.

Comme nous venons d'ajouter un nouveau pattern d'URL au fichier `blog/urls.py`, rafraîchissons la page :
<http://127.0.0.1:8000/> Boom ! Encore une erreur ! Mais on s'y attendait ;)



Est-ce que vous vous souvenez de ce que nous devons faire ensuite ? Il faudra ajouter une vue !

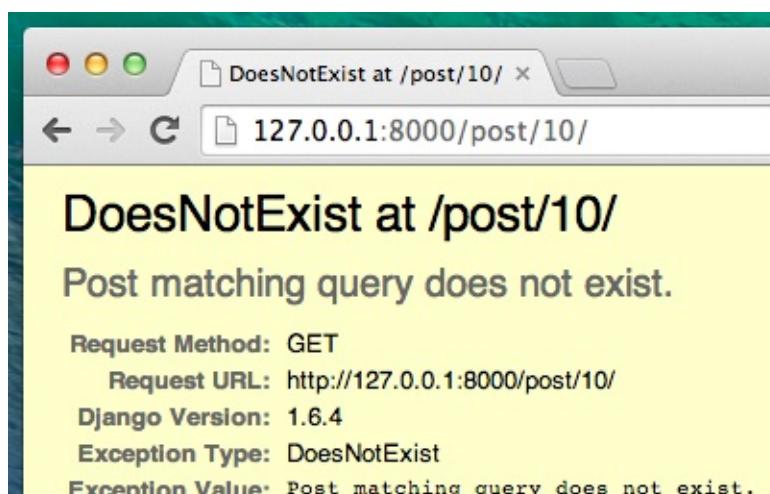
Ajouter une vue pour le contenu du post

Cette fois, nous allons donner un paramètre supplémentaire à notre vue : `pk`. Notre vue va avoir besoin de le récupérer. Pour cela, nous allons définir une fonction : `def post_detail(request, pk):`. Attention : notez bien que nous utilisons le même nom que celui que nous avons spécifié dans le fichier url (`pk`). Oublier cette variable est incorrecte et va générer une erreur !

Maintenant, nous n'aimerions obtenir qu'un seul blog post. Pour cela, nous allons utiliser des QuerySets qui ressemblent à ceux-ci :

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

Cependant, il y a un petit problème dans cette ligne de code. Si aucun de nos `Posts` ne possède cette `primary key` (clé primaire) (`pk`), nous allons nous retrouver avec une super erreur bien cracra !



Dans l'idéal, nous aimerais pouvoir éviter ça ! Comme d'habitude, Django nous offre l'outil parfait pour ça : `get_object_or_404`. Dans le cas où il n'existerait pas de `Post` avec le `pk` indiqué, une page d'erreur beaucoup plus sympathique s'affichera : c'est ce qu'on appelle une `erreur 404 : page non trouvée`.



La bonne nouvelle, c'est que vous pouvez créer vous-mêmes votre page `Page non trouvée` et en faire ce que vous voulez ! Reconnaissez que ce n'est pas le plus important pour le moment donc nous allons zapper cette partie ;).

Ok, ajoutons notre `vue` à notre fichier `views.py` !

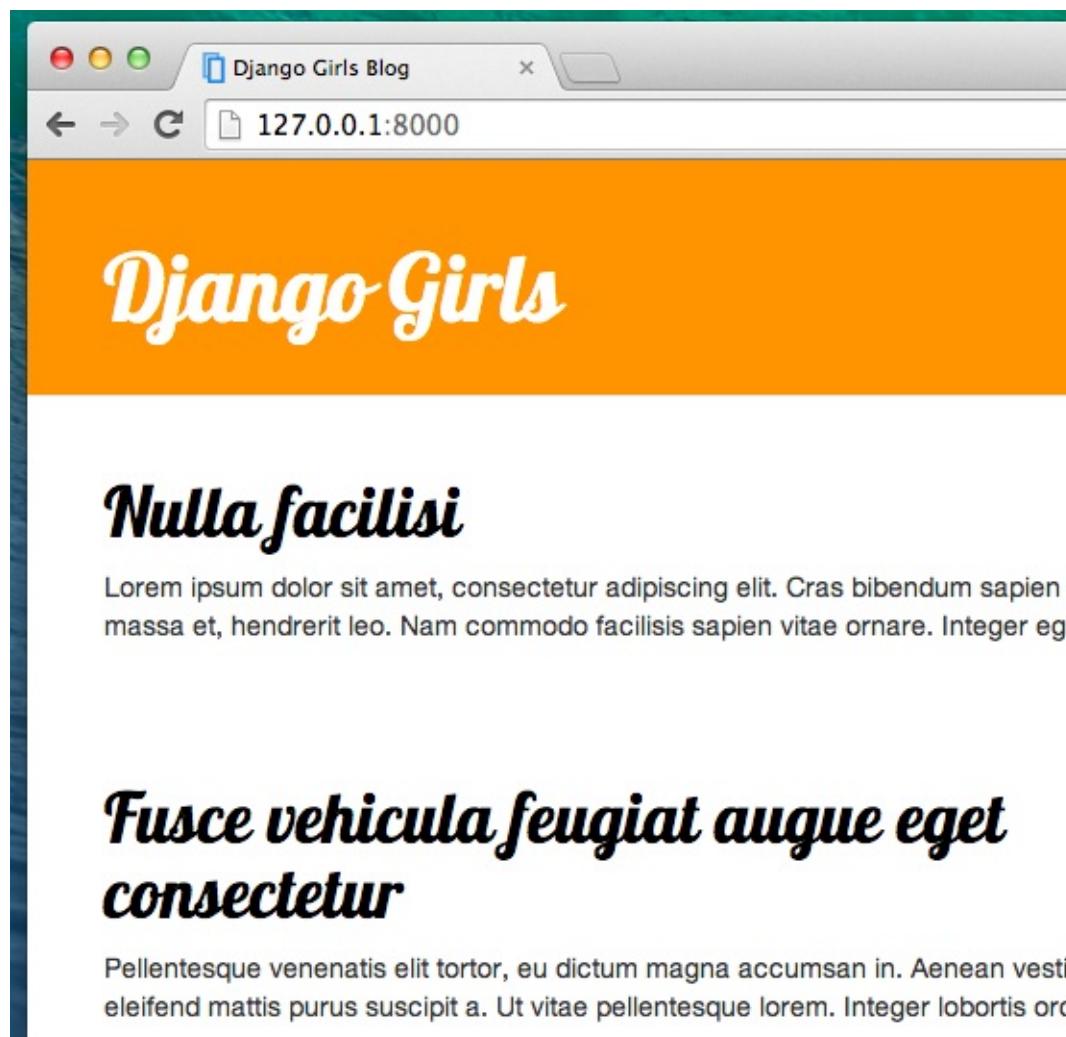
Ouvrons le fichier `blog/views.py` et ajoutons le code suivant:

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

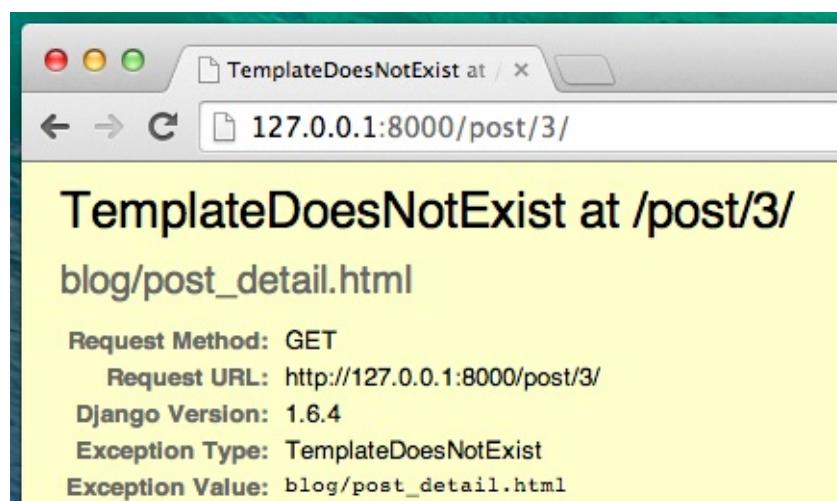
Cette ligne est à ajouter en dessous des lignes `from` situées en début de fichier. Ensuite, à la fin de notre fichier, nous allons ajouter notre `vue` proprement dite:

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

Hop, réactualisons la page <http://127.0.0.1:8000/>



C'est bon, ça a marché ! Mais que se passe-t-il lorsque nous cliquons sur un lien dans un titre de blog post ?



Oh non ! Encore une erreur ! Mais cette fois, vous savez quoi faire : nous avons besoin d'un template !

Créer un template pour le contenu du post

Nous allons créer un fichier `post_detail.html` dans le dossier `blog/templates/blog`.

Ça ressemblera à ça :

```
{% 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 %}
```

Une nouvelle fois, nous faisons hériter de `base.html`. Dans le `content` block, nous voulons que s'affiche la date de publication d'un post (si elle existe), son titre et son texte. Mais vous souhaitez peut-être quelques éclaircissements avant, non?

`{% if ... %} ... {% endif %}` est une balise de template que nous pouvons utiliser si nous voulons vérifier quelque chose : souvenez-vous de `if ... else ..` de la section **Introduction à Python**. Dans ce scénario, nous aimerions vérifier si la date de publication d'un post (`published_date`) n'est pas vide.

Ok, vous pouvez maintenant rafraîchir votre page et voir si la page `Page not found` a enfin disparu.



Yay ! Ça marche!

Encore un petit effort : déployons !

Nous ferions bien de mettre à jour la version de notre site présente sur PythonAnywhere. On s'accroche et on déploie encore une fois :)

```
$ 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
```

- Puis, dans la console bash de [PythonAnywhere](#):

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

- Enfin, cliquez sur l'onglet [Web](#) et cliquez sur **Reload**.

Normalement, ça devrait suffire ! Encore bravo :)

Formulaires Django

La dernière chose que nous voulons faire sur notre site web, c'est créer une manière sympathique d'ajouter ou d'éditer des blog posts. L'interface d'administration de Django est cool, mais elle est assez complexe à personnaliser et à rendre jolie. Les formulaires vont nous donner un pouvoir absolu sur notre interface : nous allons être capables de faire à peu près tout ce que nous pouvons imaginer !

Ce qui est pratique avec les formulaires Django, c'est que nous pouvons aussi bien en définir un à partir de rien ou créer un `ModelForm` qui va enregistrer le résultat du formulaire dans un modèle.

C'est la seconde solution que nous allons utiliser pour créer un formulaire pour notre modèle `Post`.

Comme toutes les choses importantes dans Django, les formulaires ont leur propre fichier : `forms.py`.

Nous allons devoir créer un fichier avec ce nom dans notre dossier `blog`.

```
blog
└── forms.py
```

Ouvrez maintenant ce fichier et tapez le code suivant :

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

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

Nous avons besoin tout d'abord d'importer les formulaires Django (`from django import forms`), puis, évidemment, notre modèle `Post` (`from .models import Post`).

Comme vous l'avez probablement deviné, `PostForm` est le nom de notre formulaire. Nous avons besoin de préciser à Django que ce formulaire est un `ModelForm`. Pour cela, nous utilisons `forms.ModelForm`.

Ensuite, nous avons une `Meta classe` qui nous permet de dire à Django quel modèle il doit utiliser pour créer ce formulaire (`model = Post`).

Enfin, nous précisons quels sont les champs qui doivent figurer dans notre formulaire. Dans notre cas, nous souhaitons que seuls `title` et `text` apparaissent dans notre formulaire. Nous obtiendrons les autres données différemment : par exemple, on s'attend à ce que l'auteur (`author`) soit la personne actuellement enregistrée (c'est à dire vous !) et que la date de création `created_date` soit générée automatiquement lors de la création du post (cf code que nous avons écrit).

Et voilà, c'est tout ! Tout ce qu'il nous reste à faire, c'est d'utiliser ce formulaire dans une vue et de l'afficher dans un template.

Nous allons donc une nouvelle fois suivre le processus suivant et créer : un lien vers la page, une URL, une vue et un template.

Lien vers une page contenant le formulaire

C'est le moment d'ouvrir le fichier `blog/templates/blog/base.html`. Nous allons ajouter un lien dans un `div` appelé `page-header` :

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

Remarquez que notre nouvelle vue s'appelle `post_new`.

Après avoir ajouté cette ligne, votre fichier html devrait maintenant ressembler à ceci :

```
{% load static %}
<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>
```

Sauvegardez votre fichier et rafraîchissez la page <http://127.0.0.1:8000> : vous devez normalement tomber encore une fois sur l'erreur `NoReverseMatch` !

URL

Ouvrez le fichier `blog/urls.py` et ajoutez cette ligne :

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

Votre fichier doit maintenant ressembler à ceci :

```
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'),
]
```

Une fois la page rechargée, vous allez voir une `AttributeError`, ce qui est normal. Nous n'avons pas encore implémentée la vue `post_new`. Allons nous occuper de ça maintenant.

La vue post_new

Ouvrez maintenant le fichier `blog/views.py` et ajoutez les lignes suivantes avec celles du `from` qui existent déjà :

```
from .forms import PostForm
```

Puis ajoutez la vue :

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

Afin de pouvoir créer un nouveau formulaire `Post`, nous avons besoin d'appeler la fonction `PostForm()` et de la passer au template. Nous reviendrons modifier cette `vue` plus tard, mais pour l'instant, créons rapidement un template pour ce formulaire.

Template

Pour cela, nous avons besoin de créer un fichier `post_edit.html` dans le dossier `blog/templates/blog`. Afin que notre formulaire fonctionne, nous avons besoin de plusieurs choses :

- Nous voulons afficher le formulaire. Nous pouvons le faire à l'aide d'un simple `{{ form.as_p }}` par exemple.
- La ligne précédente va avoir besoin d'être enveloppée dans des balises HTML : `<form method="POST">...</form>`
- Nous avons besoin d'un bouton `save` (`sauvegarder`). Nous allons le créer à l'aide d'un bouton HTML : `<button type="submit">Save</button>`
- Enfin, nous devons ajouter `{% csrf_token %}` juste après `<form ...>`. C'est très important car c'est ce qui va permettre de sécuriser votre formulaire ! De toute manière, si vous avez oublié ce petit morceau, Django vous le fera remarquer lorsque vous sauvegarderez le formulaire :

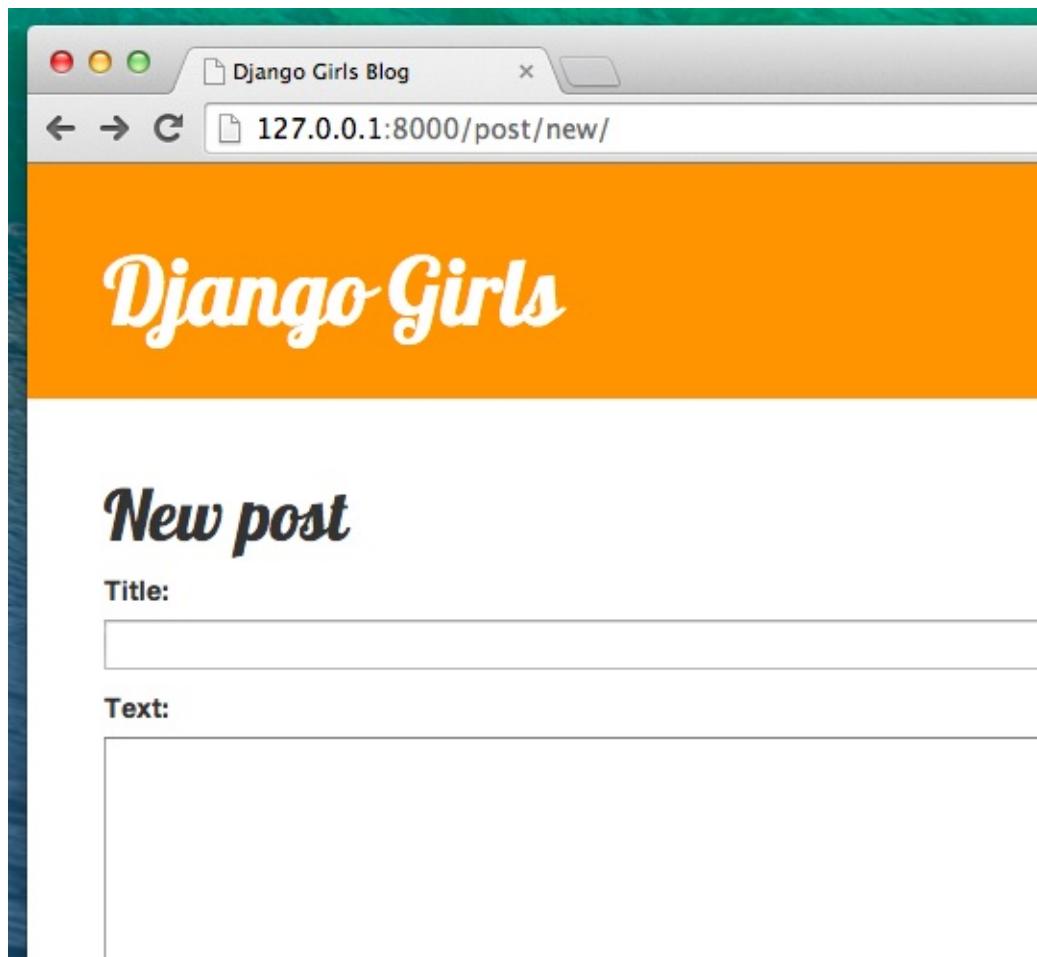


Ok, voyons maintenant à quoi devrait ressembler le fichier `post_edit.html` :

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

{% block content %}
    <h1>New post</h1>
    <form method="POST" class="post-form">{% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
    </form>
{% endblock %}
```

Rafraîchissons la page ! Et voilà : le formulaire s'affiche !



Heu, attendez une minute... Quand vous tapez quelque chose dans `title` et `text` et que vous essayez de le sauvegarder, que se passe-t-il ?

Absolument rien ! Nous retombons sur la même page sauf qu'en plus, notre texte a disparu et aucun post ne semble avoir été créé. Que s'est-il passé ?

La réponse est : rien ! Nous avons juste encore un peu de travail à accomplir. Retournons à notre vue.

Sauvegarder le contenu du formulaire

Ouvrez à nouveau `blog/views.py`. Actuellement, `post_new` n'est composé que des lignes de code suivantes :

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

Lorsque nous envoyons notre formulaire, nous revenons à la même vue. Cependant, nous avons des données dans `request`, et plus particulièrement dans `request.POST`. Prenez garde que "POST" ici n'a aucun lien avec "blog post" : le nom est lié au fait que nous envoyons des données. Rappelez-vous : nous avions défini la variable `method="POST"` dans le fichier HTML qui contient notre `<formulaire>` ? Tous les champs du formulaire se trouvent maintenant dans `request.POST`. Veillez à ne pas renommer `POST` en quoi que ce soit d'autre : la seule autre valeur autorisée pour `method` est `GET`. Malheureusement, nous n'avons pas le temps de rentrer dans les détails aujourd'hui.

Dans notre vue, nous avons donc deux situations différentes à gérer. Tout d'abord, nous avons la situation où nous accédons à la page pour la première fois et que nous voulons un formulaire vide. Ensuite, nous avons la seconde situation où nous retournons sur la vue et nous voulons que les champs du formulaire contiennent les informations que nous avions tapées. Pour gérer ces deux cas, nous allons utiliser une condition `if` (si).

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

Attaquons-nous à remplir ces [...] . Si la `method` est `POST`, c'est que nous voulons construire notre `PostForm` avec les données de notre formulaire. Pour cela, nous devons ajouter :

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

Facile ! La prochaine étape est de vérifier si le contenu de notre formulaire est correct. Nous aimerais vérifier, par exemple, que les champs obligatoires soient bien remplis et qu'aucune donnée incorrecte ne soit sauvegardée. Pour cela, nous allons utiliser `form.is_valid()` .

Testons donc si notre formulaire est valide et, si c'est le cas, sauvegardons-le !

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

En gros, nous effectuons deux choses ici : nous sauvegardons le formulaire grâce à `form.save` et nous ajoutons un auteur. Rappelez-vous, il n'y avait pas de champ `author` dans le `PostForm` mais nous avons obligatoirement besoin d'un auteur pour que le formulaire que nous avons créé soit valide. `commit=False` signifie que nous ne voulons pas encore enregistrer notre modèle `Post` . Nous voulons tout d'abord ajouter un auteur. La plupart du temps, vous utiliserez `form.save()` sans `commit=False` . Cependant, nous en avons besoin ici. `post.save()` va nous permettre de sauvegarder les changements, c'est-à-dire l'ajout d'un auteur. Et voilà, maintenant c'est sûr, un nouveau blog post sera créé !

Enfin, ce serait génial si nous pouvions tout de suite aller à la page `post_detail` du post de blog que nous venons de créer. Pour cela, nous avons besoin d'importer une dernière chose :

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

Ajoutez-le au tout début de votre page. Maintenant, nous allons ajouter la ligne qui signifie : "aller à la page `post_detail` pour le nouveau post qui vient d'être créé".

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

`post_detail` est le nom de la vue où nous voulons aller. Rappelez-vous : une vue a besoin d'une variable `pk` . Afin de le passer à la vue, nous utilisons `pk=post.pk` , où `post` désigne le blog post nouvellement créé !

Et si au lieu de parler, nous vous montrions à quoi ressemble maintenant notre `vue` ?

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

Voyons si ça marche. Allez à l'adresse <http://127.0.0.1:8000/post/new/>, ajoutez un titre et du texte, puis sauvegardez... Et voilà ! Le nouveau post est bien créé et vous êtes redirigé vers la page post_detail !

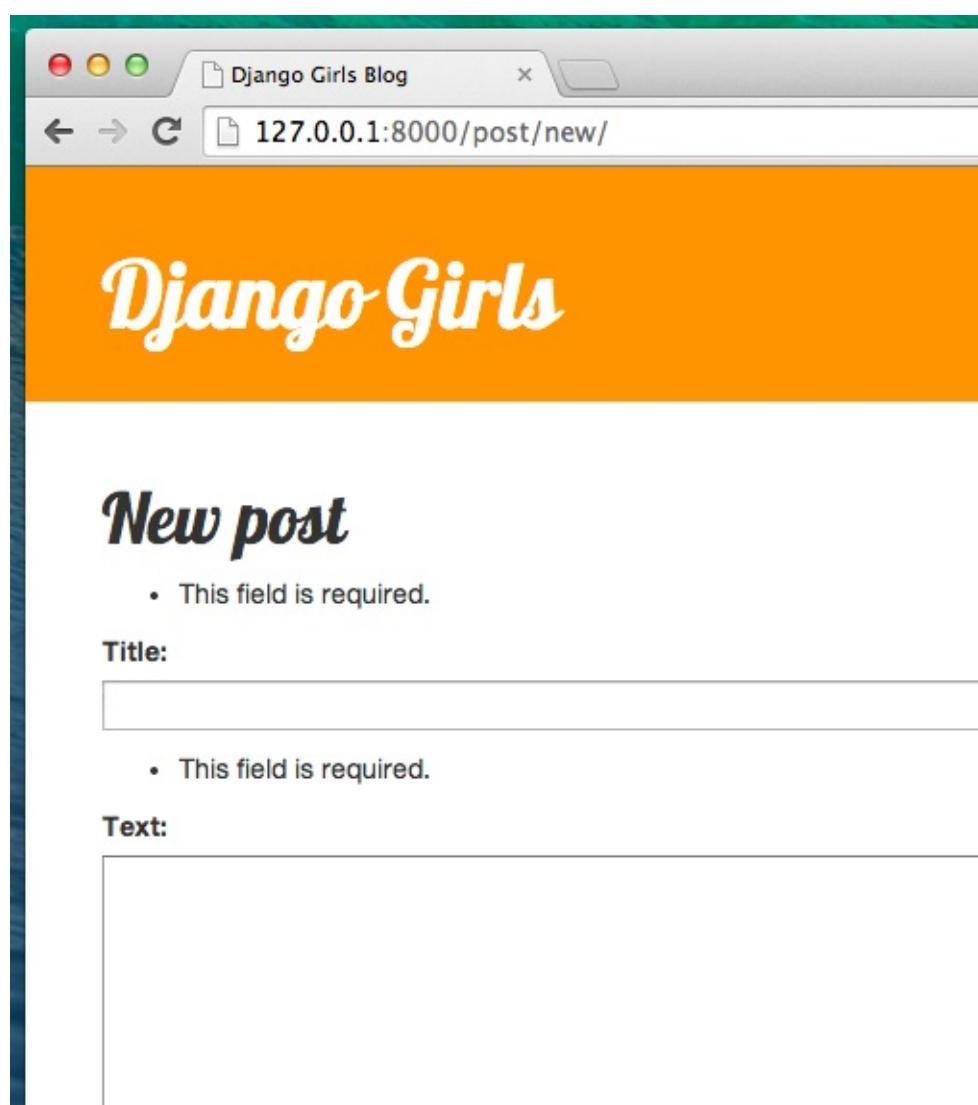
Vous avez peut-être remarqué que nous avons choisi une date de publication avant de sauvegarder le post. Nous en aurons besoin lorsque nous créerons le *publish button* (bouton publier) dans l'un des extensions du tutoriel **Django Girls** (en anglais).

Encore bravo !

Validation de formulaire

Maintenant, nous allons vous montrer à quel point les formulaires Django sont cool ! Un post de blog a besoin de champs title (titre) et text (texte). Dans notre modèle Post, nous n'avons pas signalé que ces champs n'étaient pas obligatoire (à l'inverse de published_date). Django s'attend donc à ce qu'ils soient remplis à chaque fois.

Essayez de sauvegarder un formulaire sans mettre de titre ou de texte. Devinez ce qui va se passer !



Django va s'occuper de la validation : il va regarder si tous les champs de notre formulaire sont en adéquation avec notre modèle. C'est cool, non ?

Comme nous avons récemment utilisé l'interface d'administration de Django, le système pense que nous sommes encore connectés. Cependant, il y a plusieurs cas qui peuvent amener un utilisateur à être déconnecté : fermer le navigateur, redémarrer la base de données, etc. Si jamais vous obtenez des erreurs lors de la création d'un post qui disent que vous n'êtes pas connecté, retournez sur la page d'administration présente à l'adresse <http://127.0.0.1:8000/admin> et connectez-vous à nouveau. Cependant, vous devinez bien que cette solution n'est pas suffisante à long terme. Afin de corriger ce problème, n'hésitez pas à faire la partie **Devoir : ajouter de la sécurité à son site internet !** qui est située juste après la partie principale du tutoriel.

```
ValueError at /post/new/
Cannot assign "<SimpleLazyObject: <django.contrib.auth.models.AnonymousUser object at 0x3264b50>>": "Post.author" must be a "User" instance.

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

Exception Location: /home/lemonde/Downloads/Django/django-example-project/localsite/localsite/blog/models.py in post, line 220
```

Éditer un formulaire

Maintenant, nous savons comme ajouter un nouveau formulaire. Comment faire si nous voulons éditer un formulaire déjà existant ? C'est assez proche de ce que nous venons de faire. Créons ce dont nous avons besoin rapidement. Si jamais des choses vous semblent obscures, n'hésitez pas à demander à votre coach ou à revoir les chapitres précédents. Tout ce que nous allons faire maintenant a déjà été expliqué plus en détail précédemment.

Ouvrez le fichier `blog/templates/blog/post_detail.html` et ajoutez la ligne suivante :

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

Votre template doit ressembler à ceci :

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

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

Maintenant, dans `blog/urls.py`, ajoutez cette ligne :

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

Nous allons réutiliser le template de `blog/templates/blog/post_edit.html`. Il ne va donc nous manquer qu'une vue.

Ouvrons `blog/views.py` et ajoutons à la toute fin du fichier :

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

Vous ne trouvez pas que ça ressemble presque à la vue de `post_new` ? Regardons un peu plus en détail. Tout d'abord, nous passons un paramètre `pk` supplémentaire. Ensuite, nous récupérons le modèle `Post` que nous souhaitons éditer à l'aide de `get_object_or_404(Post, pk=pk)`. Puis, lorsque nous créons un formulaire, nous faisons de ce post deux instances . Tout d'abord lorsque nous sauvageons le formulaire :

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

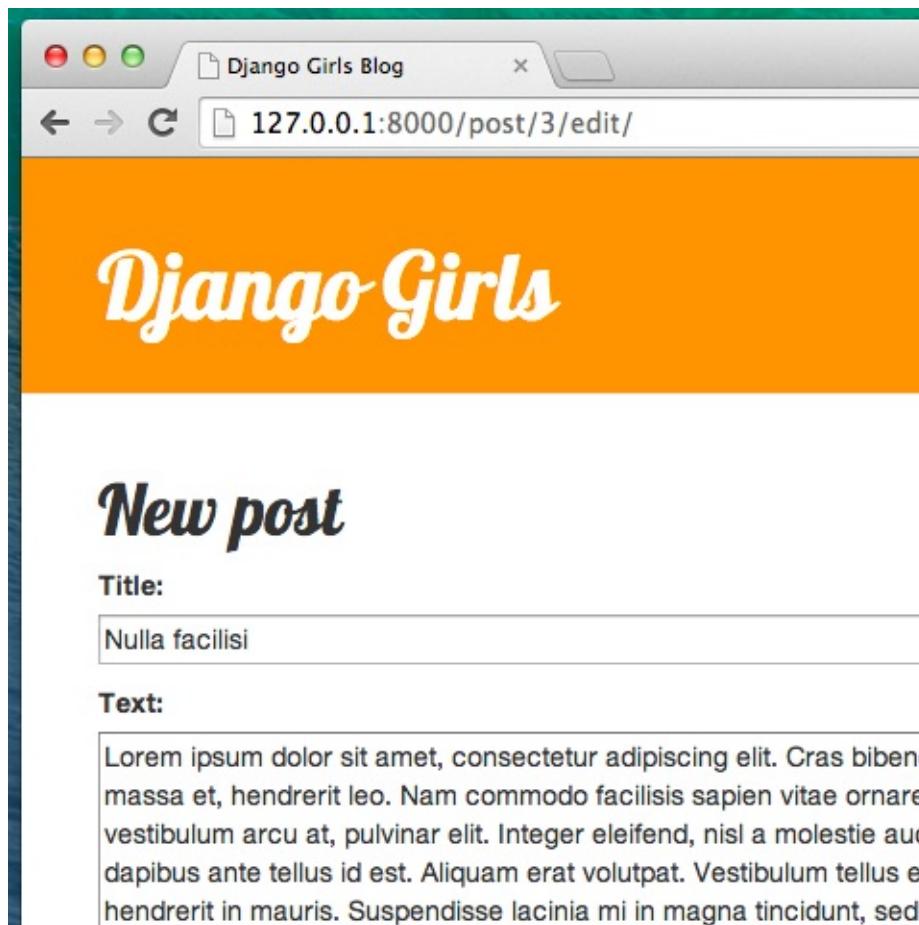
Puis ensuite lorsque nous ouvrons le formulaire associé à ce post afin de l'éditer :

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

Alors, voyons si ça marche ! Allons à la page `post_detail` . Un bouton d'édition devrait apparaître dans le coin supérieur droit de la page :



Lorsque vous cliquez dessus, vous devez voir le formulaire du post de blog apparaître :



Essayez de manipuler un peu ce que vous venez de créer : ajoutez du texte, changez le titre puis sauvegardez ces changements !

Bravo ! Votre application se complexifie et contient de plus en plus de fonctionnalités !

Si jamais vous voulez en savoir plus sur les formulaires dans Django, n'hésitez pas à lire la documentation associée : <https://docs.djangoproject.com/fr/1.11/topics/forms/>

Sécurité

C'est génial de pouvoir créer de nouveaux posts juste en cliquant sur un lien ! Mais n'importe qui visitant votre site pourra mettre un nouveau post en ligne, ce qui n'est probablement pas ce que vous souhaitez. Faisons en sorte que le bouton n'apparaisse seulement qu'à vous.

Dans `blog/templates/blog/base.html`, trouvez notre `page-header` `div` et la balise ancre que vous y avez mis plus tôt. Ça doit ressembler à ça :

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

On va y ajouter une autre balise `{% if %}` qui ne fera apparaître le lien qu'aux utilisateurs·trices connecté·e·s dans l'administration : uniquement vous pour le moment ! Changez la balise `<a>` comme ceci :

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

Ce `{% if %}` fait en sorte de n'envoyer le lien au navigateur que si l'utilisateur·trice demandant la page est connecté·e. Ce n'est pas une protection complète, mais c'est un bon début. Nous reviendrons sur les questions de sécurité dans les extensions du tutoriel.

Comme vous êtes probablement connectée, vous ne verrez aucune différence si vous rafraîchissez la page. Mais chargez la page dans un autre navigateur ou dans une fenêtre incognito, et vous verrez que le lien n'apparaît pas !

Encore un petit effort : déployons !

Nos modifications fonctionnent-elles sur PythonAnywhere ? Pour le savoir, déployons à nouveau !

- Tout d'abord, commitez votre nouveau code et poussez-le à nouveau sur GitHub

```
$ git status  
$ git add --all .  
$ git status  
$ git commit -m "Ajout de vues qui permettent de créer et d'éditer un post de blog sur le site."  
$ git push
```

- Puis, dans la console bash de [PythonAnywhere](#):

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

- Enfin, cliquez sur l'onglet [Web](#) et cliquez sur **Reload**.

Normalement, ça devrait suffire ! Encore bravo :)

La suite?

Un immense bravo à vous ! **Vous êtes totalement géniale.** Nous sommes fière de vous ! < 3

Que faire maintenant ?

Faites une pause et détendez-vous. Vous venez d'accomplir quelque chose de vraiment énorme.

Après ça, vous pouvez :

- Suivez Django Girls sur [Facebook](#) ou [Twitter](#) pour être tenu au courant

À la recherche de ressources supplémentaires ?

Jetez un coup d'œil à notre autre livre, [Django Girls Tutorial: Extensions](#) (en anglais).

Ensuite, vous pouvez essayer les ressources listées ci-dessous. Elles sont toutes recommandables !

- [Django's official tutorial](#)
- [New Coder tutorials](#)
- [Code Academy Python course](#)
- [Code Academy HTML & CSS course](#)
- [Django Carrots tutorial](#)
- [Learn Python The Hard Way book](#)
- [Getting Started With Django video lessons](#)
- [Two Scoops of Django: Best Practices for Django 1.8 book](#)