



Diese Anleitung basiert auf dem Django Girls Tutorial: <https://tutorial.djangogirls.org/>

Dieses Werk ist unter der Creative Commons Attribution-ShareAlike 4.0 International License lizenziert. Eine Kopie dieser Lizenz finden Sie auf <http://creativecommons.org/licenses/by-sa/4.0/>

## Willkommen

Willkommen beim informatics4girls-Workshop "Blog-Software programmieren"! Wir freuen uns, dass du hier bist :) In diesem Tutorial schauen wir gemeinsam unter die Haube der Technologien im Internet, geben dir einen Einblick in die Bits und Bytes, die zusammen das Internet bilden, wie wir es heute kennen.

Wie alles Unbekannte wird das ein Abenteuer sein - aber keine Sorge: Da du bereits den Mut aufgebracht hast, hier zu sein, wirst du das schon meistern :)

## Einleitung

Willst du wissen, wie Programme und Web-Applikationen entstehen, und willst du selbst eine bauen?

Dann haben wir hier gute Neuigkeiten für dich! Programmieren ist nicht so schwer, wie du denkst und wir zeigen dir hier, wie viel Spaß es machen kann.

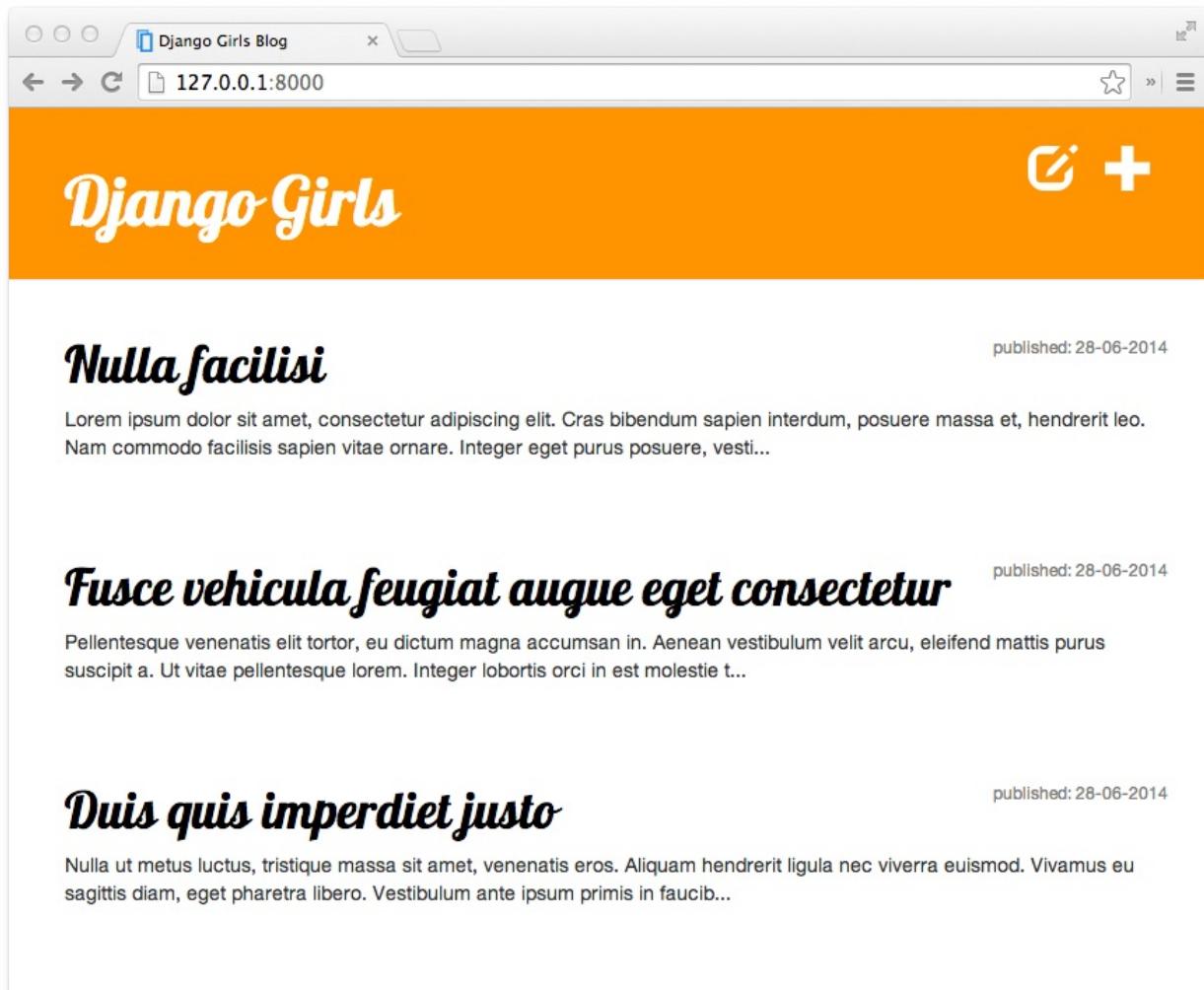
Dieses Tutorial wird dich nicht auf zauberhafte Weise in eine Programmiererin verwandeln. Wenn du gut darin sein willst, brauchst du Monate oder sogar Jahre des Lernens und Übens. Aber wir wollen dir zeigen, dass Programmieren oder Webseitenerstellen nicht so kompliziert ist, wie es scheint. Wir versuchen, dir auf einfache Art verschiedene, kleine Teile zu zeigen, so dass du davon nicht eingeschüchtert wirst.

Wir hoffen, dass du danach diese Technik und Technologien so sehr mögen wirst wie wir!

## Was lernst du in diesem Tutorial?

Wenn du mit dem Tutorial fertig bist, hast du eine einfache, aber funktionierende Webanwendung: deinen eigenen Blog. Wir zeigen dir, wie man ihn online stellt, andere können dein Werk also sehen!

Es wird (in etwa) so aussehen:



## ***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...

## **Das Tutorial daheim durcharbeiten**

Beim informatics4girls-Workshop arbeitet ihr auf Computern, auf denen die benötigte Software schon drauf ist. Daher wurden in dieser Druck-Ausgabe des Tutorials die entsprechenden Installations-Anleitungen weggelassen.

Falls du zuhause am Tutorial arbeiten willst, verwende daher die Online-Version auf <https://tutorial.djangogirls.org/>, die dich durch alle notwendigen Installationen führt.

# Wie das Internet funktioniert

Für die Leser zu Hause: Dieses Kapitel wird im Video [How the Internet Works](#) behandelt.

Dieses Kapitel wurde inspiriert durch den Vortrag "How the Internet works" von Jessica McKellar (<http://web.mit.edu/jessstess/www/>).

Wahrscheinlich nutzt du das Internet jeden Tag. Aber weißt du, was passiert, wenn du eine Adresse wie <https://djangogirls.org> im Browser eingibst und `enter` drückst?

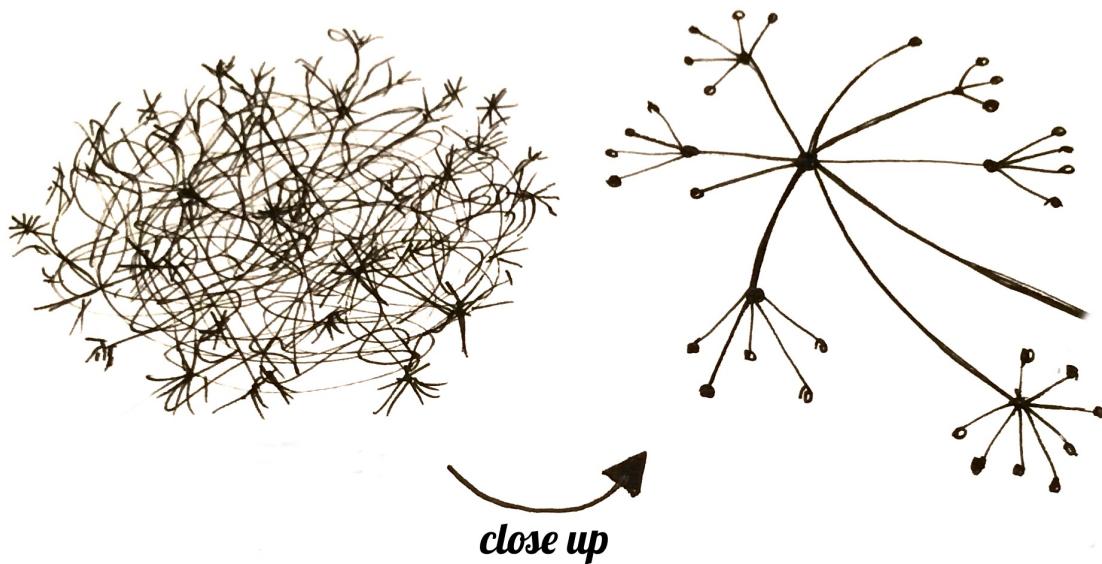
Als Erstes solltest du wissen, dass eine Webseite meist nur ein paar Dateien auf der Festplatte sind. Genau wie deine Filme, Musik oder Bilder. Das Besondere an Webseiten ist, dass sie aus speziellem Computer-Code bestehen, das sogenannte HTML.

Wenn du noch nie etwas mit Programmierung zu tun hastest, kann auch HTML zuerst abschreckend aussehen, aber dein Browser (Chrome, Safari, Firefox, etc.) liebt es. Browser sind so entworfen, dass sie diesen Code verstehen, seinen Anweisungen folgen können und diese Dateien, aus denen deine Website besteht, genau so darstellen, wie du es möchtest.

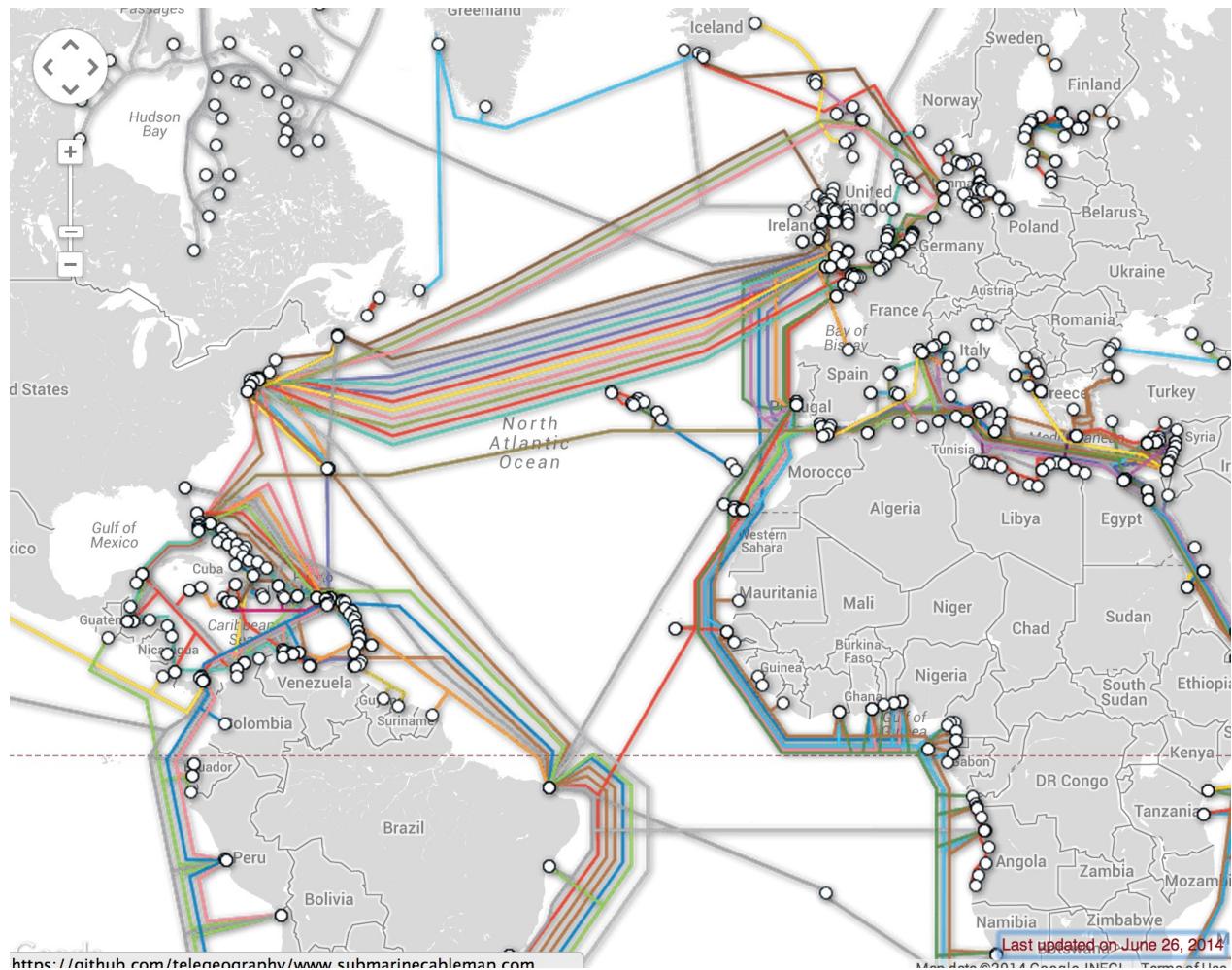
Wie jede andere Datei auch, muss die HTML-Datei irgendwo auf einer Festplatte gespeichert werden. Für das Internet verwenden wir spezielle, leistungsstarke Computer, sogenannte *Server*. An ihnen sind normalerweise weder Bildschirm, Maus oder Tastatur angeschlossen, weil der Hauptzweck der Server darin besteht, Daten zu speichern und zur Verfügung zu stellen. Darum nennt man sie *Server* - sie *bedienen* (serve) dich mit Daten.

OK, aber du willst wissen, wie das Internet aussieht oder?

Wir haben ein Bild gemalt. So sieht es aus:

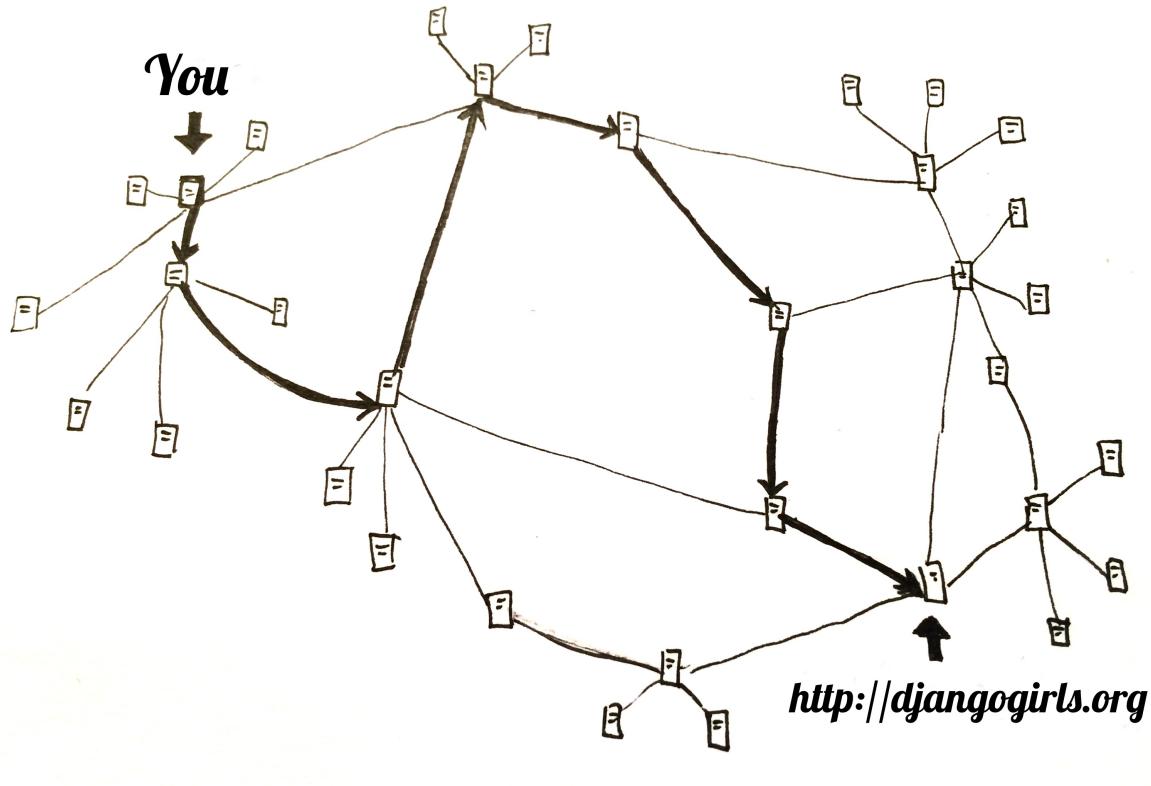


Ziemliches Durcheinander, oder? Eigentlich ist es ein Netzwerk aus verbundenen Maschinen (den oben genannten *Servern*). Hundertausende von Rechnern! Kilometer über Kilometer Kabel rund um die Welt. Auf einer Webseite über Unterseekabel (<https://submarinecablemap.com>) kannst du dir ein Bild von der Komplexität des Netzes machen. Hier ist ein Screenshot der Seite:



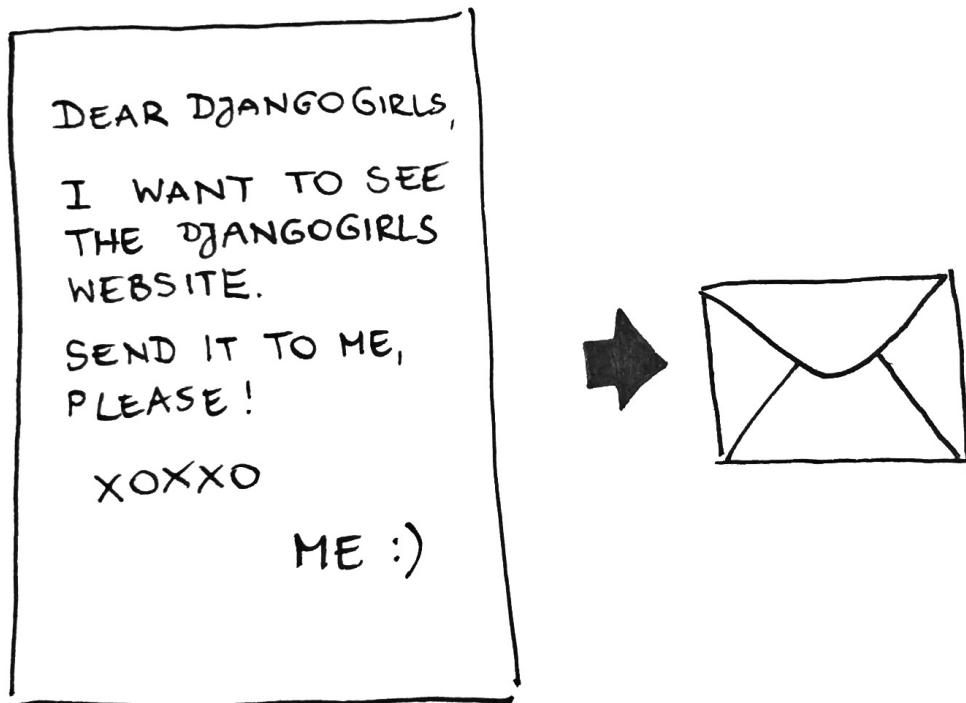
Faszinierend, oder? Es ist jedoch nicht möglich, Kabel zwischen allen Servern des Internets zu schalten. Damit wir eine Maschine (z.B. diejenige, auf welcher [https://django.org](https://.djangoproject.org) abgespeichert ist) erreichen können, muss unsere Anfrage über viele verschiedene andere Maschinen weitergeleitet werden.

Das sieht ungefähr so aus:



Stell dir vor, wenn du <https://djangogirls.org> in den Browser eingibst, würdest du einen Brief versenden, in dem steht; "Hallo Django Girls, ich möchte die djangogirls.org Webseite ansehen. Bitte schick sie mir!"

Der Brief kommt ins Postamt in deiner Nähe. Von da aus gelangt er zu einem anderen Postamt näher an der Zieladresse und näher und näher, bis der Brief zugestellt werden kann. Die einzigartige Sache ist, dass, wenn du mehrere Briefe (*Datenpakete*) zu der selben Adresse abschickst, jeder einzelne Brief durch komplett unterschiedliche Poststellen (*Router*) laufen könnte. Dies hängt davon ab, wie sie an jedem Standort verteilt werden.



So einfach ist das im Prinzip. Du sendest Nachrichten und erwartest eine Antwort. Anstelle von Papier und Stift verwendest du Daten, aber die Idee ist dieselbe!

Anstelle von Adressen mit Straße, Ort und Postleitzahl verwenden wir IP-Adressen. IP steht für Internet Protocol. Dein Computer fragt erst das DNS (Domain Name System), um die (von Menschen besser lesbare) Adresse djangogirls.org in die (besser von Maschinen lesbare) IP-Adresse umzuwandeln. Das DNS ist ein bisschen wie ein almodisches Telefonbuch aus Papier, in dem du den Namen einer Person, die Du kontaktieren willst, suchen und die Telefonnummer und Adresse nachgucken kannst.

Wenn du einen Brief versenden willst, brauchst du spezielle Eigenschaften wie: Postanschrift, Briefmarke etc. Außerdem musst du eine Sprache verwenden, die der Empfänger versteht. Das gleiche gilt für die *Datenpakete*, die du sendest, um eine Website betrachten zu können. Wir verwenden ein Protokoll namens HTTP (Hypertext Transfer Protocol).

Grundsätzlich brauchst du also für eine Website auch einen *Server*, auf dem sie abgelegt ist. Wenn der *Server* eine eingehende *Anforderung* (in einem Brief) empfängt, sendet er deine Website zurück (in einem weiteren Brief).

Da dies hier ein Django-Tutorial ist, fragst du dich vielleicht, was Django in diesem Zusammenhang macht. Wenn dein Server eine Antwort zurück sendet, soll nicht an jeden dasselbe gesendet werden. Es wäre besser, wenn die Antworten individuell personalisiert würden, entsprechend der Anfragen des jeweiligen Briefes, oder? Django hilft dir, diese personalisierten und interessanten Antworten zu erstellen. :)

Genug der Theorie, lass uns loslegen!

# Einführung in die Kommandozeile

**Hinweis:** Im informatics4girls-Workshop verwenden wir Linux. Die Anleitungen für Windows und Mac wurden in dieser Druckversion daher weggelassen. Falls du zuhause daran arbeitst, verwende die online-Version auf [https://tutorial.djangogirls.org/de/intro\\_to\\_command\\_line/](https://tutorial.djangogirls.org/de/intro_to_command_line/), die auch für diese Betriebssysteme alle Informationen enthält.

Aufregend, oder?! In ein paar Minuten wirst du deine erste Zeile Code schreiben! :)

## Erstmal stellen wir dir deine neue Freundin vor: Die Konsole!

Im Folgenden zeigen wir dir, wie du das schwarze Fenster benutzt, das alle Hackerinnen nutzen. Es sieht vielleicht erstmal etwas unheimlich aus, aber es ist nur ein Programm, das darauf wartet, Anweisungen von dir zu bekommen.

**Hinweis:** Bitte beachte, dass wir in dem gesamten Buch die Begriffe "Verzeichnis" und "Ordner" abwechselnd gebrauchen, aber sie stehen für ein und dasselbe.

## Was ist die Konsole?

Das Fenster, welches gewöhnlich die **Kommandokonsole** (command line) oder **Kommandozeilen-Interface** (command-line interface) genannt wird, ist eine textbasierte Applikation zum Betrachten, Bearbeiten und Manipulieren von Dateien auf deinem Computer. Es ist dem Windows Explorer oder Finder auf dem Mac ähnlich, aber ohne die grafische Benutzeroberfläche. Andere Bezeichnungen dafür sind: *CMD, CLI, Prompt (Eingabeaufforderung), Konsole* oder *Terminal*.

## Öffnen der Konsole

Um mit ein paar Experimenten zu beginnen, müssen wir erstmal die Kommandozeile öffnen.

Auf den Computern im PC-Labor geht das mit dem Terminal-Schnellstart-Knopf links.

## Eingabeaufforderung (Prompt)

Du solltest nun ein weißes oder schwarzes Fenster sehen, das auf deine Anweisungen wartet.

Auf einem Mac- oder Linux-Rechner siehst du wahrscheinlich ein `$`, also so:

command-line

```
$
```

Vor jedem Kommando wird das Zeichen `$` oder `>` und ein Leerzeichen vorangestellt, aber du musst das nicht hinschreiben. Dein Computer macht das für dich. :)

Ein kleiner Hinweis: Falls du etwas in der Art wie `C:\Users\ola>` oder `olas-MacBook-Air:~ ola$` sehen solltest, ist das auch 100%ig korrekt.

Der Teil bis und einschließlich `$` oder `>` heißt **Kommandozeilen-Eingabeaufforderung** oder kurz **Eingabeaufforderung**. Sie fordert dich auf, hier etwas einzugeben.

Wenn wir im Tutorial wollen, dass du einen Befehl eingibst, schreiben wir `$` oder `>` mit hin, gelegentlich auch noch die anderen Angaben links davon. Ignoriere den linken Teil und gib nur das Kommando ein, welches rechts der Eingabeaufforderung steht.

## Dein erstes Kommando (YAY!)

Lass uns mit diesem Kommando beginnen:

command-line

```
$ whoami
```

Und dann bestätige mit `Enter`. Das ist unser Ergebnis:

command-line

```
$ whoami  
olasitarska
```

Wie du sehen kannst, hat der Computer gerade deinen Benutzernamen ausgegeben. Toll, was? :)

Versuch, jeden Befehl abzuschreiben und nicht zu kopieren und einzufügen. Auf diese Weise wirst du dir mehr merken!

## Grundlagen

Jedes Betriebssystem hat einen geringfügig anderen Bestand an Befehlen für die Kommandozeile, beachte daher die Anweisungen für dein Betriebssystem. Lass uns das ausprobieren.

### Aktuelles Verzeichnis

Es wäre schön zu sehen, wo wir uns befinden, oder? Lass uns nachsehen. Gib diesen Befehl in die Konsole ein und bestätige ihn mit `Enter`:

command-line

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

Hinweis: 'pwd' steht für 'print working directory' (zeige derzeitiges Arbeitsverzeichnis).

Du wirst wahrscheinlich etwas Ähnliches auf deinem Gerät sehen. Wenn du die Konsole öffnest, befindest du dich normalerweise im Heimverzeichnis deines Benutzers.

---

## Mehr über ein Kommando lernen

Viele Befehle, die du in der Kommandozeile nutzen kannst, haben eine eingebaute Hilfe, die du anzeigen und lesen kannst! Zum Beispiel kannst du etwas über den eben verwendeten Befehl lernen:

**Command help: OS X and Linux**

OS X und Linux haben einen `man`-Befehl, mit dem du die Hilfe über die Kommandos aufrufen kannst. Gib `man pwd` ein und schau, was angezeigt wird oder setze `man` vor andere Kommandos und sieh dir deren Hilfe an. Das Ergebnis von `man` wird in der Regel seitenweise ausgegeben. Du kannst die Leertaste benutzen, um auf die nächste Seite zu gelangen und `q` (für engl. "quit", was "verlassen"/"rausgehen" heisst), um die Hilfeseiten zu schließen.

## Anzeigen von Dateien und Unterordnern

Nun, was befindet sich in deinem Verzeichnis? Es wäre toll, das herauszufinden. Lass uns mal schauen:

### List files and directories: OS X and Linux

command-line

```
$ ls
Anwendungen
Desktop
Downloads
Musik
...
```

---

## Wechseln des Verzeichnisses

Lass uns jetzt zu unserem Desktop-Verzeichnis wechseln:

### Change current directory: Linux

command-line

```
$ cd Desktop
```

Wenn dein Linux-Benutzerkonto auf Deutsch eingestellt ist, kann es sein, dass auch der Name des Desktop-Verzeichnisses übersetzt ist. Wenn dem so ist, musst du im obigen Befehl `desktop` durch den übersetzten Verzeichnisnamen `Schreibtisch` ersetzen.

Schau, ob das Wechseln des Verzeichnisses funktioniert hat:

### Check if changed: OS X and Linux

command-line

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

Passt!

Profi-Tipp: Wenn du `cd D` tippst und dann `tab` auf deiner Tastatur drückst, wird die Kommandozeile automatisch den Rest des Namens vervollständigen, wodurch du schneller navigieren kannst. Wenn es mehr als einen Ordner gibt, dessen Name mit "D" beginnt, drücke die `tab`-Taste zweimal, um eine Liste der Möglichkeiten anzuzeigen.

---

## Erstellen eines Verzeichnisses

Wie wär's damit, ein Übungsverzeichnis auf deinem Desktop zu erstellen? So kannst du das tun:

### Create directory: OS X and Linux

command-line

```
$ mkdir practice
```

Dieser kleine Befehl erstellt einen Ordner mit dem Namen `practice` auf deinem Desktop. Du kannst nun überprüfen, ob er wirklich dort ist, indem du auf deinem Desktop nachschaust oder indem du den Befehl `ls` oder `dir` ausführst! Versuch es. :)

Profi-Tipp: Wenn du die selben Befehle nicht immer wieder und wieder schreiben willst, verwende die `Pfeil aufwärts` - und `Pfeil abwärts` -Tasten deiner Tastatur, um durch die zuletzt verwendeten Befehle zu blättern.

---

## Übung!

Eine kleine Herausforderung für dich: Erstelle in deinem neu erstellten `practice`-Ordner ein Verzeichnis namens `test`. (Verwende dazu die Kommandos `cd` und `mkdir`.)

## Lösung:

### Exercise solution: OS X and Linux

command-line

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

Glückwunsch! :)

---

## Aufräumen

Wir wollen kein Chaos hinterlassen, also lass uns das bislang Geschaffene wieder löschen.

Zuerst müssen wir zurück zum Desktop wechseln:

### Clean up: OS X and Linux

command-line

```
$ cd ..
```

Durch Verwendung von `..` mit dem `cd`-Kommando wechselst du von deinem aktuellen Verzeichnis zum übergeordneten Verzeichnis (dies ist das Verzeichnis, das das aktuelle Verzeichnis enthält).

Schau nach, wo du gerade bist:

### Check location: OS X and Linux

command-line

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

Jetzt ist es an der Zeit, dein `practice`-Verzeichnis zu löschen:

**Achtung:** Wenn du Daten mit `del`, `rmdir` oder `rm` löschst, kannst du das nicht mehr rückgängig machen, das bedeutet, *die gelöschten Dateien sind für immer weg!* Sei also sehr vorsichtig mit diesem Befehl.

### Delete directory: Windows Powershell, OS X and Linux

command-line

```
$ rm -r practice
```

Geschafft! Lass uns schauen, ob es wirklich gelöscht ist:

### Check deletion: OS X and Linux

command-line

```
$ ls
```

## Beenden

Das wärs fürs Erste. Du kannst nun beruhigt deine Konsole schließen. Lass es uns wie die Hacker machen, okay? :)

### Exit: OS X and Linux

command-line

```
$ exit
```

Cool, was? :)

## Zusammenfassung

Hier ist eine Zusammenfassung einiger nützlicher Kommandos:

Befehl (Windows)	Befehl (Mac OS / Linux)	Beschreibung	Beispiel
exit	exit	Fenster schließen	<b>exit</b>
cd	cd	Verzeichnis wechseln	<b>cd test</b>
cd	pwd	aktueller Verzeichnis anzeigen	<b>cd</b> (Windows) oder <b>pwd</b> (Mac OS / Linux)
dir	ls	Unterordner/Dateien zeigen	<b>dir</b>
copy	cp	Datei kopieren	<b>copy c:\test\test.txt c:\windows\test.txt</b>
move	mv	Datei verschieben	<b>move c:\test\test.txt c:\windows\test.txt</b>
mkdir	mkdir	neues Verzeichnis erstellen	<b>mkdir testdirectory</b>
rmdir (oder del)	rm	Datei löschen	<b>del c:\test\test.txt</b>
rmdir /S	rm -r	Verzeichnis löschen	<b>rm -r testdirectory</b>
[CMD] /?	man [CMD]	Hilfe für ein Kommando aufrufen	<b>cd/?</b> (Windows) oder <b>man cd</b> (Mac OS / Linux)

Das sind nur sehr wenige der Befehle, welche du in deiner Konsole verwenden kannst, aber du wirst heute nicht mehr brauchen.

Falls du neugierig bist, findest du auf [ss64.com](http://ss64.com) eine vollständige Übersicht über alle Kommandozeilen-Befehle für alle Betriebssysteme.

## Fertig?

Lass uns mit Python anfangen!

# Lass uns mit Python anfangen

Wir sind endlich da!

Aber lass uns zuerst erklären, was Python ist. Python ist eine sehr beliebte Programmiersprache, die du zur Erstellung von Webseiten, Spielen, wissenschaftlichen Programmen, Computergrafiken und vielem mehr verwenden kannst.

Python entstand in den späten 1980ern mit dem Hauptziel, von Menschen lesbar zu sein (nicht nur von Computern). Darum sieht es einfacher aus als andere Programmiersprachen. Aber keine Sorge - Python ist auch sehr mächtig!

## Python-Installation

**Hinweis** Auf den Computern im PC-Labor ist Python bereits installiert.

Falls du das Tutorial zuhause bearbeitest, folge der Anleitung auf

[https://tutorial.djangogirls.org/de/python\\_installation/](https://tutorial.djangogirls.org/de/python_installation/)

# Der Code-Editor

Für die Leser zu Hause: Dieses Kapitel wird im Video [Installing Python & Code Editor](#) behandelt.

Gleich geht's los! Du wirst deine erste Zeile Programmcode schreiben! Daher ist es jetzt an der Zeit, einen entsprechenden Editor zu starten!

**Hinweis** Auf den Computern im PC-Labor sind einige Editoren bereits installiert.

Falls du das Tutorial zuhause bearbeitest: Auf [https://tutorial.djangogirls.org/de/code\\_editor/](https://tutorial.djangogirls.org/de/code_editor/) sind die Download-Seiten der hier genannten Editoren verlinkt.

Es gibt viele verschiedene Editoren. Welcher für dich am besten ist, ist weitestgehend Geschmackssache. Die meisten Python-Programmiererinnen verwenden komplexe, aber extrem leistungsfähige IDEs (Integrated Development Environments), z. B. PyCharm. Für Anfängerinnen sind diese jedoch weniger gut geeignet. Unsere Empfehlungen sind ebenso leistungsfähig, aber viel einfacher zu bedienen.

Unsere Vorschläge siehst du unten. Aber fühl dich ganz frei, deine Trainerin zu fragen, was ihre Vorlieben sind - wenn sie sich mit dem Editor auskennt, wird es leichter sein, Hilfe zu erhalten.

## Gedit

Gedit ist ein kostenloser Open-Source-Editor. Es gibt ihn für alle Betriebssysteme.

## Sublime Text 3

Sublime Text ist ein sehr beliebter Editor, nutzbar für einen kostenlosen Testzeitraum. Er ist einfach zu installieren und zu verwenden, und er ist für alle Betriebssysteme verfügbar.

## Atom

Atom ist ein weiterer beliebter Editor. Er ist gratis, Open-Source und für Windows, OS X und Linux verfügbar. Atom wird von [GitHub](#) entwickelt.

## Welchen Code-Editor soll ich wählen?

Falls du nicht bereits eine Vorliebe hast, verwende Atom! Auf den Computern im PC-Labor lässt der sich über einen der Schnellstart-Knöpfe in der Leiste links starten.

## Warum installieren wir einen Code-Editor?

Vielleicht wunderst du dich, warum wir so spezielle Code-Editor-Software verwenden, statt einfach etwas wie Word oder Notepad zu benutzen.

Erstens muss Code "plain text" (unformatierter Text) sein. Das Problem mit Programmen wie Word und Textedit ist, dass sie nicht "plain text" sondern "rich text" (mit Schriftarten und Formatierungen) produzieren und besondere Formate wie RTF (Rich Text Format) verwenden.

Ein weiterer Grund ist, dass Code-Editoren (bisweilen auch Programmier- oder Text-Editoren genannt) auf das Bearbeiten von Programm-Code spezialisiert sind und Funktionen aufweisen, die normale Textverarbeitungen nicht haben. Beispielsweise sogenanntes "Syntax-Highlighting", also farbliches Hervorheben bestimmter Code-Stellen, oder auch das automatische Schließen von Klammern und vieles mehr.

Einiges davon werden wir später in Aktion sehen. Glaub uns: es wird nicht lange dauern, bis du deinen Code-Editor nicht mehr missen möchtest. :)

# Einführung in Python

Fangen wir an, schreiben wir Code!

## Der Python-Prompt

Um Python zu starten, musst du an die *Kommandozeile* deines Computers. Wie das geht, weißt du bereits - denn du hast es im Kapitel [Einführung in die Kommandozeile](#) gelernt.

Also öffne die Konsole, dann fangen wir an.

Wir wollen eine Python Konsole öffnen, also tippe unter Windows `python` oder im Mac OS/Linux Terminal `python3` und drücke `Enter`.

command-line

```
$ python3
Python 3.6.1 (...)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

## Dein erster Python-Befehl!

Nach Eingabe von `python3` in der Konsole ändert sich das Prompt-Zeichen zu `>>>`. Für uns bedeutet das, dass wir ab nun nur noch Python-Code eingeben können. Den Python-Prompt `>>>` musst du nicht jedesmal eingeben - dies macht Python für dich.

Wenn du die Python-Konsole wieder verlassen möchtest, gib `exit()` ein oder nutze das Tastatur-Kürzel `Strg + Z` unter Windows bzw. `Strg + D`, wenn du einen Mac hast oder Linux verwendest. Dann bist du wieder in der normalen Konsole und der Python-Prompt `>>>` ist weg.

Fürs Erste bleiben wir in der Python Konsole, wir wollen mehr darüber lernen. Lass uns mit ein wenig Mathematik anfangen, gib `2 + 3` ein und drück `enter`.

command-line

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

Cool! Schon ist das Ergebnis da. Python kann rechnen! Probier einfach einige andere Befehle aus, wie z.B.:

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

Um Potenzen zu berechnen, sagen wir  $2$  hoch  $3$ , müssen wir Folgendes eingeben:

command-line

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

Spiel ein wenig herum, dann machen wir weiter. :)

Wie du siehst, kann Python richtig toll rechnen. Aber Python kann noch viel mehr ...

## Strings

Strings sind Zeichenketten. Das ist eine Folge von Buchstaben, die von Anführungszeichen umgeben sind. Gib einfach mal deinen Namen ein (bei mir "Ola"):

command-line

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

Nun hast du deinen ersten String erzeugt! Dies ist eine Folge von Zeichen (also nicht nur Buchstaben, wie ich oben schrieb, sondern Zeichen aller Art), die von einem Computer verarbeitet werden können. Ein String muss stets mit dem gleichen Zeichen beginnen und enden. Dies kann entweder ein einzelnes Gänsefußchen sein ( ' ) oder ein doppeltes ( " ), da gibt es keinen Unterschied! Die Anführungszeichen zeigen Python nur an, dass alles dazwischen ein String ist.

Strings können zusammengesetzt werden. Versuch es einmal:

command-line

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

Du kannst Strings auch vervielfältigen:

command-line

```
>>> "Ola" * 3  
'olaolaOla'
```

Brauchst du einen Apostroph in einem String, so hast du zwei Möglichkeiten.

Du kannst für den String doppelte Anführungszeichen verwenden:

command-line

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

oder du kannst den Apostroph mit einem Backslash (``) markieren:

command-line

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

Toll, was? Um deinen Namen in Großbuchstaben anzuzeigen, gib Folgendes ein:

command-line

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

Hier hast du die `upper`-**Methode** auf den String angewendet! Eine Methode (wie `upper()`) ist eine Abfolge von Anweisungen, die Python für ein gegebenes Objekt (hier `"Ola"`) ausführt, wenn sie aufgerufen wird.

Nehmen wir an, du möchtest die Zahl der Buchstaben in deinem Namen wissen. Auch dafür gibt es eine **Funktion!**

command-line

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

Nun fragst du dich sicher, warum du manchmal eine Methode mit einem `.` am Ende des Strings (wie bei `"Ola".upper()`) schreibst und manchmal eine Funktion direkt aufrufst und den String dahinter in Klammern setzt? Im ersten Fall gehören solche Methoden, wie `upper()`, zu Objekten (hier: ein String) und funktionieren auch nur bei diesen. In solchen Fällen bezeichnen wir eine Funktion als **Methode**. Andere Funktionen sind dagegen allgemeiner und können auf unterschiedliche Datentypen angewendet werden, wie beispielsweise `len()`. Daher übergeben wir `"Ola"` als Parameter an die `len` Funktion.

## Zusammenfassung

Ok, genug über Strings. Bisher haben wir Folgendes kennengelernt:

- **Der Prompt** - Wenn wir beim Python-Prompt Anweisungen (oder Programm-Code) in Python eingeben, dann erhalten wir auch Ergebnisse in Python. Man sagt zu dieser Python-Umgebung auch "Python-Shell".
- **Zahlen und Strings** - In Python nutzen wir Zahlen für Berechnungen und Strings für Text-Objekte.
- **Operatoren**, wie `+` und `*`, verarbeiten mehrere Werte und erzeugen als Ergebnis einen neuen Wert.
- **Funktionen** - wie `upper()` und `len()`, tun etwas mit Objekten (in unserem Beispiel ändern sie diese, wie bei `upper()`), oder sie geben eine Eigenschaft zurück, wie bei `len()`.

Das sind Grundlagen jeder Programmiersprache, die Du lernen wirst. Bist Du bereit für mehr? Bestimmt!

## Fehler

Probieren wir etwas Neues: Errors. Können wir die Länge einer Zahl auf die gleiche Weise ermitteln, wie die Länge eines Namens? Gib dazu `len(304023)` ein und drücke auf Enter:

command-line

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

Wir haben unsere erste Fehlermeldung (Error) erhalten! Mit dem Icon teilen wir dir in diesem Tutorial jeweils mit, dass der einzugebende Code nicht wie erwartet funktionieren wird. Fehler zu machen (selbst absichtlich) ist ein wesentlicher Teil beim Lernen!

Unser erster Fehler sagt, dass Objekte vom Typ "int" (Integers, das sind ganze Zahlen) keine Länge haben. Was also nun? Vielleicht sollten wir unsere Zahl als String schreiben? Denn bei Strings funktioniert es ja, wie wir wissen.

command-line

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

Ja, das funktioniert! Hier haben wir die `str`-Funktion innerhalb der Funktion `len` aufgerufen. `str()` konvertiert alles zu einem String.

- Die `str`-Funktion wandelt den übergebenen Wert in einen **String** um.
- Die `int`-Funktion wandelt den übergebenen Wert in einen **Integer** um.

Wichtig: Zwar können wir Zahlen in Text umwandeln, aber nicht immer auch Text in Zahlen - was beispielsweise sollte `int('hello')` ergeben?

## Variablen

Ein wichtiger Bestandteil beim Programmieren sind Variablen. Eine Variable ist einfach ein Name für etwas, das wir später unter genau diesem Namen wieder verwenden können. Programmiererinnen nutzen Variablen, um Daten zu speichern, den Code lesbar zu halten und um sich nicht immer alles merken zu müssen.

Lass uns eine Variable mit der Bezeichnung `name` anlegen:

command-line

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

Wir geben ein: name ist gleich "Ola".

Du hast sicher schon bemerkt, dass Python diesmal kein Ergebnis zurückgegeben hat. Woher sollen wir nun wissen, dass es die Variable jetzt auch tatsächlich gibt? Gib `name` ein und drücke wieder auf `Enter`:

command-line

```
>>> name
'ola'
```

Hurra! Deine erste Variable :)! Nun kannst du auch stets ändern, was sie enthalten soll:

command-line

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

Du kannst die Variable auch in Funktionen verwenden:

command-line

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

Das ist toll, oder? Variablen können alles enthalten, also auch Zahlen. Versuche Folgendes:

command-line

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

Was aber, wenn wir für eine Variable den falschen Namen verwenden? Uns einfach vertippen. Hast du schon eine leise Ahnung, was dann passiert? Probieren wir es aus!

command-line

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

Ein Fehler! Wie du siehst, kennt Python verschiedene Arten von Fehlern. In unserem Fall hier ist es ein **NameError**. Python liefert diesen Fehler immer dann, wenn du versuchst, eine Variable zu verwenden, die es noch gar nicht gibt. Wenn du einen solchen Fehler erhältst, prüfe einfach in deinem Code, ob du dich irgendwo vertippt hast.

Spiel einfach ein wenig rum und schaue, was alles so passiert.

## Die print-Funktion

Gib einmal Folgendes ein:

command-line

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

Wenn du in der zweiten Zeile `name` eintippst, dann gibt der Python-Interpreter die *String-Darstellung* (engl. 'representation') der Variable 'name' aus. In unserem Beispiel die Buchstaben M-a-r-i-a, umschlossen von einfachen Anführungszeichen ('). Wenn du hingegen `print(name)` schreibst, dann gibt Python den Inhalt der Variablen ohne die Anführungszeichen zurück, was etwas schöner aussieht.

Wie wir später sehen werden, ist `print()` auch recht nützlich, wenn wir etwas aus Funktionen heraus ausgeben möchten oder auch eine Ausgabe über mehrere Zeilen darstellen wollen.

## Listen

Außer Strings (Zeichenketten) und Integern (ganze Zahlen) hat Python noch viele andere Arten von Datentypen. Von denen wollen wir uns nun **Listen** anschauen. Listen sind genau das, was du wahrscheinlich schon vermutest: Es sind Objekte, die Listen von anderen Objekten enthalten. :)

Legen wir los und erzeugen eine Liste:

command-line

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

Ja, dies ist eine leere Liste. Für uns noch nicht sehr nützlich. Legen wir nun eine Liste von Lottozahlen an. Da wir uns nicht dauernd wiederholen wollen, ordnen wir diese Liste auch direkt einer Variablen zu:

command-line

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

So, nun haben wir eine Liste mit Lottozahlen! Was aber können wir damit tun? Zuerst einmal wollen wir feststellen, wie viele Zahlen in ihr enthalten sind. Hast du schon eine Idee, wie dies geht? Klar, das weißt du ja bereits!

command-line

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

Genau! `len()` liefert die Anzahl von Objekten in einer Liste zurück. Praktisch, nicht wahr? Nun wollen wir die Liste sortieren:

command-line

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

Diese Anweisung gibt nichts zurück, sie hat aber die Reihenfolge der Objekte in der Liste geändert. Um zu sehen, was passiert ist, müssen wir die Liste ausgeben:

command-line

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

Wie du siehst, sind die Zahlen in der Liste nun aufsteigend sortiert. Super!

Aber vielleicht wollten wir es genau andersherum haben? Nichts leichter als das!

command-line

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

Einfach, oder? Du kannst auch etwas zu deiner Liste hinzufügen:

command-line

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

Falls du nicht immer die gesamte Liste, sondern beispielsweise nur den ersten Eintrag sehen möchtest, kannst du dafür **Indizes** benutzen. Ein Index gibt die Stelle innerhalb einer Liste an, die uns interessiert. Programmierer bevorzugen es, bei 0 mit dem Zählen anzufangen. Also hat das erste Objekt in deiner Liste den Index 0, das nächste die 1 und so weiter. Gib einmal Folgendes ein:

command-line

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

Wie du siehst, kannst du auf die einzelnen Objekte in deiner Liste zugreifen, indem du den Namen der Liste verwendest und anschließend den Index in eckigen Klammern anfügst.

Um etwas aus deiner Liste zu löschen, musst du die **Indizes** wie gerade gelernt benutzen und die `pop()`-Methode. Lass uns ein Beispiel versuchen und das festigen, was wir zuvor gelernt haben; wir werden die erste Nummer aus unserer Liste löschen.

command-line

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

Das hat doch super geklappt!

Probier einmal andere Indizes aus: 6, 7, 1000, -1, -6 oder -1000 und versuch dir das Ergebnis vorzustellen, bevor du den jeweiligen Index verwendest. Sind die Ergebnisse sinnvoll?

Eine Liste aller Methoden, die du auf Listen anwenden kannst, findest du in der Python-Dokumentation:

<https://docs.python.org/3/tutorial/datastructures.html>

## Dictionaries

Ein Wörterbuch (von nun an mit dem englischen Begriff 'Dictionary' bezeichnet) verhält sich ähnlich wie eine Liste, jedoch greifen wir auf die enthaltenen Objekte nicht mit einem Index, sondern mit einem Schlüssel zu (auf englisch 'key', und auch hier verwenden wir im weiteren den englischen Begriff). Ein 'key' kann ein String oder eine Zahl sein. Ein leeres Dictionary legen wir wie folgt an:

command-line

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

Und schon hast du ein leeres Dictionary erstellt. Super!

Nun gib einmal Folgendes ein (verwende statt 'Ola' usw. deine eigenen Informationen):

command-line

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

Du hast nun soeben die Variable mit dem Namen `participant` angelegt, die ein Dictionary mit drei key-value Paaren enthält (values, also Werte, sind die Objekte in einem Dictionary, - aber auch hier bleiben wir beim englischen Begriff):

- Der key `name` verweist auf den value `'Ola'` (welches ein `string` Objekt ist),
- `country` verweist auf `'Poland'` (ebenfalls ein `string` Objekt),
- und `favorite_numbers` schließlich verweist auf `[7, 42, 92]` (eine `Liste` mit drei Zahlen).

Auf die einzelnen Objekte in einem Dictionary kannst du wie folgt zugreifen:

command-line

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

Also ganz ähnlich wie bei einer Liste. Aber statt dir einen Index merken zu müssen, benutzt du bei einem Dictionary einfach einen key (hier: den String `'name'`).

Was aber geschieht, wenn wir Python nach dem Wert eines keys fragen, den es gar nicht gibt? Errätst du es schon? Probieren wir es einfach aus und schauen was passiert!

command-line

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

Ah, wieder ein Fehler! Diesmal ein **KeyError**. Python hilft uns auch hier und sagt uns, dass es den key `'age'` in diesem Dictionary gar nicht gibt.

Wenn du zwischen Dictionaries und Listen wählen kannst, wann sollte welche Datenstruktur verwendet werden? Das ist eine gute Frage, über die es sich nachzudenken lohnt; und vielleicht möchtest du dies kurz tun, bevor du weiterliest.

- Du brauchst nur eine geordnete Folge von Elementen? Dann wähle eine Liste.
- Du brauchst eine Sammlung von Elementen, auf die du später einzeln, gezielt und effizient mit Hilfe eines Namens (d.h. keys) zugreifen kannst? Dann wähle ein Dictionary.

Dictionaries sind, so wie auch Listen, *mutable*, d. h. nachträglich veränderbar. So kannst du bei Dictionaries später noch weitere key-value Paare hinzufügen:

command-line

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

Wie bei Listen können wir auch bei Dictionaries die `len()`-Funktion verwenden, um die Zahl der enthaltenen Einträge (das sind die key-value Paare) zu ermitteln. Probier es gleich aus und tippe dieses Kommando ein:

command-line

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

Wir hoffen, dass das Alles für dich bisher Sinn ergibt. :) Bist du bereit für mehr Spaß mit Dictionaries? Machen wir weiter.

Zum Löschen von Elementen kannst du den `pop()`-Befehl verwenden. Nehmen wir an, du möchtest den Eintrag mit dem key `'favorite_numbers'` entfernen, dann tippe:

command-line

```
>>> participant.pop('favorite_numbers')
[7, 42, 92]
>>> participant
{'country': 'Poland', 'favorite_language': 'Python', 'name': 'Ola'}
```

Wie du an der Ausgabe erkennst, ist nun das key-value Paar von `'favorite_numbers'` gelöscht.

Ebenso kannst du auch den Wert eines bestehenden Eintrages ändern:

command-line

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

Wie du siehst, hast du nun im key-value Paar mit dem key `'country'` den Wert von `'Poland'` nach `'Germany'` geändert. :) Hurra! Schon wieder was gelernt.

## Zusammenfassung

Großartig! Inzwischen hast du schon einiges über Programmierung gelernt und die folgenden Dinge sind dir vertraut:

- **Errors** - Du weißt, wie sie zu lesen sind und dass Python sie dann ausgibt, wenn es eine Anweisung von dir nicht ausführen kann.
- **Variablen** - sind Namen für Objekte, die dir dabei helfen, deinen Code leichter zu schreiben und ihn dabei auch gut lesbar zu halten.
- **Listen** - können Objekte in einer geordneten Reihenfolge speichern.
- **Dictionaries** - speichern Objekte als key-value Paare.

Schon gespannt auf den nächsten Teil? :)

## Vergleichen

Ein großer Teil beim Programmieren besteht darin, Dinge zu vergleichen. Was lässt sich am besten vergleichen? Zahlen! Schauen wir uns mal an, wie das funktioniert:

command-line

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

Hier haben wir Python einige Zahlen zum Vergleichen gegeben. Wie du siehst, kann Python nicht nur die Zahlen vergleichen, sondern auch die Ergebnisse von Berechnungen. Cool, nicht wahr?

Womöglich wunderst du dich aber über die beiden `==` Gleichheitszeichen zum Vergleich, ob zwei Zahlen den selben Wert haben? Ein einfaches Gleichheitszeichen `=` verwenden wir bereits, um Variablen bestimmte Werte zuzuweisen. Da beim Programmieren alle Anweisungen eindeutig sein müssen, benötigst du in Python daher stets zwei `==` Zeichen, um Dinge auf Gleichheit zu testen. Wir können auch feststellen, ob Werte unterschiedlich sind. Dafür verwenden wir das Symbol `!=`, wie im obigen Beispiel.

Nun noch zwei weitere Vergleiche:

command-line

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

`>` und `<` sind klar, was aber sollen `>=` und `<=` bedeuten? Vergleiche liest du folgendermaßen:

- `x > y` bedeutet: x ist größer als y
- `x < y` bedeutet: x ist kleiner als y
- `x <= y` bedeutet: x ist kleiner oder gleich y
- `x >= y` bedeutet: x ist größer oder gleich y

Sensationell! Lust auf mehr? Dann probier das:

command-line

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

Du kannst Python beliebig viele Vergleiche vornehmen lassen und wirst ein Ergebnis erhalten. Das ist wirklich cool, oder?

- **and** - wenn Du den `and`-Operator verwendest, müssen beide Vergleiche True (d.h. wahr) ergeben, damit das Gesamtergebnis auch True ist
- **or** - wenn Du den `or`-Operator verwendest, genügt es, wenn einer der beiden Vergleiche True ergibt, damit das Gesamtergebnis True ergibt

Die Redewendung "Äpfel mit Birnen zu vergleichen" hast du bestimmt schon einmal gehört. Machen wir dies doch einmal in Python:

command-line

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

Unterschiedliche Dinge, hier die Datentypen Zahlen (`int`) und Strings (`str`), lassen sich auch in Python nicht miteinander vergleichen. In solch einem Fall liefert uns Python einen **TypeError** und sagt uns, dass diese zwei Datentypen nicht miteinander verglichen werden können.

## Boolean

Du hast gerade einen neuen Typ von Python-Objekten kennen gelernt. Er heisst **Boolean**.

Es gibt zwei Boolean-Objekte:

- True (wahr)
- False (falsch)

Damit Python diese beiden Werte versteht, musst du sie auch genau so schreiben (den ersten Buchstaben groß, alle weiteren klein). **true**, **TRUE** und **tRUE** funktionieren nicht – nur **True** ist korrekt. (Dasselbe gilt auch für False.)

Auch Booleans können Variablen zugewiesen werden:

command-line

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

Auch Folgendes geht:

command-line

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

Übe ein wenig, indem du mit Booleans rumspielst, zum Beispiel mit diesen Anweisungen:

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

Glückwunsch! Booleans sind echt eine der coolsten Features beim Programmieren und du hast gerade gelernt, damit umzugehen!

## Speicher es!

Bisher haben wir den Python-Code nur im Interpreter eingegeben, wodurch wir immer nur eine Zeile Code auf einmal ausführen konnten. Richtige Programme dagegen werden in Dateien gespeichert und, je nach Programmiersprache, durch einen **Interpreter** ausgeführt oder durch einen **Compiler** übersetzt. Unseren bisherigen Code haben wir dagegen im Python-**Interpreter** Zeile für Zeile eingegeben und einzeln ausgeführt. Für die nächsten Beispiele brauchen wir mehr als eine Zeile, daher werden wir nun:

- Den Python-Interpreter beenden
- Einen Code-Editor unserer Wahl öffnen
- Code eingeben und diesen in einer Python-Datei sichern
- Und diesen dann laufen lassen!

Um den Python-Interpreter zu beenden, nutze die `exit()`-Funktion

command-line

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

Nun siehst du wieder den normalen Kommandozeilen-Prompt.

Im Kapitel [Code-Editor](#) haben wir uns bereits einen Code-Editor ausgesucht. Nun öffnen wir den Code-Editor und schreiben folgenden Code in eine neue Datei (wenn du ein Chromebook benutzt, dann erstelle eine neue Datei in der Cloud-IDE, öffne sie und du befindest dich automatisch im integrierten Code-Editor):

editor

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

Da du nun schon einige Python-Erfahrung hast, schreibe ein wenig Code mit dem, was du bislang gelernt hast.

Als nächstes wollen wir diesen Code in einer Datei mit einem aussagekräftigen Namen speichern. Lass uns die Datei **python\_intro.py** nennen und auf dem Desktop speichern. Wir können der Datei jeden Namen geben, den wir wollen, aber es ist wichtig sicherzustellen, dass der Dateiname auf **.py** endet. Die Erweiterung **.py** gibt unserem Betriebssystem an, dass dies ein **Python executable file** ist und Python diese ausführen kann.

**Hinweis:** Du wirst eines der coolsten Eigenschaften von Code-Editoren bemerken: Farben! In der Python-Konsole hatte alles die gleiche Farbe. Der Code-Editor dagegen sollte dir nun die `print`-Funktion in einer anderen Farbe anzeigen als der von ihr auszugebende Text. Dies wird "Syntax Hervorhebung" ("syntaxhighlighting") genannt und ist ein wirklich sehr nützliches Werkzeug beim Programmieren. Die Farbe von Dingen gibt dir Hinweise auf z.B. nicht geschlossene Zeichenfolgen oder Tippfehler in einem Schlüsselwort (wie das `def` in einer Funktion, das wir weiter unten sehen werden). Dies ist einer der Gründe, warum wir Code-Editoren verwenden. :)

Nun, da die Datei gesichert ist, wollen wir sie ausführen! Nutze, was du bisher über die Kommandozeile (das mit dem Prompt) gelernt hast, um in der Konsole in das Desktop-Verzeichnis zu wechseln.

### Change directory: OS X

Auf einem Mac sieht das etwa so aus:

command-line

```
$ cd ~/Desktop
```

### Change directory: Linux

Unter Linux ist es ähnlich:

command-line

```
$ cd ~/Desktop
```

(Denk daran, dass das "Desktop"-Verzeichnis bei dir "Schreibtisch" heißen kann.)

### Change directory: Windows Command Prompt

In der Eingabeaufforderung von Windows wird's so sein:

command-line

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

### Change directory: Windows Powershell

Und in der Powershell von Windows so:

command-line

```
> cd $Home\Desktop
```

Wenn du nicht weiterkommst, frag' um Hilfe. Denn genau dafür sind die Coaches da!

Benutze jetzt Python, um den Code in der Datei auszuführen:

command-line

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

Hinweis: Unter Windows gibt es den 'python3'-Befehl nicht. Verwende stattdessen 'python', um die Datei auszuführen:

command-line

```
> python python_intro.py
```

Prima! Du hast soeben dein erstes Python-Programm aus einer Datei heraus ausgeführt. Großartig, oder?

Nun wollen wir uns einem wichtigen Teil der Programmierung zuwenden:

## Wenn ... sonst-wenn ... sonst (If ... elif ... else)

Oft sollen manche Programmteile nur ausgeführt werden, wenn bestimmte Vorbedingungen erfüllt sind. Dafür gibt es in Python sogenannte **if-Anweisungen**.

Nun ändere den Code in deiner **python\_intro.py** Datei:

python\_intro.py

```
if 3 > 2:
```

Würden wir das nun speichern und anschließend ausführen, würden wir einen Fehler erhalten:

command-line

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

Python erwartet hier noch weiteren Programmcode, der ausgeführt werden soll, wenn die Bedingung `3 > 2` wahr ist (also `True` ergibt). Versuchen wir, Python "It works!" ausgeben zu lassen. Ändere den Code in **python\_intro.py** zu:

python\_intro.py

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

Du fragst dich nun, warum wir die angefügte Zeile mit 4 Leerzeichen eingerückt haben? Damit teilen wir Python mit, dass dieser Code ausgeführt werden soll, wenn die vorhergehende Bedingung True ergeben hat. Du könntest auch eine andere Anzahl von Leerzeichen wählen, aber fast alle Python-Programmierer nutzen 4 Leerzeichen, damit's gut aussieht. Ein einfaches Tab zählt auch wie 4 Leerzeichen, sofern dies in deinem Editor so eingestellt ist. Wenn du dich einmal entschieden hast, bleib dabei! Wenn du mit 4 Leerzeichen angefangen hast, solltest du alle weiteren Einrückungen auch mit 4 Leerzeichen machen, anderweitig könnte das Probleme verursachen.

Nun sichere die Datei und führe sie noch einmal aus:

command-line

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

Hinweis: Denk daran, dass Windows den 'python3'-Befehl nicht kennt. Falls du auf Windows arbeitest, verwende ab jetzt immer 'python', wenn in dieser Anleitung 'python3' steht.

## Was passiert, wenn eine Bedingung nicht wahr (not True) ist?

In den vorigen Beispielen wurde Code ausgeführt, wenn eine vorhergehende Bedingung True (wahr) ergab. Aber Python kennt auch `elif` - und `else` -Anweisungen:

python\_intro.py

```
if 5 > 2:  
    print('5 ist wirklich größer als 2')  
else:  
    print('5 ist nicht größer als 2')
```

Wenn dies ausgeführt wird, wird es anzeigen:

command-line

```
$ python3 python_intro.py  
5 ist wirklich größer als 2
```

Wenn 2 größer als 5 wäre, würde die zweite Anweisung (die nach dem `else`) ausgeführt. Schauen wir uns nun an, wie `elif` funktioniert:

python\_intro.py

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

und ausgeführt erhalten wir:

command-line

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

Hast du bemerkt, was passiert ist? `elif` lässt dich zusätzliche Bedingungen hinzufügen, die geprüft werden, falls die vorherige fehlschlägt.

Du kannst so viele `elif`-Bedingungen nach der anfänglichen `if`-Anweisung hinzufügen, wie du magst. Zum Beispiel: `python_intro.py`

```
volume = 57 # "volume" ist Englisch für "Lautstärke"
if volume < 20:
    print("Das ist etwas leise.")
elif 20 <= volume < 40:
    print("Das ist gut für Hintergrund-Musik.")
elif 40 <= volume < 60:
    print("Perfekt, ich kann alle Details hören.")
elif 60 <= volume < 80:
    print("Gut für Partys.")
elif 80 <= volume < 100:
    print("Etwas laut!")
else:
    print("Mir tun die Ohren weh! :(")
```

Python läuft durch jeden Test der Reihe nach und gibt dann aus:

command-line

```
$ python3 python_intro.py
Perfekt, ich kann alle Details hören.
```

## Kommentare

Kommentare sind Zeilen, die mit `#` beginnen. Du kannst nach dem `#` schreiben, was auch immer du willst, und Python wird es ignorieren. Kommentare können deinen Code für andere Leute einfacher zu verstehen machen.

Schauen wir, wie das aussieht:

`python_intro.py`

```
# Ändert die Lautstärke, wenn sie zu leise oder zu laut ist
if volume < 20 or volume > 80:
    volume = 50
    print("So ist's besser!")
```

Du musst nicht für jede Codezeile einen Kommentar schreiben, aber Kommentare sind nützlich um zu erklären, wieso dein Code etwas macht, oder um zusammenzufassen, wenn er etwas Komplexes tut.

## Zusammenfassung

In den letzten paar Übungen hast du gelernt:

- **Vergleiche vorzunehmen** – in Python kannst du Vergleiche mit den folgenden Operatoren `>`, `>=`, `==`, `<=`, `<` sowie `and` und `or` vornehmen

- **Bool'sche Datentypen** zu verwenden – dies sind Objekte, die nur zwei Werte annehmen können: `True` bzw. `False`
- **Dateien zu speichern** – also Programmcode in Dateien abzulegen, so dass du auch umfangreichere Programme schreiben kannst.
- **if ... elif ... else** – Anweisungen, die dir erlauben, bestimmte Programmteile nur auszuführen, wenn bestimmte Bedingungen erfüllt sind.
- **Kommentare** – Zeilen, die Python nicht ausführt und die dir ermöglichen deinen Code zu dokumentieren

Zeit für den letzten Teil dieses Kapitels!

## Deine eigenen Funktionen!

Erinnerst du dich an Funktionen wie `len()`, die du in Python aufrufen kannst? Prima! Du wirst nun lernen, eigene Funktionen zu schreiben!

Eine Funktion ist eine Folge von Anweisungen, die Python ausführen soll. Jede Funktions-Definition beginnt mit dem Schlüsselwort (engl. "Keyword") `def`, bekommt einen Namen und kann Argumente (manchmal auch "Parameter" genannt) haben. Probieren wir's aus! Ersetze den Code in der Datei **python\_intro.py** mit dem folgenden:

`python_intro.py`

```
def hallo():
    print("Halli-hallo!")
    print("Wie geht's?")

hallo()
```

Und schon hast du deine erste Funktion erstellt!

Nun fragst du dich vielleicht, warum wir am Ende der Datei den Namen der Funktion nochmal hingeschrieben haben. Python liest die Datei und führt sie von oben nach unten aus. Um die Funktion also auch zu benutzen, müssen wir sie noch einmal unten hinschreiben.

Schauen wir, was passiert, wenn wir die Datei ausführen:

command-line

```
$ python3 python_intro.py
Halli-hallo!
Wie geht's?
```

Falls das nicht funktioniert hat, keine Panik! Die Ausgabe wird dir dabei helfen, herauszufinden wieso:

- Wenn du einen `NameError` erhältst, hast du dich vermutlich irgendwo im Code vertippt. Prüfe also, ob du bei der Funktionsdefinition `def hallo():` und beim Funktionsaufruf `hallo()` den Funktionsnamen gleich geschrieben hast.
- Wenn du einen `IndentationError` bekommst, prüfe, ob beide `print`-Zeilen die gleichen Whitespaces am Zeilenanfang haben: Python will den ganzen Code in einer Funktion hübsch ausgerichtet haben.
- Wenn du gar keine Ausgabe erhältst, stelle sicher, dass `hallo()` am Datei-Ende *nicht* eingerückt ist. Wenn es eingerückt ist, ist dieser Aufruf selbst Teil der Funktion und sie wird gar nicht ausgeführt.

Als Nächstes bauen wir Funktionen mit sogenannten Argumenten. Wir werden das gerade gebaute Beispiel benutzen – eine Funktion, die die ausführende Person begrüßt – aber diesmal mit Namen:

`python_intro.py`

```
def hallo(name):
```

Wie du siehst, geben wir der Funktion jetzt einen Parameter, den wir `name` nennen:

python\_intro.py

```
def hallo(name):
    if name == 'Ola':
        print('Hallo Ola!')
    elif name == 'Sonja':
        print('Hallo Sonja!')
    else:
        print('Hallo Unbekannte(r)!')

hallo()
```

Denk daran: Die `print`-Funktion ist 4 Leerzeichen innerhalb der `if`-Anweisung eingerückt. Das ist sinnvoll, da die Funktion ausgeführt wird, wenn die Bedingung eintritt. Mal sehen, wie das jetzt funktioniert:

command-line

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

Hoppa, ein Fehler. Zum Glück gibt uns Python eine recht nützliche Fehlermeldung. Diese besagt, dass die Funktion `hallo()` (welche wir definiert haben) ein erforderliches Argument (namens `name`) hat und dass wir vergessen haben, dieses beim Funktionsaufruf mitzugeben. Lass uns das am unteren Ende der Datei schnell beheben:

python\_intro.py

```
hallo("Ola")
```

Und wir führen sie erneut aus:

command-line

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

Und wenn wir den Namen ändern?

python\_intro.py

```
hallo("Sonja")
```

Und ausgeführt:

command-line

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

Nun, was denkst du, wird passieren, wenn du einen anderen Namen dort hinein schreibst? (Weder Ola noch Sonja.) Probier es aus und schau, ob du richtig liegst. Es sollte das Folgende herauskommen:

command-line

```
Hallo Unbekannte(r)!
```

Das ist genial, oder? Auf diese Weise musst du dich nicht jedesmal wiederholen, wenn du den Namen der Person änderst, die die Funktion grüßen soll. Und das ist genau der Grund, warum wir Funktionen brauchen – du willst auf keinem Fall deinen Code wiederholen!

Lass uns noch etwas Schlaues probieren – es gibt schließlich mehr als zwei Namen und für jeden eine eigene Bedingung aufzuschreiben, wäre ziemlich aufwendig, oder? Ersetze also deinen Code in der Datei durch den folgenden:

python\_intro.py

```
def hallo(name):
    print('Hallo ' + name + '!')

hallo("Rachel")
```

Lass uns den Code aufrufen:

command-line

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

Herzlichen Glückwunsch! Du hast gerade gelernt, wie du Funktionen schreibst! :)

## Schleifen

Dies ist bereits der letzte Teil. Das ging doch schnell, oder? :)

Programmierer wiederholen sich nicht gerne. Beim Programmieren geht es darum, Dinge zu automatisieren. Wir wollen also nicht jede Person mit ihrem Namen manuell grüßen müssen, oder? Für so etwas kommen Schleifen gelegen.

Erinnerst du dich noch an Listen? Lass uns eine Liste mit Mädchennamen erstellen:

python\_intro.py

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

Wir wollen alle mit ihrem Namen grüßen. Wir besitzen bereits die `hallo`-Funktion, um dies zu tun, also lass sie uns in einer Schleife verwenden:

python\_intro.py

```
for name in girls:
```

Die `for`-Anweisung verhält sich ähnlich wie die `if`-Anweisung; Code unter beiden muss 4 Leerzeichen eingerückt werden.

Hier ist der vollständige Code für die Datei:

## python\_intro.py

```
def hallo(name):
    print('Hallo ' + name + '!')

girls = ['Rachel', 'Monica', 'Phoebe', 'Ola', 'du']
for name in girls:
    hallo(name)
    print('Nächstes Mädchen')
```

Und wenn wir es ausführen:

### command-line

```
$ python3 python_intro.py
Hallo Rachel!
Nächstes Mädchen
Hallo Monica!
Nächstes Mädchen
Hallo Phoebe!
Nächstes Mädchen
Hallo Ola!
Nächstes Mädchen
Hallo du!
Nächstes Mädchen
```

Wie du sehen kannst, wird alles, was du innerhalb einer `for`-Anweisung eingerückt hast, für jedes Element der Liste `girls` wiederholt.

Du kannst auch `for` auf Ganzzahlen beziehen, wenn du die `range`-Funktion benutzt:

## python\_intro.py

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

Das würde ausgeben:

### command-line

```
1
2
3
4
5
```

`range` ist eine Funktion, die eine Liste von aufeinander folgenden Zahlen erschafft (die Randwerte werden von dir als Argumente bereitgestellt).

Beachte, dass der zweite der Werte, die du als Argumente übergibst, nicht in der Liste enthalten ist, die von Python ausgegeben wird. (Das bedeutet, dass `range(1, 6)` von 1 bis 5 zählt, aber nicht die Zahl 6 enthält). Das liegt daran, dass "range" halboffen ist, was wiederum bedeutet, dass es den ersten Wert enthält, aber nicht den letzten.

## Zusammenfassung

Das ist alles. **Du rockst total!** Das war ein kniffliges Kapitel, du darfst also ruhig stolz auf dich sein. Wir sind definitiv stolz auf dich und darauf, dass du es so weit geschafft hast!

Besuche <https://docs.python.org/3/tutorial/>, wenn du zum offiziellen und vollständigen Python-Tutorial willst. Dort gibt's (bislang jedoch noch nicht auf Deutsch) eine gründlichere und umfassendere Einführung in diese Programmiersprache. :)

Bevor du zum nächsten Kapitel übergehst, mach kurz 'was anderes – streck dich, lauf etwas 'rum, ruh' deine Augen aus. :)



## Django - Was ist das?

Django (/dʒæŋgəʊ/) ist ein freies, quelloffenes Web-Anwendungs-Framework, geschrieben in Python. Ein Web-(Anwendungs-)Framework ist eine Art Baukastensystem, das dir mit vielen vorgefertigten Teilen die Entwicklung von Web-Anwendungen stark erleichtert.

Wenn du eine Website entwickelst, brauchst du immer wieder sehr ähnliche Elemente: Einen Weg, Benutzer zu verwalten (Registrierung, Anmeldung, Abmeldung etc.), einen Administrationsbereich, Formulare, Upload von Dateien usw.

Glücklicherweise wurde schon vor einiger Zeit erkannt, dass Web-Entwickler immer wieder die gleichen Probleme zu lösen haben. Gemeinsam entstanden so verschiedene Frameworks (Django ist so eines), welche die Web-Entwicklung durch vorgefertigte Elemente erleichtern.

Frameworks sind dazu da, damit du das Rad nicht neu erfinden musst. Du kannst dich auf die konkret zu erfüllenden Anforderungen der Webseite kümmern. Die grundlegende Basis der Webseite stellt dir das Framework zur Verfügung.

## Warum brauchst du ein Framework?

Um besser zu verstehen, welche Vorteile dir Django bietet, werfen wir einen Blick auf Server im Allgemeinen. Als Erstes muss der Server wissen, dass er eine Webseite ausliefern soll.

Der Server hat mehrere "Ports". Ein Port ist vergleichbar mit einem Briefkasten, der auf eingehende Briefe ("Anfragen", "requests") überwacht wird. Das macht der Webserver. Der Webserver liest die eingeworfenen Briefe (requests) und beantwortet sie mit der Webseite (response). Um etwas ausliefern zu können, brauchen wir Inhalte. Und Django hilft dir dabei, diese Inhalte zu erstellen.

## Was passiert, wenn jemand eine Webseite beim Server anfordert?

Wenn die Anfrage beim Web-Server ankommt, reicht er diese an Django weiter. Und Django versucht herauszufinden, welche Seite genau angefordert wurde. Django wertet zuerst die Adresse der Webseite aus und versucht herauszufinden, was getan werden soll. Dafür ist der **urlresolver** von Django verantwortlich (Hinweis: URL - Uniform Resource Locator ist ein anderer Name für die Web-Adresse, daher der Name *urlresolver*). Sehr schlau ist er aber nicht. Er hat eine Musterliste und vergleicht diese mit der URL. Der Vergleich der Muster erfolgt von oben nach unten. Wenn ein Muster auf die URL zutrifft, wird der damit verknüpften Funktion (der sogenannten *view*) der Request/die Anfrage übergeben.

Stell dir eine Postbotin mit einem Brief vor. Sie geht die Straße entlang und prüft jede Hausnummer mit der Adresse auf dem Brief. Wenn beide passen, dann steckt sie den Brief in den Briefkasten. So funktioniert der urlresolver!

In der *view* Funktion passieren all die interessanten Dinge: wir können in eine Datenbank gucken und dort nach Informationen suchen. Vielleicht wollte die Benutzerin irgendetwas in den Daten ändern? So, als ob der Brief sagen würde: "Bitte ändere meine Stellenbeschreibung!" Die Funktion *view* kann nun prüfen, ob du dazu berechtigt bist, im positiven Fall die Änderungen durchführen und im Anschluss eine Bestätigungs-Nachricht zurücksenden. Die *view* generiert dann eine Antwort und Django kann diese an den Webbrower der Benutzerin senden.

Die Beschreibung oben ist ein wenig vereinfacht, aber du musst noch nicht all die technischen Details wissen. Eine generelle Vorstellung zu haben, reicht erstmal.

Anstatt zu sehr ins Detail zu gehen, fangen wir lieber an, mit Django etwas zu erschaffen, und du wirst dabei alles Wichtige lernen!



# Django-Installation

**Hinweis:** Falls du dich bereits durch die Installationsschritte gearbeitet hast, gibt es keinen Grund dies erneut zu tun - du kannst direkt zum nächsten Kapitel springen!

Ein Teil dieses und des nächsten Kapitels basiert auf den Tutorials von Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Teile dieses und des nächsten Kapitels basieren auf dem [django-marcador Tutorial](#) lizenziert unter der Creative Commons Attribution-ShareAlike 4.0 International License. Für das "django-marcador Tutorial" liegt das Urheberrecht bei Markus Zapke-Gründemann et al.

## Virtuelle Umgebung

Bevor wir mit der Installation von Django beginnen, lernen wir ein sehr hilfreiches Tool kennen, das uns hilft, unsere Arbeitsumgebung zum Coden sauber zu halten. Es ist möglich, diesen Schritt zu überspringen, aber wir legen ihn dir dennoch besonders ans Herz. Mit dem bestmöglichen Setup zu starten, wird dir in der Zukunft eine Menge Frust ersparen!

Wir erzeugen eine virtuelle Arbeitsumgebung - ein **virtual environment** oder auch `virtualenv`. Das isoliert die Python-/Django-Setups verschiedener Projekte voneinander. Das bedeutet, dass deine Änderungen an einem Website-Projekt keine anderen Projekte beeinträchtigen, an welchen du sonst noch entwickelst. Klingt nützlich, oder?

Du musst nur das Verzeichnis festlegen, in dem du das `virtualenv` erstellen willst; zum Beispiel dein Home-Verzeichnis.

In diesem Tutorial erstellen wir darin ein neues Verzeichnis `djangogirls`:

command-line

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

Wir erstellen eine virtuelle Umgebung namens `myvenv`. Das Kommando dazu lautet dann:

command-line

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

### Virtual environment: Linux and OS X

Auf Linux oder OS X kann ein `virtualenv` durch das Ausführen von `python3 -m venv myvenv` erzeugt werden. Das wird so aussehen:

command-line

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

`myvenv` ist der Name deiner neuen virtuellen Arbeitsumgebung, deines neuen `virtualenv`. Du kannst auch irgend einen anderen Namen wählen, aber bleibe bei Kleinbuchstaben und verwende keine Leerzeichen. Eine Gute Idee ist, den Namen kurz zu halten. Du wirst ihn oft benutzen bzw. eingeben müssen!

**Hinweis:** Eine Anleitung für Windows findest du auf [https://tutorial.djangogirls.org/de/django\\_installation/](https://tutorial.djangogirls.org/de/django_installation/)

## Mit der virtuellen Umgebung arbeiten

Die obigen Kommandos erstellen ein Verzeichnis `myvenv` (bzw. den von Dir vergebenen Namen). Es enthält unsere virtuelle Arbeitsumgebung (im Wesentlichen ein paar Verzeichnisse und Dateien).

### Working with virtualenv: Linux and OS X

Starte deine virtuelle Umgebung, indem du eingibst:

command-line

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

Der Name `myvenv` muss mit dem von Dir gewählten Namen des `virtualenv` übereinstimmen!

**Anmerkung:** Manchmal ist das Kommando `source` nicht verfügbar. In diesen Fällen geht es auch so:

command-line

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

Du erkennst, dass dein `virtualenv` gestartet ist, wenn du vor der Eingabeaufforderung eine Klammer mit dem Namen deiner Umgebung siehst, `(myvenv)`.

In deiner neuen virtuellen Umgebung wird automatisch die richtige Version von `python` verwendet. Du kannst also `python` statt `python3` eingeben.

Ok, jetzt ist die erforderliche Umgebung startklar und wir können endlich Django installieren!

## Django-Installation

Da du nun dein `virtualenv` gestartet hast, kannst du Django installieren.

Bevor wir damit loslegen, sollten wir jedoch sicherstellen, dass wir die neueste Version von `pip` haben, eine Software, mit Hilfe derer wir Django installieren werden:

command-line

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

## Pakete mittels requirements-Datei installieren

Eine requirements-Datei enthält eine Liste von Abhängigkeiten, die von `pip install` installiert werden sollen:

Erstelle mit dem zuvor installierten Code-Editor eine Datei namens `requirements.txt` im Verzeichnis `djangogirls/`. Das machst du, indem du eine neue Datei in deinem Code-Editor öffnest und als `requirements.txt` im Ordner `djangogirls/` abspeicherst. Dein Ordner sieht jetzt so aus:

```
djangogirls
└──requirements.txt
```

Schreibe in die Datei `djangogirls/requirements.txt` folgenden Text:

`djangogirls/requirements.txt`

```
Django==2.0.6
```

Führe nun `pip install -r requirements.txt` aus, um Django zu installieren.

command-line

```
(myvenv) ~$ pip install -r requirements.txt
Collecting Django==2.0.6 (from -r requirements.txt (line 1))
  Downloading Django-2.0.6-py3-none-any.whl (7.1MB)
Installing collected packages: Django
Successfully installed Django-2.0.6
```

### Installing Django: Linux

Falls der pip-Aufruf auf Ubuntu 12.04 zu einer Fehlermeldung führt, rufe `python -m pip install -U --force-reinstall pip` auf, um die Installation von pip im virtualenv zu reparieren.

Das war's! Du bist nun (endlich) bereit, deine erste Django-Anwendung zu starten!

# Dein erstes Django-Projekt!

Wir werden einen kleinen Blog erstellen!

Der erste Schritt ist, ein neues Django-Projekt zu starten. Im Grunde bedeutet das, dass wir einige Skripte ausführen werden, die Django zur Verfügung stellt, um ein Skelett eines Django-Projekts für uns zu erzeugen. Das Projekt beinhaltet einen Haufen von Verzeichnissen und Dateien, die wir später verwenden werden.

Die Namen einiger Dateien und Verzeichnisse sind sehr wichtig für Django. Die Dateien, die erstellt werden, solltest du nicht umbenennen. Sie an eine andere Stelle zu verschieben, ist auch keine gute Idee. Django muss zwingend eine gewisse Struktur erhalten, um wichtige Dinge wiederzufinden.

Denk daran, alles in der "Virtualenv"-Umgebung auszuführen. Wenn du kein Präfix `(myvenv)` in deiner Konsole siehst, musst du deine Virtualenv-Umgebung aktivieren. Wie das gemacht wird, erklären wir im Kapitel **Django-Installation**, im Abschnitt **Arbeiten mit Virtualenv**. Zur Erinnerung: Gib dazu auf Windows `myvenv\Scripts\activate` ein, bzw. auf OS X oder Linux `source myvenv/bin/activate`.

## Create project: OS X or Linux

In deiner OS X- oder Linux-Konsole solltest du den folgenden Befehl ausführen; **vergiss den Punkt ( . ) am Ende nicht!**

command-line

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

Der Punkt `.` ist sehr wichtig, weil er dem Skript mitteilt, dass Django in deinem aktuellen Verzeichnis installiert werden soll. (Der Punkt `.` ist eine Schnellreferenz dafür.)

**Hinweis:** Wenn du das oben angegebene Kommando eingibst, denk daran, nur das einzutippen, was mit `django-admin` anfängt. Der `(myvenv) ~/djangogirls$`-Teil hier ist nur ein Beispiel für die Eingabeaufforderung (den "Prompt") auf der Kommandozeile.

`django-admin.py` ist ein Skript, welches Verzeichnisse und Dateien für dich erstellt. Du solltest jetzt eine Verzeichnisstruktur haben, die folgendermaßen aussieht:

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

**Hinweis:** In deiner Verzeichnisstruktur wirst du auch den `venv`-Ordner sehen, den wir vorhin erzeugt haben.

`manage.py` ist ein Script, das das Management deines Projektes unterstützt. Mit dem Script bist du unter anderem in der Lage, den Webserver auf deinem Rechner zu starten, ohne etwas Weiteres installieren zu müssen.

Die Datei `settings.py` beinhaltet die Konfiguration deiner Website.

Erinnerst du dich, als wir über den Postboten gesprochen haben, der überlegt, wohin er den Brief liefern soll? Die `urls.py` Datei beinhaltet eine Liste von Patterns (Mustern), die vom `urlresolver` benutzt werden.

Lass uns kurz die anderen Dateien vergessen - wir werden sie nicht verändern. Denk aber dran, sie nicht versehentlich zu löschen!

## Einstellungen anpassen

Wir machen nun ein paar Änderungen in `mysite/settings.py`. Öffne die Datei mit dem Code-Editor, den du schon installiert hast.

**Hinweis:** `settings.py` ist eine Datei wie jede andere. Du kannst sie aus deinem Code-Editor heraus öffnen, in dem du im "Datei"-Menü die "Öffnen"-Aktion wählst. So solltest du das normale Fenster zur Dateiauswahl bekommen, in dem du zur `settings.py`-Datei navigieren und sie auswählen kannst. Stattdessen kannst du die Datei aber auch öffnen, in dem du im Dateimanager zum "djangogirls"-Ordner navigierst und auf die Datei rechtsklickst. Wähle dann deinen Code-Editor aus der Liste der "Öffnen mit"-Programme. Das ist wichtig, da du andere Programme installiert haben könntest, die diese Datei zwar öffnen können, aber die dich die Datei nicht editieren lassen würden.

Es wäre schön, wenn die richtige Zeit auf deiner Webseite eingestellt ist. Gehe zur [Zeitzonen-Liste auf Wikipedia](#) und kopiere die für dich geltende Zeitzone (Spalte "TZ"), z.B. `Europe/Berlin`.

Suche in `settings.py` die Zeile, die `TIME_ZONE` enthält und ändere sie ab, um deine eigene Zeitzone auszuwählen. Zum Beispiel:

`mysite/settings.py`

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

Ein Sprachkennung besteht aus einem Kürzel für die Sprache, z.B. `en` für Englisch oder `de` für Deutsch, und einem Länder-Kürzel z.B. `de` für Deutschland oder `ch` für die Schweiz. Falls Englisch nicht deine Muttersprache ist, kannst du damit die Standard-Knöpfe und -Meldungen von Django auf deine Sprache wechseln. Der "Cancel"-Knopf würde dann in diese Sprache übersetzt (und z.B. bei Deutsch mit "Abbrechen" beschriftet). [Django enthält viele fix-fertige Übersetzungen](#).

Wenn du eine andere Sprache als Englisch willst, ändere die Sprachkennung, indem du die folgende Zeile änderst:

`mysite/settings.py`

```
LANGUAGE_CODE = 'de-ch'
```

Ausserdem müssen wir einen Pfad für statische Dateien festlegen. (Über statische Dateien und CSS lernst du später in diesem Tutorial etwas.) Geh hinunter zum Ende der Datei und füge direkt unter dem `STATIC_URL`-Eintrag einen neuen Eintrag namens `STATIC_ROOT` ein:

`mysite/settings.py`

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

Wenn `DEBUG` auf `True` gesetzt ist und `ALLOWED_HOSTS` leer, dann wird der "Host" gegen `['localhost', '127.0.0.1', '[::1]']` validiert. Unser Hostname auf PythonAnywhere, wo wir unsere Anwendung deployen werden, würde da nicht passen. Deswegen ändern wir folgende Einstellung:

`mysite/settings.py`

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

## Eine Datenbank erstellen

Es gibt viele verschiedene Datenbank Programme, welche die Daten unserer Website verwalten können. Wir werden die Standard-Datenbanksoftware nehmen, `sqlite3`.

Das sollte schon in der `mysite/settings.py`-Datei eingestellt sein:

`mysite/settings.py`

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

Um eine Datenbank für unseren Blog zu erstellen, müssen wir folgenden Befehl in der Konsole ausführen (Dazu müssen wir in dem `djangogirls`-Verzeichnis sein, in dem sich auch die `manage.py`-Datei befindet). Wenn alles glatt läuft, sollte das so aussehen:

command-line

```
(myvenv) ~/djangogirls$ python manage.py migrate
Operations to perform:
  Apply all migrations: auth, admin, contenttypes, sessions
Running migrations:
  Rendering model states... DONE
    Applying contenttypes.0001_initial... OK
    Applying auth.0001_initial... OK
    Applying admin.0001_initial... OK
    Applying admin.0002_logentry_remove_auto_add... OK
    Applying contenttypes.0002_remove_content_type_name... OK
    Applying auth.0002_alter_permission_name_max_length... OK
    Applying auth.0003_alter_user_email_max_length... OK
    Applying auth.0004_alter_user_username_opts... OK
    Applying auth.0005_alter_user_last_login_null... OK
    Applying auth.0006_require_contenttypes_0002... OK
    Applying auth.0007_alter_validators_add_error_messages... OK
    Applying auth.0008_alter_user_username_max_length... OK
    Applying auth.0009_alter_user_last_name_max_length... OK
    Applying sessions.0001_initial... OK
```

Und wir sind fertig! Zeit, unseren Webserver zu starten, um zu sehen, ob unsere Website funktioniert!

## Den Webserver starten

Kontrolliere, dass du in dem Verzeichnis bist, in dem die `manage.py`-Datei liegt (das `djangogirls`-Verzeichnis). Wir starten den Webserver, indem wir in der Konsole `python manage.py runserver` ausführen:

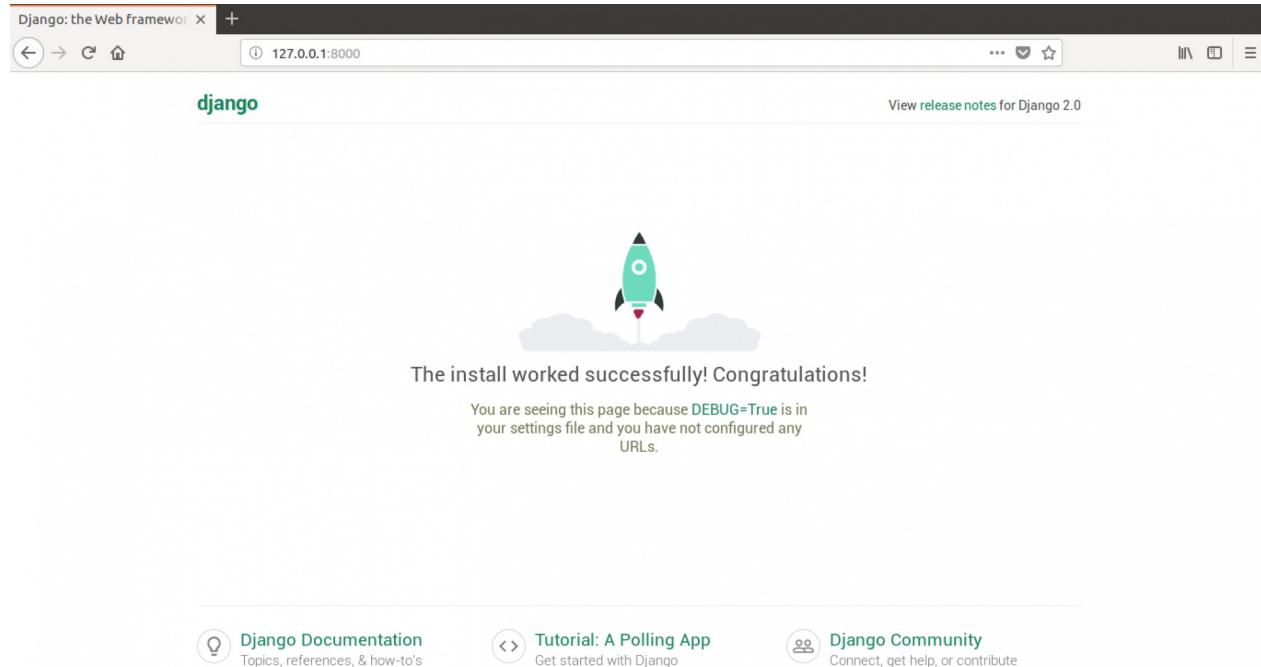
command-line

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

Jetzt wollen wir schauen, ob unsere Website funktioniert: Öffne deinen Browser (Firefox, Chrome, Safari, Edge oder was du sonst nutzt) und gib diese Adresse ein:

browser

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



Beachte, dass ein Terminalfenster immer nur eine Sache zur selben Zeit erledigen kann, und in dem Terminalfenster, was du vorhin geöffnet hast, läuft gerade der Webserver. Und solange der Webserver läuft und auf einkommende Anfragen wartet, akzeptiert das Terminal zwar Texteingaben, aber es wird keine neuen Befehle ausführen.

Wie Webserver funktionieren, haben wir im Kapitel "[Wie das Internet funktioniert](#)" angesehen.

Um weitere Kommandos einzugeben, während der Webserver läuft, öffne ein neues Kommandozeilen-Fenster und aktiviere dort deine Virtualenv-Umgebung. Siehe [Einführung in die Kommandozeile](#), um nachzulesen, wie du ein zweites Kommandozeilen-Fenster öffnen kannst. Um den Webserver zu stoppen, wechsel zurück in das Fenster, in dem er läuft, und drücke STRG+C - Steuerung und C gleichzeitig. (In Windows kann es sein, dass du STRG und "Pause"-Taste drücken musst).

Bereit für den nächsten Schritt? Es wird Zeit, ein paar Inhalte hinzuzufügen!

# Django-Models

Wir erstellen jetzt etwas, damit wir alle Posts von unserem Blog speichern können. Aber um das zu tun, müssen wir zunächst über Objekte sprechen.

## Objekte

Eine Herangehensweise an das Programmieren ist das so genannte objektorientierte Programmieren. Die Idee dahinter ist, dass wir Dinge und ihre Interaktionen untereinander modellieren können und nicht alles als langweilige Kette von Programmbefehlen hintereinander aufschreiben müssen.

Was ist denn nun ein Objekt? Ein Objekt ist eine Sammlung von Eigenschaften und Aktionsmöglichkeiten, das anhand einer Vorlage (Klasse) erstellt wird. Das klingt erst einmal komisch, aber hier haben wir gleich ein Beispiel.

Wenn wir zum Beispiel eine Katze modellieren wollen, erschaffen wir eine Objektvorlage `Katze`, eine Art Blaupause oder Schema, nach welcher zukünftig jedes spezifische Katzenobjekt erstellt werden kann. Die Vorlage beschreibt typische Eigenschaften von Katzen, z.B. `farbe`, `alter`, `stimmung` (also gut, schlecht oder müde ;)), `besitzerin` (die ein Person-Objekt ist oder – im Falle einer Streunerkatze – leer bleibt).

Jedes Objekt einer `Katze` soll natürlich auch einige Aktionsmöglichkeiten besitzen: `schnurren`, `kratzen` oder `füttern` (hier bekäme die Katze ein bisschen `katzenfutter`, welches wieder durch ein eigenes Objekt mit Eigenschaften wie `Geschmack` repräsentiert sein könnte).

```
Katze
-----
farbe
alter
stimmung
besitzerin
schnurren()
kratzen()
fuettern(katzen_futter)

Katzenfutter
-----
geschmack
```

Der Gedanke dahinter ist also, echte Dinge mit Hilfe von Eigenschaften (genannt `Objekteigenschaften`) und Aktionsmöglichkeiten (genannt `Methoden`) im Programmcode zu beschreiben.

Wie also modellieren wir Blogposts? Schließlich wollen wir ja einen Blog bauen, nicht wahr?

Wir müssen folgende Fragen beantworten: Was ist ein Blogpost? Welche Eigenschaften sollte er haben?

Nun, zum einen braucht unser Blogpost Text mit einem Inhalt und einen Titel, oder? Außerdem wäre es schön zu wissen, wer ihn geschrieben hat – wir brauchen also noch einen Autor. Schließlich wollen wir wissen, wann der Post geschrieben und veröffentlicht wurde.

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

Was für Dinge könnte man mit einem Blogpost machen? Es wäre schön, wenn wir eine `Methode` hätten, die den Post veröffentlicht, nicht wahr?

Wir brauchen also eine `veröffentlichen`-Methode.

Da wir jetzt wissen, was wir erreichen wollen, können wir nun damit anfangen, es in Django zu formulieren!

## Ein Django-Model erstellen

Da wir jetzt in etwa wissen, was ein Objekt ist, wollen wir ein Django-Model, eine Vorlage, für unsere Blogposts anlegen, nach welcher wir zukünftig unsere Blogpostobjekte erstellen können.

Ein "Modell" ist in Django ein Objekt einer speziellen Sorte – eines das in der `Datenbank` gespeichert wird. Eine Datenbank ist eine Sammlung von Daten. Dies ist ein Ort, an dem du Informationen zu Benutzern, deinen Blogposts usw. speichern wirst. Wir benutzen dafür eine SQLite-Datenbank. Das ist die Voreinstellung in Django – für uns wird das erst einmal ausreichen.

Du kannst dir ein Model wie eine Tabelle mit Spalten ("Feldern", englisch "fields") und Zeilen (Datensätzen) vorstellen.

## Eine Applikation für unseren Blog

Um unsere Webseite aufgeräumt zu halten, werden wir eine eigene Anwendung für unser Projekt erstellen, wir nennen das eine Applikation. Wir wollen uns gleich daran gewöhnen, alles ordentlich und sortiert zu halten. Um eine Applikation zu erstellen, müssen wir das folgende Kommando in der Konsole ausführen (wieder in dem `djangogirls`-Verzeichnis, in dem die `manage.py`-Datei liegt):

Mac OS X und Linux:

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

Windows:

```
(myvenv) C:\Users\Name\djangogirls> python manage.py startapp blog
```

Wie du sehen kannst, wurde ein neues `blog`-Verzeichnis erstellt, welches schon einige Dateien enthält. Das Verzeichnis und die Dateien unseres Projektes sollten jetzt so aussehen:

```
djangogirls
└── blog
    ├── __init__.py
    ├── admin.py
    ├── apps.py
    ├── migrations
    │   └── __init__.py
    ├── models.py
    ├── tests.py
    └── views.py
├── db.sqlite3
├── manage.py
└── mysite
    ├── __init__.py
    ├── settings.py
    ├── urls.py
    └── wsgi.py
└── requirements.txt
```

Nach dem Erstellen der Applikation müssen wir Django noch sagen, dass diese auch genutzt werden soll. Das tun wir in der Datei `mysite/settings.py` -- öffne diese in deinem Code-Editor. Wir suchen den Eintrag `INSTALLED_APPS` und fügen darin die Zeile `'blog'`, direkt über der schließenden Klammer `]` ein. Danach sollte es also so aussehen:

`mysite/settings.py`

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

## Das Blogpost-Model

Alle `Models` unserer Applikation werden in der `blog/models.py`-Datei definiert. Dies ist also der richtige Platz für unser Blogpost-Model.

Öffnen wir also `blog/models.py` im Code-Editor, löschen alles darin und schreiben Code wie diesen:

`blog/models.py`

```

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

class Post(models.Model):
    author = models.ForeignKey(settings.AUTH_USER_MODEL, 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

```

Kontrolliere nochmal, dass du zwei Unterstriche ( `__` ) vor und hinter dem `str` gesetzt hast. Diese Konvention wird häufig in Python benutzt und manchmal nennen wir es "dunder" (kurz für "double-underscore").

Es sieht kompliziert aus, oder? Aber keine Sorge, wir werden erklären, was diese Zeilen bedeuten!

Die Zeilen, die mit `from` oder `import` beginnen, sind Anweisungen, um Sachen aus anderen Dateien mitzunutzen. Anstatt häufig genutzten Code in jede Datei einzeln zu kopieren, binden wir ihn ein mit: `from... import ...`.

`class Post(models.Model):` – Diese Zeile definiert unser Model (eine Objekt-Vorlage).

- `class` ist ein spezielles Schlüsselwort, das angibt, dass wir hier eine Klasse, eine Vorlage für zukünftige Objekte, definieren wollen.
- `Post` ist der Name unseres Models. Wir können ihm auch einen anderen Namen geben (aber wir müssen Sonderzeichen und Leerzeichen vermeiden). Beginne einen Klassennamen immer mit einem Großbuchstaben.
- `models.Model` gibt an, dass das Post-Model ein Django-Model ist, so weiß Django, dass es in der Datenbank gespeichert werden soll.

Jetzt definieren wir die Eigenschaften, über welche wir gesprochen haben: `title`, `text`, `created_date`, `published_date` und `author`. Um dies zu tun, müssen wir den Typ jedes Felds definieren. (Ist es Text? Eine Zahl? Ein Datum? Eine Beziehung zu einem anderen Objekt, z.B. einem Benutzer?)

- `models.CharField` – so definierst du ein Textfeld mit einer limitierten Anzahl von Zeichen.
- `models.TextField` – so definierst du ein langes Textfeld ohne Größenbeschränkung. Klingt doch perfekt für unsere Blogpostinhalte, oder?
- `models.DateTimeField` – ein Feld für einen Zeitpunkt (ein Datum und eine Uhrzeit).
- `models.ForeignKey` – definiert eine Verknüpfung/Beziehung zu einem anderen Model.

Wir werden nicht den gesamten Code hier erklären, da das zu lange dauern würde. Du solltest einen Blick in die offizielle Django-Dokumentation werfen, wenn du mehr über Modelfelder und darüber wissen möchtest, wie man auch andere Dinge als die oben beschriebenen definiert (<https://docs.djangoproject.com/en/2.0/ref/models/fields/#field-types>).

Was ist mit `def publish(self):`? Das ist genau die `publish`-Methode zum Veröffentlichen unserer Blogposts, über die wir vorher bereits sprachen. `def` zeigt an, dass es sich nachfolgend um eine Funktion/Methode handelt, und `publish` ist der Name der Methode. Du kannst den Namen der Methode auch ändern, wenn du möchtest. Die Benennungsregel ist, dass wir Kleinbuchstaben verwenden, und anstatt Leerzeichen (die in Funktionsnamen nicht vorkommend dürfen) Unterstriche. Eine Methode, die einen Durchschnittspreis berechnet, könnte zum Beispiel `calculate_average_price` genannt werden.

Oft geben Methoden einen Wert zurück (englisch: `return`). Ein Beispiel dafür ist die Methode `__str__`. In diesem Szenario, wenn wir `__str__()` aufrufen, bekommen wir einen Text (**string**) mit einem Blogpost-Titel zurück.

Beachte, dass sowohl `def publish(self):` als auch `def __str__(self):` in unserer Klasse eingerückt sind. Mit der Einrückung sagen wir Python, dass diese Methoden Teil der Klasse sind. Ohne die Einrückung wären es für Python Funktionen ausserhalb der Klasse, was zu anderem Verhalten führen würde.

Wenn dir über Methoden noch etwas nicht klar ist, dann zögere nicht, deinen Coach zu fragen! Wir wissen, dass es kompliziert ist, vor allem, wenn du gleichzeitig lernst, was Objekte und Funktionen sind. Aber hoffentlich sieht es für dich jetzt etwas weniger nach Magie aus!

## Tabellen für Models in deiner Datenbank erstellen

Als letzten Schritt wollen wir unser neues Model der Datenbank hinzufügen. Dazu müssen wir Django erst 'mal mitteilen, dass wir einige Änderungen an unserem Model vorgenommen haben. (Wir haben es eben erst erstellt!) Schreibe `python manage.py makemigrations blog` in dein Kommandozeilen-Fenster. Das sieht dann so aus:

command-line

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

    - Create model Post
```

**Hinweis:** Denke daran, die Dateien nach dem Editieren zu speichern. Ansonsten führt dein Computer die vorherige Version aus, was zu unerwarteten Fehlermeldungen führen kann.

Django hat eine Migrationsdatei für uns vorbereitet, die wir nun auf unsere Datenbank anwenden müssen. Schreibe `python manage.py migrate blog`. Die Ausgabe sollte so aussehen:

command-line

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

Hurra! Unser Post-Model ist ab sofort in unserer Datenbank gespeichert! Es wäre doch schön, zu wissen, wie es aussieht, oder? Springe zum nächsten Kapitel, um es dir anzusehen!

# Django-Administration

Wir benutzen den Django-Admin, um die soeben modellierten Posts hinzuzufügen, zu ändern oder zu löschen.

Öffne die Datei `blog/admin.py` im Code-Editor und ersetze den Inhalt wie folgt:

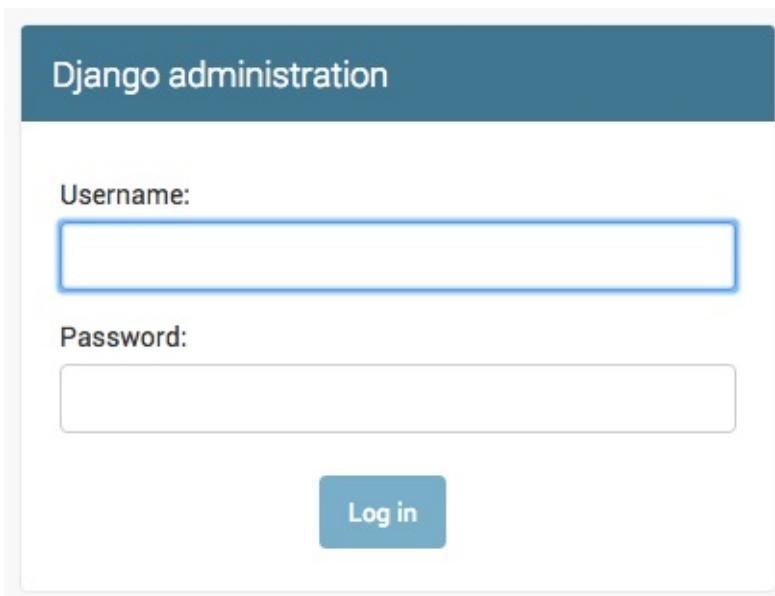
`blog/admin.py`

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

admin.site.register(Post)
```

Wie du siehst, importieren wir hier das Model "Post", das wir im vorherigen Kapitel erstellt haben. Damit unser Model auf der Admin-Seite sichtbar wird, müssen wir es mit `admin.site.register(Post)` registrieren.

Okay, wir sehen uns nun unser Post-Model an. Denk daran, `python manage.py runserver` in die Konsole einzugeben, um den Webserver zu starten. Öffne deinen Browser und gib die Adresse <http://127.0.0.1:8000/admin/> ein. Du siehst eine Anmeldeseite:



Um dich einloggen zu können, musst du zunächst einen *superuser* erstellen - einen User, der alles auf der Website steuern darf. Geh zurück zur Kommandozeile, tippe `python manage.py createsuperuser` und drücke Enter.

Denke daran, damit du neue Kommandos eingeben kannst während der Webserver läuft, musst du ein neues Terminal öffnen und deine virtualenv aktivieren. Wie man neue Kommandos eingeben kann, haben wir im Kapitel **Dein erstes Django-Projekt!** unter **Den Webserver starten** behandelt.

Mac OS X oder Linux:

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

Windows:

```
(myvenv) C:\Users\Name\djangogirls> python manage.py createsuperuser
```

Wenn du dazu aufgefordert wirst, gib einen Benutzernamen (Kleinbuchstaben, keine Leerzeichen), eine Mailadresse und ein Passwort ein. **Mach dir keine Gedanken, wenn du das Passwort bei der Eingabe nicht sehen kannst - so soll es sein.** Tippe weiter und drücke `Enter`, um weiterzumachen. Du solltest nun Folgendes sehen (wobei Benutzername und Email deine eigenen sein sollten):

```
Username: ola
Email address: ola@example.com
Password:
Password (again):
Superuser created successfully.
```

Geh nochmal in deinen Browser und log dich mit den Daten des Superusers ein, den du gerade erstellt hast. Du solltest nun das Django-Admin-Dashboard sehen.

## Django administration

### Site administration

#### AUTHENTICATION AND AUTHORIZATION

<a href="#">Groups</a>	<a href="#"> Add</a>	<a href="#"> Change</a>
------------------------	----------------------	-------------------------

<a href="#">Users</a>	<a href="#"> Add</a>	<a href="#"> Change</a>
-----------------------	----------------------	-------------------------

#### BLOG

<a href="#">Posts</a>	<a href="#"> Add</a>	<a href="#"> Change</a>
-----------------------	----------------------	-------------------------

Gehe auf Posts und experimentiere ein bisschen damit. Füge fünf oder sechs Blogeinträge hinzu. Mach dir keine Sorgen wegen des Inhalts -- der ist nur auf deinem lokalen Computer sichtbar. Um Zeit zu sparen kannst du etwas Text aus diesem Tutorial kopieren und einfügen. :-)

Achte darauf, dass bei wenigstens zwei oder drei Posts (aber nicht bei allen) das Veröffentlichungsdatum (publish date) eingetragen ist. Das werden wir später noch brauchen.

Django administration

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

Home > Blog > Posts > Add post

Add post

Author: kojo

Title:

Text:

Created date: Date: 2015-12-25 Today |  Time: 20:50:01 Now |

Published date: Date:  Today |  Time:  Now |

Mehr zum Django-Admin-Dashboard kannst du in der Django-Dokumentation erfahren:

<https://docs.djangoproject.com/en/2.0/ref/contrib/admin/>.

Jetzt ist wahrscheinlich ein guter Moment, um dir einen Kaffee (oder Tee) zu gönnen und neue Kraft zu tanken. Du hast dein erstes Django-Model erstellt - du hast dir eine kleine Pause verdient!

# Veröffentlichen!

**Hinweis:** Durch das folgende Kapitel muss man sich manchmal durchbeißen. Bleib dran und gib nicht auf; die Website zu veröffentlichen, ist ein sehr wichtiger Schritt. Dieses Kapitel ist in der Mitte des Tutorials platziert, damit dir dein Mentor mit dem etwas anspruchsvolleren Vorgang der Veröffentlichung deiner Website helfen kann. Den Rest des Tutorials kannst du dann auch alleine beenden, sollte die Zeit nicht ausreichen.

Bis jetzt war deine Webseite nur auf deinem Computer verfügbar. Jetzt wirst du lernen wie du sie 'deployst'! Deployen bedeutet, dass du deine Anwendung im Internet veröffentlicht, so dass endlich jeder darauf zugreifen kann. :)

Wie du schon gelernt hast, muss eine Webseite auf einem Server liegen. Es gibt eine Vielzahl von Hosting (Server)-Anbietern im Internet, wir werden [PythonAnywhere](#) verwenden. PythonAnywhere ist kostenlos für kleine Anwendungen, die nicht von vielen Besuchern aufgerufen werden. Also erstmal genau das Richtige für dich.

Als weiteren externen Dienst werden wir [GitHub](#) nutzen, einen "Code Hosting"-Dienst. Es gibt noch andere solcher Dienste, aber die meisten Programmierer haben heute ein Konto bei GitHub, und du auch gleich!

Diese drei Orte werden für dich wichtig sein. Die Entwicklung und das Testen wirst du auf deinem lokalen Rechner durchführen. Wenn du mit deinen Änderungen zufrieden bist, wirst du eine Kopie deines Programms auf GitHub veröffentlichen. Deine Webseite wird auf PythonAnywhere gehostet werden. Ändern kannst du sie, indem du eine neue Version deines Codes von GitHub herunter lädst.

## Git

Git ist ein "Versionsverwaltungssystem" für Dateien und Code, das von vielen Programmierern benutzt wird. Diese Software kann Änderungen an Dateien über die Zeit verfolgen, so dass du bestimmte Versionen im Nachhinein wieder aufrufen kannst. Sie hat Ähnlichkeit mit der Funktion "Änderungen nachverfolgen" in Textverarbeitungsprogrammen (z. B. Microsoft Word oder LibreOffice Writer), ist jedoch weitaus leistungsfähiger.

## Unser Git-Repository

Git verwaltet die Veränderungen an einer Sammlung von Dateien in einem sogenannten Repository (oder kurz "Repo"). Legen wir eines für unser Projekt an. Öffne deine Konsole und gib folgende Kommandos im `djangogirls`-Verzeichnis ein:

**Hinweis:** Überprüfe dein aktuelles Arbeitsverzeichnis mit dem Befehl `pwd` (OSX/Linux) oder `cd` (Windows) bevor du das Repository initialisierst. Du musst dich im `djangogirls`-Verzeichnis befinden, bevor du fortfährst.

command-line

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

Die Initialisierung des Git-Repositorys müssen wir für jedes Projekt nur einmal machen (danach musst Du Benutzernamen und Mail-Adresse nie wieder eingeben).

Git wird die Änderungen an all den Dateien und Ordnern in diesem Verzeichnis aufzeichnen. Wir wollen aber, dass einige Dateien ignoriert werden. Dazu legen wir eine Datei `.gitignore` im Hauptordner (`djangogirls`) des Repos an. Öffne deinen Editor und erstelle eine neue Datei mit dem folgenden Inhalt:

`.gitignore`

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

Speichere die Datei mit dem Namen `.gitignore` im "djangogirls"-Root-Verzeichnis.

**Hinweis:** Der Punkt vor dem Dateinamen ist wichtig! Wenn du Schwierigkeiten beim Erstellen hast (z.B. lassen Macs im Finder keine Dateien mit Punkt am Anfang erzeugen, Punkt-Dateien sind auf Linux und OS X "versteckte Dateien"), dann verwende die "Speichern unter"-Funktion im Editor, das sollte immer funktionieren. Wichtig ist, dass du den Dateinamen nicht mit `.txt`, `.py` oder einer anderen Dateinamen-Erweiterung ergänzt -- die Datei wird von Git nur erkannt, wenn ihr Name exakt nur `.gitignore` ist.

**Hinweis:** Eine der Dateien, die du in deiner `.gitignore`-Datei definiert hast, ist `db.sqlite3`. Diese Datei ist deine lokale Datenbank, in welcher alle deine Benutzer und Posts gespeichert werden. Wir werden die gängige Web-Entwicklungs-Praxis befolgen, was heißt, dass wir separate Datenbanken für unsere lokale Test-Website und unsere öffentliche Website auf PythonAnywhere verwenden werden. Die Datenbank für letztere könnte SQLite sein, wie auf deiner Entwicklungsmaschine, aber normalerweise wirst du eine sogenannte MySQL-Datenbank nutzen, welche mit viel mehr Besuchern umgehen kann als SQLite. So oder so, dadurch, dass du deine SQLite-Datenbank für die GitHub-Kopie nicht verwendest, werden alle deine bisherigen Posts der Superuser nur lokal zur Verfügung stehen und du musst in der produktiven Umgebung neue hinzufügen. Betrachte deine lokale Datenbank als tollen Spielplatz, auf welchem du verschiedene Dinge ausprobieren kannst, ohne Angst zu haben, dass du deine wirklichen Post auf deinem Blog löschenst.

Es ist hilfreich den Befehl `git status` vor `git add` auszuführen oder immer dann, wenn du dir unsicher bist, was geändert wurde. Das schützt vor manchen Überraschungen, wie z. B. das falsche Hinzufügen oder Übertragen von Dateien. Das `git status`-Kommando gibt Informationen über unbeobachtete/veränderte/hinzugefügte Dateien, den Status der Verzweigung und einiges mehr wieder. Deine Ausgabe sollte dem hier ähneln:

command-line

```
$ 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/  
requirements.txt  
  
nothing added to commit but untracked files present (use "git add" to track)
```

Nun speichern wir unsere Änderungen durch folgende Eingabe in der Konsole:

command-line

```
$ 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
```

## Den Code auf GitHub veröffentlichen

Gehe auf [GitHub.com](https://GitHub.com) eröffne ein neues, kostenloses Nutzerkonto. (Falls Du es bereits während der Workshop-Vorbereitung eingerichtet hast, ist das großartig!) Stelle sicher, dass Du Dein Passwort nicht vergisst (füge es zu Deinem Passwort-Manager hinzu, falls Du einen solchen verwendest).

Erstelle dann ein neues Repository und gib ihm den Namen "my-first-blog". Lass das Kontrollkästchen "initialise with a README" deaktiviert und die Einstellung der Option .gitignore leer (das haben wir schon von Hand gemacht) und lass die Lizenz auf "None".

Owner: hwpg / 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

**Achtung:** Der Name `my-first-blog` ist wichtig -- du kannst auch einen anderen wählen, aber er wird im Folgenden noch sehr oft vorkommen und du wirst immer daran denken müssen, ihn in den Anweisungen entsprechend anzupassen. Es ist wahrscheinlich einfacher, bei `my-first-blog` zu bleiben.

In der nächsten Ansicht wirst du die clone-URL deines Repositorys sehen, die du in manchen der folgenden Kommandozeilenbefehlen verwenden wirst:

The screenshot shows a GitHub repository page for 'hjwp / my-first-blog'. A red arrow points to the 'HTTPS' button, which is highlighted. Below it, there are instructions for quick setup and command-line cloning.

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

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

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

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

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

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

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

The right sidebar shows repository statistics: 0 issues, 0 pull requests, 0 wiki pages, 0 pulse, 0 graphs, and settings.

Nun müssen wir das Git-Repository auf deinem Computer mit dem auf GitHub verbinden.

Gib das Folgende auf der Kommandozeile ein (ersetze `<your-github-username>` mit dem Benutzernamen, den du beim Erstellen deines GitHub-Accounts gewählt hast, aber ohne die spitzen Klammern -- die URL sollte der clone-URL entsprechen, die du vorhin gerade gesehen hast):

command-line

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

Wenn du zu GitHub pushst, wirst du nach deinem Benutzernamen und Passwort gefragt (entweder direkt im Kommandozeilen-Fenster oder in einem Pop-Up-Fenster), und nach der Eingabe deiner Zugangsdaten solltest du etwas Ähnliches wie das hier sehen:

command-line

```
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/ola/my-first-blog.git

 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Dein Code ist jetzt auf GitHub. Schau gleich mal nach! Dort ist dein Code in guter Gesellschaft - [Django](#), das [Django Girls Tutorial](#) und viele andere großartige Open Source Software-Projekte haben ihren Code auf GitHub. :)

## Deinen Blog auf PythonAnywhere einrichten

### Registriere dich für ein PythonAnywhere Konto

**Hinweis:** Es ist möglich, dass du bereits ein PythonAnywhere Konto angelegt hast. Wenn ja, dann brauchst du das nicht noch einmal zu tun.

PythonAnywhere ist ein Dienst, mittels dem Python auf Servern "in der Cloud" ausgeführt werden kann. Wir werden ihn verwenden, um unsere Seite zu hosten, live und im Internet.

Wir werden den Blog, den wir bauen, auf PythonAnywhere hosten. Registriere dich auf PythonAnywhere für ein "Beginner"-Konto (die kostenfreie Stufe ist ausreichend, du brauchst keine Kreditkarte).

- [www.pythonanywhere.com](http://www.pythonanywhere.com)

## Plans and pricing

### Beginner: Free!

A limited account with one web app at `your-username.pythonanywhere.com`, restricted outbound Internet access from your apps, low CPU/bandwidth, no IPython/Jupyter notebook support.  
It works and it's a great way to get started!

[Create a Beginner account](#)

**Hinweis** Beachte bei der Wahl deines PythonAnywhere-Benutzernamens, dass die URL deines Blogs die Form `deinbenutzername.pythonanywhere.com` haben wird. Wähle also einen Namen, unter dem du veröffentlichen willst, oder einen Namen, der den Inhalt deines Blogs beschreibt. Und vergiss dein Passwort nicht (füge es deinem Passwortmanager hinzu, wenn du einen benutzt).

## Erstellen eines PythonAnywhere API-Tokens

Das Folgende musst du nur einmal machen. Wenn du dich auf PythonAnywhere angemeldet hast, kommst du aufs "Dashboard". Oben rechts findest du den Link zu deiner "Account"-Seite:

The screenshot shows the PythonAnywhere dashboard. At the top, there is a blue header bar with white text containing links: "Send feedback", "Forums", "Help", "Blog", "Account", and "Log out". Below the header, the PythonAnywhere logo is on the left, followed by the word "pythonanywhere". To the right of the logo is a navigation menu with tabs: "Dashboard" (which is bolded), "Consoles", "Files", "Web", "Tasks", and "Databases". A red arrow points upwards from the bottom of the page towards the "Databases" tab. In the center of the dashboard, there is a yellow warning box with the text: "Warning You have not confirmed your email address yet. This means that you will not be able to reset your password if you lose it. If you cannot find your confirmation email anymore, send yourself a new one [here](#)."

### Dashboard

Klicke dort auf den Reiter namens "API token" und dann auf den Knopf, der mit "Create new API token" beschriftet ist.

[Upgrade/Downgrade Account](#)

[Email and security](#)

[Teacher](#)

[API Token](#)

## Your API token

You do not have an API token yet.

[Create a new API token](#)

*By clicking this button you agree that you understand that this API is new and*

## **Unsere Site auf PythonAnywhere konfigurieren**

Gehe zurück zum [Haupt-Dashboard PythonAnywhere](#), indem du auf das Logo klickst. Dann wähle die Option zum Start einer "Bash"-Konsole – die PythonAnywhere Version einer Kommandozeile wie du sie auf deinen Computer hast.

# Recent Consoles

+ 5 -

*You have no recent consoles.*

[View all](#)

## New console:

\$ Bash

>>> Python ▾

[More...](#)

**Hinweis:** PythonAnywhere basiert auf Linux. Wenn du Windows benutzt, dann sieht die Konsole etwas anders aus als die Konsole auf deinem Computer.

Um eine Web App auf PythonAnywhere publizieren zu können, muss dein Code von GitHub heruntergeladen und PythonAnywhere dazu gebracht werden, diesen zu erkennen und als Web Applikation anzubieten. Du kannst das auch manuell machen. Aber PythonAnywhere stellt ein Hilfstoß zur Verfügung, das das alles für dich erledigt. Lass es uns installieren:

PythonAnywhere command-line

```
$ pip3.6 install --user pythonanywhere
```

Nach diesem Befehl solltest du in etwa Folgendes sehen: `collecting pythonanywhere`, und irgendwann den Schluss `Successfully installed (...) pythonanywhere- (...)`.

Nun können wir mit dem Hilfstoßtoll unsere App von GitHub automatisch konfigurieren. Gib das Folgende in der Konsole auf PythonAnywhere ein (vergiss nicht, deinen GitHub-Benutzernamen an Stelle von `<your-github-username>` zu benutzen, so dass die URL der clone-URL von GitHub entspricht):

PythonAnywhere command-line

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

Während du die Ausführung verfolgst, wirst du sehen, was passiert:

- Den Code von GitHub herunterladen
- Eine virtuelle Umgebung auf PythonAnywhere einrichten, genau wie die auf deinem eigenen Computer
- Deine Einstellungen mit ein paar Veröffentlichungseinstellungen aktualisieren
- Eine Datenbank auf PythonAnywhere einrichten mit dem Befehl `manage.py migrate`
- Deine statischen Dateien einrichten (darauf lernen wir später etwas)
- PythonAnywhere so einrichten, dass es deine Web-App über seine Schnittstelle (API) präsentieren kann

Diese Schritte wurden auf PythonAnywhere automatisiert, aber es sind die selben Schritte, die du bei jedem anderen Server-Provider machen müsstest.

Das Wichtigste im Moment ist, dass du weißt, dass Deine Datenbank auf PythonAnywhere vollständig unabhängig von deiner Datenbank auf deinem eigenen PC ist, so dass sie unterschiedliche Posts und Administratorenkonten haben kann. Aus diesem Grund müssen wir das Administratorenkonto mittels `createsuperuser` initialisieren - wie wir das auf deinem eigenen Computer getan haben. PythonAnywhere hat deine virtualenv automatisch für dich aktiviert. Du musst nur noch Folgendes ausführen:

PythonAnywhere command-line

```
(ola.pythonanywhere.com) $ python manage.py createsuperuser
```

Trage die Informationen für deinen Administrator ein. Am Besten verwendest du die selben Daten wie auf deinem eigenen Computer, um Verwechslungen zu vermeiden - es sei denn, du willst das Passwort auf PythonAnywhere sicherer machen.

Nun kannst auch einen Blick auf deinen Code auf PythonAnywhere werfen mittels `ls`:

PythonAnywhere command-line

```
(ola.pythonanywhere.com) $ ls
blog db.sqlite3 manage.py mysite requirements.txt static
(ola.pythonanywhere.com) $ ls blog/
__init__.py __pycache__ admin.py apps.py migrations models.py
tests.py views.py
```

Du kannst auch auf die "Files"-Seite gehen und mit PythonAnywheres eingebautem Datei-Manager navigieren. (Von der "Console"-Seite gelangst du über das Menü in der rechten oberen Ecke zu anderen PythonAnywhere-Seiten. Sobald du auf einer dieser Seiten bist, findest du die Links zu den anderen Seiten oben über dem Seiteninhalt.)

## Du bist jetzt live!

Nun ist deine Site also live im öffentlichen Internet! Klick dich zur PythonAnywhere "Web"-Seite durch und hole dir den Link. Teile ihn, so oft du willst :)

**Hinweis:** Da es sich hier um ein Anfänger-Tutorial handelt, haben wir ein paar Abkürzungen genommen, um die Seite zu veröffentlichen, welche sicherheitstechnisch nicht ideal sind. Fall du dich entscheidest, dieses Projekt weiterzubauen oder ein neues Projekt anzufangen, dann solltest du die [Django deployment checklist](#) durchgehen, um einige Tipps zur Absicherung deiner Seite zu erhalten.

## Debugging Tipps

Solltest du beim Ausführen des `pa_autoconfigure_django.py` Skripts eine Fehlermeldung erhalten, findest du folgend ein paar bekannte Gründe hierfür:

- Du hast vergessen deinen PythonAnywhere API-Token zu erstellen.
- Du hast in deiner GitHub-URL einen Fehler gemacht.
- Falls du die Fehlermeldung "*Could not find your settings.py*" erhältst, liegt das wahrscheinlich daran, dass du nicht alle Files zum Git hinzugefügt und/oder diese nicht erfolgreich auf GitHub veröffentlicht hast. Schau dir nochmals den Git-Abschnitt weiter oben an.

Falls du eine Fehlermeldung erhältst, wenn du versuchst, deine Site aufzurufen, solltest du als Erstes die Debugging-Informationen im **error log** anschauen. Den Link dazu findest du über [die PythonAnywhere-Seite "Web"](#). Schau nach, ob darin Fehlermeldungen enthalten sind; die neuesten findest du ganz unten.

Du findest einige [Allgemeine Debugging Tipps im PythonAnywhere Wiki](#).

Und denke daran, dein Coach ist da, um zu helfen!

## Schau dir deine Website an!

Auf der Defaultseite deiner Site sollte "It worked!" stehen - genau so wie auf deinem lokalen Computer. Füge nun `/admin/` ans Ende deiner URL an und du kommst auf die Admin-Site. Logge dich mit Benutzernamen und Passwort ein, und du wirst sehen, dass du auf dem Server neue Posts hinzufügen kannst -- die Posts aus deiner lokalen Test-Datenbank wurden ja nicht auf deinen öffentlichen Blog geschickt.

Wenn du ein paar Posts erstellt hast, kannst du zurück auf dein lokales Setup (nicht PythonAnywhere) wechseln. Ab jetzt solltest du für Änderungen auf deinem lokalen Setup arbeiten. So wird in der Web-Entwicklung gearbeitet - Änderungen lokal machen und diese dann auf GitHub veröffentlichen und dann deine Änderungen auf den produktiven Webserver ziehen. So kannst du Sachen ausprobieren, ohne deine produktive Website kaputt zu machen. Ziemlich cool, oder?

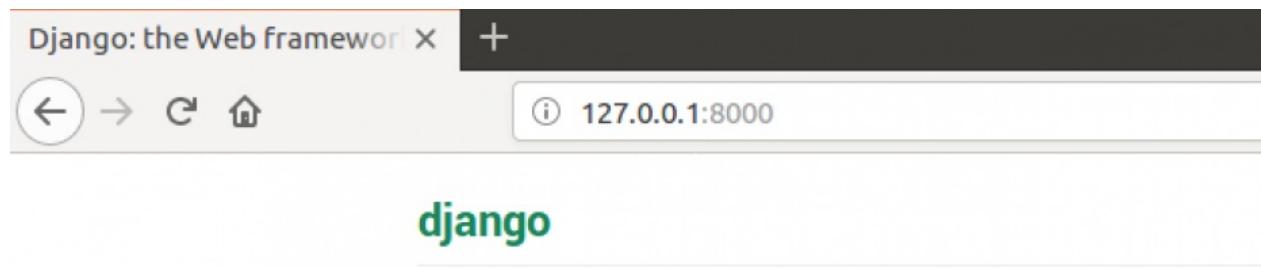
Klopft dir *kräftig* auf die Schulter! Server-Deployment ist eines der kompliziertesten Dinge der Web-Entwicklung und es dauert oftmals mehrere Tage, bis alles läuft. Aber du hast deine Site jetzt live, im echten Internet!

# Django-URLs

Gleich werden wir unsere erste Website basteln: eine Homepage für deinen Blog! Zuerst sollten wir uns jedoch mit Django URLs beschäftigen.

## Was ist eine URL?

Eine URL ist eine Web-Adresse. Jedes Mal, wenn du eine Website besuchst, kannst du eine URL sehen - sie ist in der Adressleiste des Browsers sichtbar. (Ja! `127.0.0.1:8000` ist eine URL! Und `https://djangogirls.org` ist auch eine URL.)



Jede Seite im Internet braucht ihre eigene URL. Dadurch weiß deine Applikation, was sie dem Nutzer, der eine URL öffnet, zeigen soll. In Django verwenden wir eine sogenannte `URLconf` (URL-Konfiguration). URLconf ist eine Ansammlung von Mustern, die Django mit der empfangenen URL abgleicht, um die richtige View zu finden, das heißt, um letztlich die richtige Seite für den Nutzer anzuzeigen.

## Wie funktionieren URLs in Django?

Öffne die `mysite/urls.py`-Datei in deinem Code-Editor nach Wahl und schaue dir an, wie sie aussieht:

`mysite/urls.py`

```
"""mysite URL Configuration

[...]
"""

from django.contrib import admin
from django.urls import path

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Wie du siehst, hat Django hier schon etwas für uns eingefügt.

Zeilen zwischen dreifachen Gänsefüßchen (`'''` oder `"""`) heißen Docstrings (Kommentare) - man kann sie am Anfang der Datei, Klasse oder Methode platzieren, um zu beschreiben, was sie tut. Sie werden von Python nicht ausgeführt.

Die admin-URL, die du im vorangegangenen Kapitel bereits besucht hast, ist schon da:

mysite/urls.py

```
path('admin/', admin.site.urls),
```

Diese Zeile bedeutet, dass Django für jede URL, die mit `admin/` beginnt, die entsprechende View finden wird. Hier wird mit `admin.site.urls` eine ganze Sammlung von admin-URLs referenziert. Dadurch müssen nicht alle in dieser kleinen Datei aufgeführt werden und sie bleibt lesbarer und übersichtlicher.

## Deine erste Django-URL!

Es wird Zeit, unsere erste URL zu erstellen! Wir wollen, dass '<http://127.0.0.1:8000/>' die Homepage unseres Blogs wird und eine Liste unserer Posts zeigt.

Wir wollen auch, dass die `mysite/urls.py`-Datei sauber bleibt. Deshalb importieren wir die URLs unserer `Blog`-Applikation in die `mysite/urls.py`-Hauptdatei.

Also los: Füge eine Zeile hinzu, die `blog.urls` importiert. Außerdem wirst du die Zeile `from django.urls...` ändern müssen, da wir hier die Funktion `include` verwenden, die du in dieser Zeile noch importieren musst.

Deine `mysite/urls.py`-Datei sollte jetzt so aussehen:

mysite/urls.py

```
from django.contrib import admin
from django.urls import path, include

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

Django wird nun alle Aufrufe von '<http://127.0.0.1:8000/>' auf `blog.urls` umleiten und dort nach weiteren Anweisungen schauen.

## blog.urls

Erstelle eine neue leere Datei namens `urls.py` im Verzeichnis `blog` und öffne sie im Code-Editor. Alles klar! Füge folgende zwei Zeilen ein:

blog/urls.py

```
from django.urls import path
from . import views
```

Hier importieren wir die Django-Funktion `path` und alle unsere `views` aus unserer `blog`-Applikation. (Wir haben noch keine, aber dazu kommen wir gleich!)

Jetzt können wir unser erstes URL-Muster hinzufügen:

blog/urls.py

```
urlpatterns = [
    path('/', views.post_list, name='post_list'),
]
```

Wie du siehst, fügen wir nun eine `view` mit dem Namen `post_list` zur Root-URL hinzu. Leere Zeichenfolgen passen auf dieses Muster und der Django-URL-Resolver ignoriert den Domain-Namen (z.B. <http://127.0.0.1:8000/>), der im vollständigen Pfad voransteht. Dieses Muster sagt Django also, dass `views.post_list` das gewünschte Ziel ist, wenn jemand deine Website über '<http://127.0.0.1:8000/>' aufruft.

Der letzte Teil `name='post_list'` ist der Name der URL, der genutzt wird, um die View zu identifizieren. Er kann identisch mit dem Namen der View sein, aber es kann auch ein komplett anderer sein. Wir werden später die Namen der URLs im Projekt benutzen. Daher ist es wichtig, jede URL in der App zu benennen. Wir sollten außerdem versuchen, eindeutige und einfach zu merkende URL-Namen zu wählen.

Wenn du jetzt versuchst, <http://127.0.0.1:8000/> aufzurufen, dann erscheint eine Meldung der Art "Webseite nicht verfügbar". Das erscheint deshalb, weil der Server nicht mehr läuft. (Erinnerst du dich noch, `runserver` eingegeben zu haben?) Schau mal in deiner Server-Konsole nach und finde heraus, warum der Server nicht mehr läuft.

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

Die Konsole zeigt einen Fehler, aber keine Sorge – der ist eigentlich ziemlich nützlich: Er sagt dir, dass **kein Attribut 'post\_list'** vorhanden ist. Das ist der Name der View, die Django zu finden und zu verwenden versucht, aber wir haben sie noch gar nicht erstellt. In diesem Zustand wird dein `/admin/` auch nicht funktionieren. Keine Sorge, das regeln wir gleich. Wenn du eine andere Fehlermeldung siehst, versuche es nochmal nach einem Neustart des Webservers. Um das zu tun, stoppst du den Webserver, indem du im Kommandozeilen-Fenster, in dem er läuft, Strg+C bzw. Ctrl+C drückst (Strg-/Ctrl-Taste und C-Taste zusammen), und startest ihn danach mit dem Kommando `python manage.py runserver neu`.

Wenn du mehr über Django-URLconfs lernen willst, dann öffne die offizielle Dokumentation:  
<https://docs.djangoproject.com/en/2.0/topics/http/urls/>

# Django-Views - leg los!

Es wird jetzt Zeit, den Bug, den wir im letzten Kapitel erzeugt haben, zu beheben! :)

In der *View* schreiben wir die Logik unserer Anwendung. So werden Informationen aus dem `Model` abgefragt werden, welches du zuvor erzeugt hast und diese werden an ein `Template` weitergegeben. Ein Template erzeugen wir im nächsten Kapitel. Views sind Python-Funktionen, die ein bisschen komplizierter sind als die, die wir im Kapitel **Einführung in Python** kennengelernt haben.

Views kommen in die `views.py` Datei. Wir fügen nun also unsere *Views* zur Datei `blog/views.py` hinzu.

## blog/views.py

OK, wir öffnen jetzt diese Datei in unserem Code-Editor und schauen, was darin steht:

blog/views.py

```
from django.shortcuts import render

# Create your views here.
```

OK, hier steht noch nicht so viel.

Denk daran, Zeilen mit einem `#` am Anfang sind Kommentare – das bedeutet, dass diese Zeilen von Python nicht ausgeführt werden.

Lass uns eine *View* erstellen, wie der Kommentar das vorschlägt. Füge die folgende Mini-View unter dem Kommentar ein:

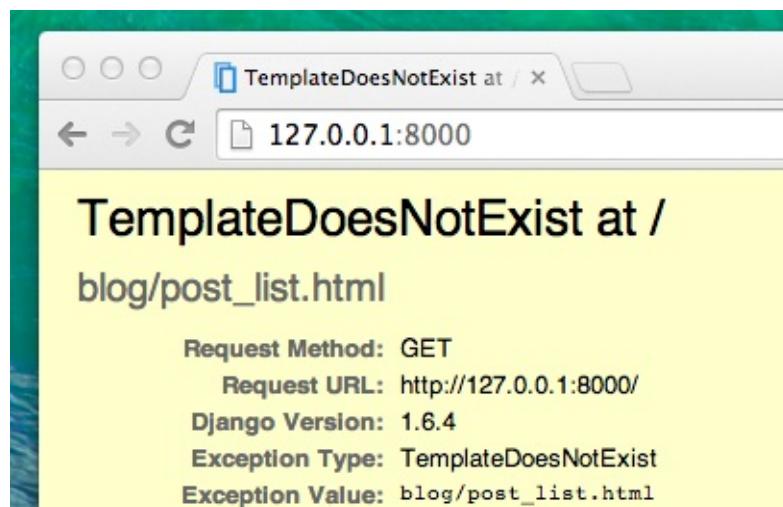
blog/views.py

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

Wie du siehst, definieren wir eine Funktion (`def`) mit dem Namen `post_list`, die den Parameter `request` entgegen nimmt und die mit `return` den Rückgabewert einer anderen Funktion namens `render` zurück gibt. Letztere wird unser Template `blog/post_list.html` "rendern" (zu einer fertigen HTML-Seite zusammensetzen).

Speichere die Datei, öffne <http://127.0.0.1:8000/> im Browser und schau nach, was wir jetzt haben.

Einen anderen Fehler! Lies, was da jetzt los ist:



Das zeigt, dass der Server zumindest wieder läuft, aber es sieht immer noch nicht richtig aus, oder? Mach dir keine Sorgen, es ist nur eine Fehlerseite, nichts zu befürchten! Genau wie die Fehlermeldungen in der Konsole sind auch die hier ziemlich nützlich. Du erhältst den Hinweis *TemplateDoesNotExist*, also dass es das Template nicht gibt. Lass uns diesen Bug beheben, indem wir im nächsten Kapitel ein Template erstellen!

Erfahre mehr über Django Views in der offiziellen Dokumentation:

[https://docs.djangoproject.com/en/2.0/topics/http/views/.](https://docs.djangoproject.com/en/2.0/topics/http/views/)

# Einführung in HTML

Vielleicht fragst du dich, was ein Template (Vorlage) ist?

Ein Template (zu deutsch "Vorlage") ist eine Textdatei und ermöglicht es uns, verschiedene Inhalte in einer einheitlichen Darstellung zu erzeugen. Eine Vorlage für z.B. einen Brief hilft uns, immer gleich aussehende Nachrichten zu versenden, in denen sich Empfänger, Betreff und Text jeweils ändern, das äußere Format jedoch bestehen bleibt.

Ein Django-Template wird mit einer Sprache namens HTML beschrieben. (Genau das HTML aus dem ersten Kapitel: **Wie das Internet funktioniert**).

## Was ist HTML?

HTML ist recht einfacher Code, der von deinem Browser – z.B. Chrome, Firefox oder Safari – interpretiert wird, um dem Benutzer eine Website darzustellen.

HTML steht für "HyperText Markup Language". Als **HyperText** wird Text bezeichnet, der über markierte Textstellen, den "Hyperlinks" (die umgangssprachlichen "Links"), auf andere (meist ebenfalls in HTML geschriebene) Seiten verweist. **Markup** bedeutet, dass wir ein Dokument nehmen und mit Code versehen, um einem Empfänger mitzuteilen (in diesem Fall dem Browser), wie diese Seite interpretiert werden muss. HTML-Code besteht aus **Tags**, wovon jeder mit `<` beginnt und mit `>` endet. Diese Tags stellen die **Markup-Elemente** dar.

## Dein erstes Template!

Ein Template zu erstellen, heißt, eine entsprechende Datei dafür zu erstellen. Alles ist eine Datei, wie du vielleicht schon bemerkt hast.

Templates werden im Verzeichnis `blog/templates/blog` gespeichert. Als Erstes erzeugen wir das Verzeichnis `templates` in deinem Blog-Verzeichnis. Im Template-Verzeichnis selbst erstellen wir ein weiteres Verzeichnis `blog` :

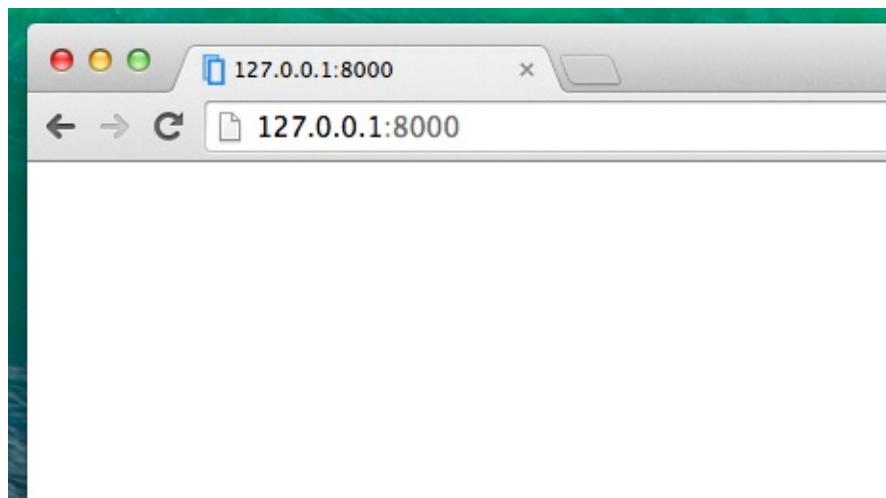
```
blog
└── templates
    └── blog
```

(Falls du dich wunderst, warum wir zwei `blog`-Verzeichnisse brauchen – das hängt mit den nützlichen Namenskonventionen von Django zusammen, die das Leben einfacher machen, wenn deine Projekte immer komplizierter und komplexer werden.)

Als nächstes erstellen wir eine Datei `post_list.html` (erst mal ohne Inhalt) innerhalb des Verzeichnisses `blog/templates/blog`.

Kontrolliere deine überarbeitete Webseite unter: <http://127.0.0.1:8000>

Falls du die Fehlermeldung `TemplateDoesNotExist` angezeigt bekommst, versuche den Server neu zu starten. Auf der Kommandozeile drückst du dafür Strg+C bzw. Ctrl+C (Strg-/Ctrl- und die C-Taste C zusammen) und startest danach den Server erneut mit dem Kommando `python manage.py runserver`.



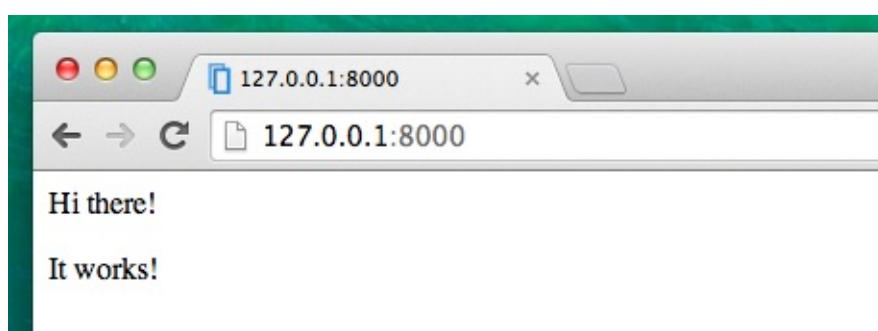
Der Fehler sollte weg sein! Toll :) Deine Webseite ist allerdings noch leer, weil dein Template leer ist. Das müssen wir ändern.

Öffne die neue Datei im Code-Editor, und füge Folgendes hinzu:

blog/templates/blog/post\_list.html

```
<html>
<body>
    <p>Halli-Hallo!</p>
    <p>Es funktioniert!</p>
</body>
</html>
```

Hat sich die Seite geändert? Besuche <http://127.0.0.1:8000>, um nachzusehen.



Es funktioniert! Gute Arbeit! :)

- Jede Webseite sollte mit dem Tag `<html>` beginnen. Und `</html>` steht immer am Ende. Zwischen den beiden Tags `<html>` und `</html>` steht der gesamte Inhalt der Webseite
- `<p>` ist der Tag für ein Absatz-Element (paragraph), `</p>` beendet einen Absatz

## "Head" und "body"

Jede HTML-Seite gliedert sich in zwei Teile: **head** und **body**.

- Das Element **head** speichert im "Kopf" der Seite Informationen über die Seite, die dir nicht angezeigt werden.
- Das Element **body** enthält den "Körper" - also den dargestellten Inhalt der Seite.

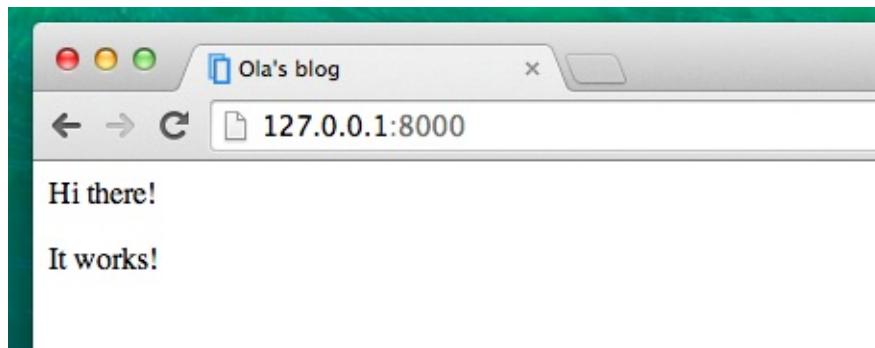
Im `<head>` informieren wir den Browser über Einstellungen und Konfigurationen der Webseite, z.B. wie sie dargestellt werden soll, und im `<body>` darüber, was tatsächlich dargestellt werden soll.

In den `<head>` können wir z.B. noch den Titel (`title`) der Seite mit aufnehmen:

blog/templates/blog/post\_list.html

```
<html>
  <head>
    <title>Olas Blog</title>
  </head>
  <body>
    <p>Halli-Hallo!</p>
    <p>Es funktioniert!</p>
  </body>
</html>
```

Speichere die Datei und aktualisiere die Seite im Browser.



Der Titel "Olas Blog" wird nun im Browser angezeigt. Hast du es bemerkt? Der Browser hat `<title>Olas Blog</title>` interpretiert und in die Titelleiste übernommen (dieser Titel wird auch in den Lesezeichen usw. verwendet).

Wie du vielleicht bemerkst, hat jedes Element zu Beginn einen öffnenden Tag und einen zugehörigen *schließenden Tag* mit `/` und innerhalb davon sind Elemente *eingebettet*. Ein innerer Tag kann nicht außerhalb des umschließenden Tags geschlossen werden, die Reihenfolge muss immer stimmen.

Es ist, wie wenn man Sachen in Kisten steckt. Du hast eine große Kiste, `<html></html>`. In der ist als weitere, etwas kleinere Kiste `<body></body>` drin, und in der wiederum weitere kleine Kistchen: `<p></p>`.

Die Regeln und Reihenfolgen von *schließenden Tags* und *Verschachtelung* der Elemente musst du immer einhalten. Andernfalls können Browser die Seite nicht richtig interpretieren und darstellen.

## Dein Template anpassen

Jetzt kannst du ein bisschen rumprobieren und dein Template umgestalten! Hier sind ein paar nützliche Tags dafür:

- `<h1>Überschrift</h1>` (headline) für wichtigste Überschriften
- `<h2>Unter-Überschrift</h2>` die nächst tiefere Überschiftenebene
- `<h3>Unter-Unter-Überschrift</h3>` ... und so weiter bis `<h6>`
- `<p>Ein Fliesstext-Absatz</p>`
- `<em>Text</em>` hebt deinen Text hervor
- `<strong>Text</strong>` hebt deinen Text stark hervor
- `<br>` fängt eine neue Zeile an (du kannst nichts in das br schreiben und es gibt keinen schließenden Tag)
- `<a href="https://.djangoproject.org">link</a>` erstellt einen Link
- `<ul><li>Erster Punkt</li><li>second item</li></ul>` generiert eine Liste so wie diese hier!

- `<div></div>` definiert einen Abschnitt auf einer Seite

Hier ist ein vollständiges Beispiel eines Templates. Kopiere es und füge es in `blog/templates/blog/post_list.html` ein:

`blog/templates/blog/post_list.html`

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

    <div>
      <p>published: 14.06.2014, 12:14</p>
      <h2><a href="">My first post</a></h2>
      <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id e
lit non mi porta gravida at eget metus. Fusce dapibus, tellus ac cursus commodo, tortor mauris condiment
um nibh, ut fermentum massa justo sit amet risus.</p>
    </div>

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

Wir haben hier drei verschiedene `div` Abschnitte erstellt.

- Das erste `div` Element enthält den Titel unseres Blogs – eine Überschrift und einen Link
- Zwei weitere `div` Elemente beinhalten unsere Blogposts und ein Publikationsdatum, `h2` mit dem Titel des Posts und zwei `p` (paragraph) Tags mit Text, eines für das Datum und eines für den Blogpost.

Wir bekommen das Folgende:

The screenshot shows a web browser window titled "Django Girls blog" with the URL "127.0.0.1:8000". The page displays two blog posts:

- My first post**  
published: 14.06.2014, 12:14  
Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec massa justo sit amet risus.
- My second post**  
published: 14.06.2014, 12:14  
Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec massa justo sit amet risus.

Yaaay! Bis jetzt zeigt unser Template aber immer genau die **gleichen Inhalte** – obwohl wir aber vorhin davon gesprochen haben, dass Templates uns erlauben, **verschiedene** Informationen im **gleichen Format** darzustellen.

Eigentlich wollen wir richtige Posts anzeigen, die in unserer Django-Admin-Oberfläche hinzugefügt wurden – und das wollen wir als Nächstes tun.

## Noch eine Sache: Deployment!

Es wäre gut, das alles live draußen im Internet zu sehen, oder? Lass uns noch eine PythonAnywhere-Anwendung erstellen:

### Committe und pushe deinen Code auf GitHub

Lass uns nachsehen, welche Dateien sich nach dem letzten Veröffentlichen (Deployment) geändert haben. (Führe diese Befehle lokal aus und nicht auf PythonAnywhere):

command-line

```
$ git status
```

Stelle sicher, dass du im `djangogirls` Verzeichnis bist und sag `git`, dass alle Änderungen in diesem Verzeichnis hinzugefügt werden sollen:

command-line

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

**Beachte:** `--all` bedeutet, dass `git` auch Dateien berücksichtigt, die du gelöscht hast (in der Standardeinstellung werden nur neue/geänderte Dateien hinzugefügt). Denk auch daran (Kapitel 3), dass `.` das aktuelle Verzeichnis meint.

Bevor wir alle Dateien hochladen, prüfen wir noch einmal, was `git` hochladen will (alle Dateien, die `git` hochladen wird, sind jetzt grün):

command-line

```
$ git status
```

Fast fertig, wir sagen nun noch, dass diese Änderung in der Verlaufsübersicht gespeichert werden soll. Wir erstellen eine "Commit Message", die beschreibt, was wir verändert haben. Du kannst an diesem Punkt hier alles reinschreiben, aber es ist sehr nützlich, etwas zu Sinnvolles einzutragen, damit du dich in Zukunft erinnern kannst, was du geändert hast.

command-line

```
$ git commit -m "HTML der Site geändert."
```

**Beachte:** Du musst Anführungszeichen um den Commit-Kommentar setzen.

Nachdem wir das gemacht haben, laden (push) wir unsere Änderungen auf GitHub:

command-line

```
$ git push
```

## Hol dir den neuen Code auf PythonAnywhere und aktualisiere deinen Browser

- Öffne die [PythonAnywhere consoles page](#) und gehe zu deiner **Bash-Konsole** (oder starte eine neue). Dann, führe Folgendes aus:

PythonAnywhere command-line

```
$ cd ~/<dein pythonanywhere-Benutzername>.pythonanywhere.com  
$ git pull  
[...]
```

(Denke daran <dein pythonanywhere-Benutzername> durch deinen PythonAnywhere-Benutzernamen zu ersetzen - ohne spitze Klammern).

Und sieh zu, wie dein Code heruntergeladen wird. Wenn du überprüfen willst, dass er angekommen ist, geh' hinüber zur **Seite "Files"** und schau deinen Code auf PythonAnywhere an (du kannst andere PythonAnywhere-Seiten über den Menü-Knopf auf der Konsolen-Seite erreichen).

- Spring anschließend rüber zur [Seite "Web"](#) und klick auf **Neu laden** in deinem Browser.

Dein Update sollte live sein! Lade die Seite neu in deinem Browser. Es sollten nun Änderungen zu sehen sein. :)

# Django-ORM und QuerySets

In diesem Kapitel lernst du, wie sich Django mit der Datenbank verbindet und Daten darin speichert. Lass uns loslegen!

## Was ist ein QuerySet?

Zusammengefasst ist ein QuerySet eine Liste von Objekten eines bestimmten Models. QuerySets erlauben es dir, Daten aus der Datenbank zu lesen, zu filtern und zu sortieren.

Am besten wir sehen uns das an einem Beispiel an. Versuchen wir's?

## Django-Shell

Öffne deine lokale Konsole (nicht in PythonAnywhere) und tippe dieses Kommando ein:

command-line

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

Das sollte angezeigt werden:

command-line

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

Nun bist du in der interaktiven Konsole von Django. Die funktioniert wie der Python-Prompt, aber hat noch etwas zusätzliche Django-Magie. :) Du kannst hier auch alle Python-Befehle verwenden.

## Alle Objekte

Zunächst wollen wir alle unsere Blogposts ansehen. Das kannst du mit folgendem Kommando erreichen:

command-line

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

Hoppa! Eine Fehlermeldung ist erschienen. Sie sagt uns, dass Python "Post" nicht kennt. Und sie hat recht: Wir haben vergessen zu importieren!

command-line

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

Wir importieren das Model `Post` aus `blog.models`. Versuchen wir nochmal, alle Posts anzuzeigen:

command-line

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

Das ist eine Liste all der Posts, die wir zuvor lokal erstellt haben! Wir haben diese Posts mit der Django-Admin-Oberfläche erstellt. Aber nun wollen wir weitere Posts mit Python erstellen! Wie geht das also?

## Objekt erstellen

So erstellst du ein neues Post-Objekt in der Datenbank:

command-line

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

Allerdings fehlt noch eine Zutat: `me`. Wir müssen eine Instanz des Models `User` als Autor übergeben. Wie macht man das?

Als Erstes müssen wir das User-Model importieren:

command-line

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

Welche User sind in unserer Datenbank vorhanden? Finde es damit heraus:

command-line

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

Das ist der Superuser, den wir vorhin erstellt haben! Lass uns jetzt eine Instanz des Users erstellen (passt diese Zeile an, so dass dein eigener Benutzername verwendet wird):

command-line

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

Wie du siehst, holen (`get`) wir jetzt ein `User`-Objekt mit einem `username` 'ola'. Prima!

Jetzt können wir schließlich unseren Post erstellen:

command-line

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

Super! Wollen wir nachsehen, ob es funktioniert hat?

command-line

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

Da ist er, ein weiterer Post in der Liste!

## Mehrere Posts hinzufügen

Du kann jetzt nach Belieben weitere Blogposts hinzufügen, um ein Gefühl dafür zu bekommen, wie es funktioniert. Füge doch noch zwei, drei hinzu bevor du zum nächsten Teil übergehst.

## Objekte filtern

Eine wichtige Eigenschaft von QuerySets ist, dass die Einträge gefiltert werden können. Zum Beispiel wollen wir alle Posts finden, die der User Ola geschrieben hat. Dafür nehmen wir `filter` statt `all` in `Post.objects.all()`. In Klammern schreiben wir die Bedingung(en), die ein Blogpost erfüllen muss, damit er in unser Queryset kommt. So soll jetzt z.B. `author` gleich `me` sein, damit nur die Blogposts des Autors "me" herausgefiltert werden. In Django schreiben wir deshalb: `author=me`. Jetzt sieht unser Code folgendermaßen aus:

command-line

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

Oder vielleicht wollen wir alle Posts haben, die das Wort "title" im `title`-Feld haben?

command-line

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

**Anmerkung:** Zwischen `title` und `contains` befinden sich zwei Unterstriche (`__`). Das ORM von Django nutzt diese Regel, um Feldnamen ("title") und Operationen oder Filter ("contains") voneinander zu trennen. Wenn du nur einen Unterstrich benutzt, bekommst du einen Fehler wie "FieldError: Cannot resolve keyword title\_contains".

Du kannst auch eine Liste aller bereits publizierten Posts erhalten, indem wir nach allen Posts suchen, deren `published_date` in der Vergangenheit liegt:

command-line

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

Unglücklicherweise ist der Post, den wir über die Python-Konsole hinzugefügt haben, noch nicht veröffentlicht. Aber das können wir ändern! Als Erstes holen wir eine Instanz des Posts, den wir veröffentlichen wollen:

command-line

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

Dann publizieren wir ihn mit unserer `publish`-Methode:

command-line

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

Jetzt versuch nochmal, eine Liste von veröffentlichten Posts zu bekommen (drücke dreimal "Pfeil nach oben" und `enter`):

command-line

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

## Objekte ordnen

Mit den QuerySets kannst du eine Liste auch nach bestimmten Kriterien ordnen. Lass uns das mit dem `created_date` Feld ausprobieren:

command-line

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

Wir können die Reihenfolge auch umdrehen, indem wir `" - "` davor schreiben:

command-line

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

## Komplexe Queries durch Methoden-Verkettung

Wie du gesehen hast, geben einige Methoden auf `Post.objects` ein QuerySet zurück. Die selben Methoden können wiederum auch auf einem QuerySet aufgerufen werden und geben dann ein neues QuerySet zurück. Das ermöglicht es, ihre Wirkung zu kombinieren, indem du die Methoden **verkettet**:

```
>>> Post.objects.filter(published_date__lte=timezone.now()).order_by('published_date')
<QuerySet [<Post: Post number 2>, <Post: My 3rd post!>, <Post: 4th title of post>, <Post: Sample title>]
>
```

Dies ist wirklich mächtig und lässt dich ziemlich komplexe Abfragen schreiben.

Cool! Jetzt bist du bereit für den nächsten Teil! Um die Konsole zu schließen, schreib das:

command-line

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

# Dynamische Daten in Templates

Wir haben nun schon einige Dinge an verschiedenen Orten fertiggestellt: das `Post`-Model ist in der `models.py` definiert, wir haben die `post_list` in der `views.py` und das Template hinzugefügt. Aber wie schaffen wir es nun, dass unsere Posts wirklich im HTML-Template erscheinen? Denn wir wollen ja erreichen, dass bestimmte Inhalte (die in der Datenbank gespeicherten Models) auf schöne Weise in unserem Template angezeigt werden, richtig?

Genau dafür sind die Views zuständig: die Verbindung zwischen den Models und den Templates. In unserer `post_list`-View müssen wir die Models, die wir anzeigen wollen, hernehmen und diese dem Template übergeben. In einer View entscheiden wir, was (welches Model) wir in einem Template anzeigen werden.

Okay, und wie machen wir das jetzt?

Wir öffnen unsere Datei `blog/views.py` in unserem Code-Editor. Bisher sieht unsere `post_list`-View folgendermaßen aus:

`blog/views.py`

```
from django.shortcuts import render

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

Erinnerst du dich, als wir davon gesprochen haben, dass wir den Code in verschiedene Dateien einfügen müssen? Jetzt ist es an der Zeit, das Model, dass wir in `models.py` beschrieben haben, einzufügen. Wir fügen den Befehl `from .models import Post` folgendermaßen ein:

`blog/views.py`

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

Der Punkt vor `models` bedeutet *aktuelles Verzeichnis* oder *aktuelle Anwendung*. Die Dateien `views.py` und `models.py` liegen im selben Verzeichnis. Deswegen können wir `.` und den Namen der Datei (ohne `.py`) benutzen. Dann ergänzen wir für den Import den Namen des Models (`Post`).

Und nun? Um vom `Post`-Model die tatsächlichen Blogposts zu erhalten, brauchen wir etwas, das `QuerySet` heißt.

## QuerySet

Dir sollte schon ungefähr klar sein, wie QuerySets funktionieren. Wir haben darüber im Kapitel [Django-ORM und QuerySets](#) gesprochen.

Wir wollen nun also eine Liste von publizierten Blogposts ausgeben, sortiert nach dem `published_date`, oder? Das haben wir bereits im Kapitel QuerySets gemacht!

`blog/views.py`

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

Öffnen wir also die Datei `blog/views.py` im Code-Editor und setzen wir dieses Stück Code in die Funktion `def post_list(request)` ein. Aber vergiss nicht, zuerst `from django.utils import timezone` hinzufügen:

`blog/views.py`

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

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

Als Letztes fehlt noch, dass wir das `posts` -QuerySet dem Template übergeben. Wie wir es dann anzeigen, erfährst du im nächsten Kapitel.

Beachte, dass wir eine *Variable* für unser QuerySet erstellt haben: `posts`. Das ist sozusagen der Name unseres QuerySets. Ab jetzt bezeichnen wir das QuerySet mit diesem Namen.

In der `render` -Funktion haben wir einen Parameter `request` (also alles, was wir vom User über das Internet bekommen) und einen weiteren Parameter, der den Template-Namen (`'blog/post_list.html'`) angibt. Der letzte Parameter, `{}`, ist eine Stelle, in der wir weitere Dinge hinzufügen können, die das Template dann benutzen kann. Wir müssen diesen einen Namen geben (wir verwenden einfach wieder `'posts'`). Es sollte nun so aussehen: `{'posts': posts}`. Bitte achte darauf, dass der Teil vor `:` ein String ist; das heißt, du musst ihn mit Anführungszeichen `''` umschließen.

Schließlich sollte deine `blog/views.py` Datei folgendermaßen aussehen:

`blog/views.py`

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

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

Das war's! Zeit, zurück zum Template zu gehen und das QuerySet anzuzeigen!

Wenn du mehr über QuerySets in Django erfahren willst, dann sieh unter diesem Link nach:

<https://docs.djangoproject.com/en/2.0/ref/models/querysets/>

# Django-Templates

Es wird Zeit, ein paar Daten anzuzeigen! Django bringt dafür bereits ein paar sehr hilfreiche **Template-Tags** mit.

## Was sind Template-Tags?

Also, in HTML kann man nicht wirklich Python-Code schreiben, weil es der Browser nicht verstehen würde. Denn der kennt nur HTML. Wir wissen, dass HTML ziemlich statisch ist, während Python dynamischer ist.

Die **Django-Template-Tags** erlauben uns, Python-artige Dinge ins HTML zu bringen, so dass man einfach und schnell dynamische Websites erstellen kann. Super!

## Anzeigen des Post-List-Templates

Im vorangegangen Kapitel haben wir unserem Template in der `posts`-Variable eine Liste von Posts übergeben. Diese wollen wir jetzt in HTML anzeigen.

Um eine Variable in einem Django-Template darzustellen, nutzen wir doppelte, geschweifte Klammern mit dem Namen der Variable drin, so wie hier:

`blog/templates/blog/post_list.html`

```
 {{ posts }}
```

Versuche das in deinem `blog/templates/blog/post_list.html` Template. Öffne es in deinem Code-Editor und ersetze alles vom zweiten `<div>` bis zum dritten `</div>` mit  `{{ posts }}` . Speichere die Datei und aktualisiere die Seite, um die Ergebnisse anzuzeigen.



Wie du siehst, haben wir nun das:

`blog/templates/blog/post_list.html`

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

Das heißt, Django versteht es als Liste von Objekten. Kannst du dich noch an die Einführung von Python erinnern, wie man Listen anzeigen kann? Ja, mit for-Schleifen! In einem Django-Template benutzt du sie so:

`blog/templates/blog/post_list.html`

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

Versuch das in deinem Template.



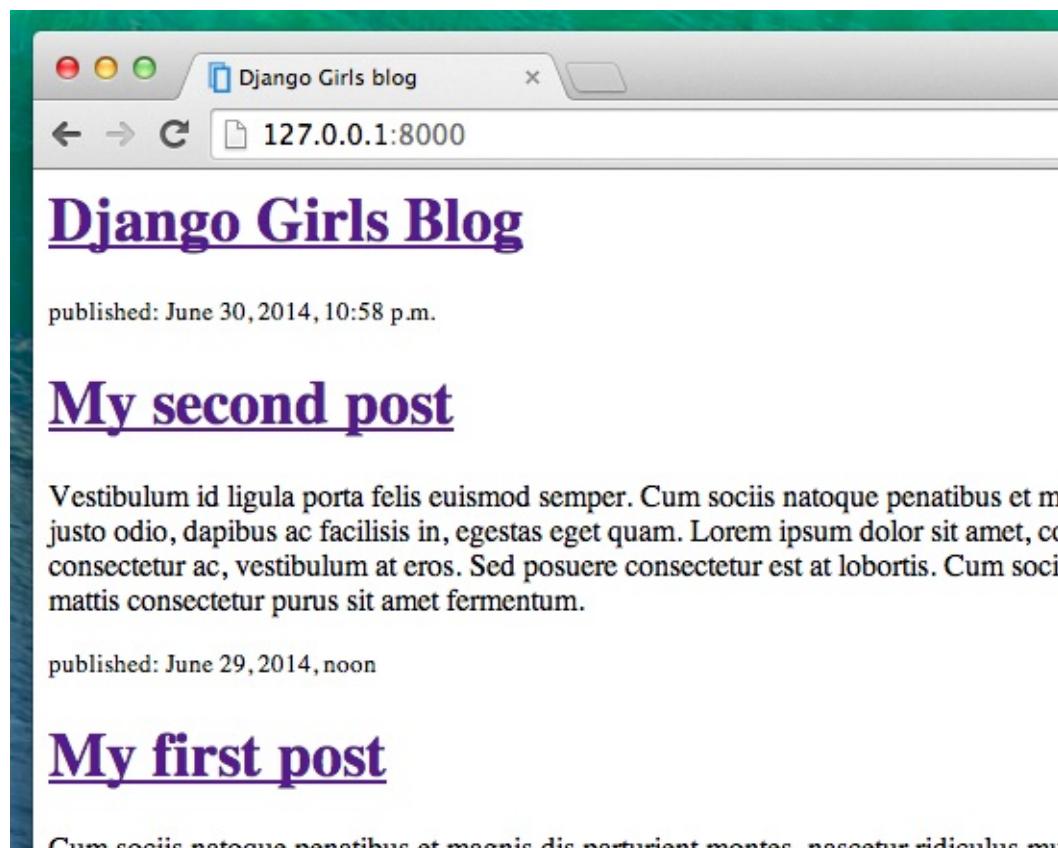
Es funktioniert! Aber wir wollen, dass die Posts so wie die statischen Posts angezeigt werden, die wir vorhin im **Einführung in HTML**-Kapitel erstellt haben. Du kannst HTML und Template Tags mischen. Unser `body` sollte dann so aussehen:

`blog/templates/blog/post_list.html`

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

{% for post in posts %}
    <div>
        <p>published: {{ post.published_date }}</p>
        <h2><a href="{{ post.title }}">{{ post.title }}</a></h2>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endfor %}
```

Alles, was du zwischen `{% for %}` und `{% endfor %}` schreibst, wird für jedes Objekt in der Liste wiederholt. Aktualisiere deine Seite:



## Und zum Schluss

Es wäre gut zu sehen, ob deine Website noch immer im öffentlichen Internet funktioniert, richtig? Lass uns versuchen, unsere Aktualisierungen wieder zu PythonAnywhere hochzuladen. Hier ist eine Zusammenfassung der Schritte...

- Schiebe zuerst deinen Code auf GitHub

command-line

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

- Dann logg dich wieder bei [PythonAnywhere](#) ein, geh zu deiner **Bash-Konsole** (oder starte eine neue) und gib ein:

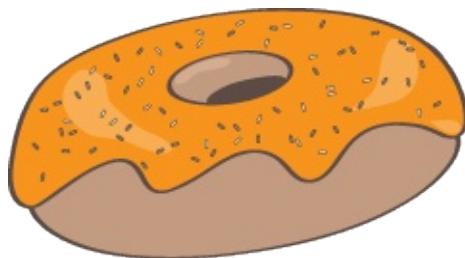
PythonAnywhere command-line

```
$ cd $USER.pythonanywhere.com
$ git pull
[...]
```

- Spring anschließend rüber zur Seite "Web" und klick auf **Neu laden** in deinem Browser. (Um von der Konsolen-Seite aus zu anderen PythonAnywhere-Seiten zu gelangen, benutze den Menü-Knopf in der rechten oberen Ecke.) Ein Update sollte auf <https://yourname.pythonanywhere.com> live sein -- guck's dir im Browser an! Wenn die Blogposts auf deiner PythonAnywhere-Seite anders sind als die auf deinem lokalen Server, ist das in Ordnung so. Die Datenbanken auf deinem lokalen Computer und auf PythonAnywhere werden nicht zusammen mit den restlichen Dateien abgeglichen.

Glückwunsch! Nun kannst du dich daran machen, neue Posts in deinem Django Admin zu erstellen (denk daran, auch ein published\_date einzufügen!). Stell sicher, dass du im Django Admin der PythonAnywhere Seite <https://yourname.pythonanywhere.com/admin> arbeitest. Dann aktualisiere die Seite und schau nach, ob die Posts dort erscheinen.

Funktioniert super? Wir sind so stolz auf dich! Steh kurz auf und geh ein Stück weg vom Computer. Du hast dir eine Pause verdient :)



# CSS - mach es hübsch!

Unser Blog sieht immer noch ziemlich unfertig aus, oder? Zeit, das zu ändern! Dafür nutzen wir CSS.

## Was ist CSS?

Cascading Style Sheets (CSS) ist eine Sprache, die das Aussehen und die Formatierung einer Website beschreibt. Es handelt sich wie bei HTML um eine Auszeichnungssprache (Markup Language). Sie ist sowas wie das "Make-up" unserer Website. ;)

Aber wir wollen nicht nochmal bei Null anfangen, oder? Einmal mehr werden wir etwas verwenden, dass ProgrammiererInnen entwickelt haben und gratis im Internet zur Verfügung stellen. Wir wollen ja nicht das Rad neu erfinden.

## Lass uns Bootstrap verwenden!

Bootstrap ist eines der bekanntesten HTML- und CSS-Frameworks für die Entwicklung von schönen Webseiten:  
<https://getbootstrap.com/>

Es wurde ursprünglich von ProgrammiererInnen bei Twitter geschrieben. Heute wird es von Freiwilligen aus der ganzen Welt weiterentwickelt!

## Bootstrap installieren

Öffne deine `.html`-Datei in deinem Code-Editor und füge Folgendes zum `<head>`-Abschnitt hinzu, um Bootstrap zu installieren:

`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">
```

Dadurch werden deinem Projekt keine Dateien hinzugefügt. Der Code verweist nur auf Dateien, die im Internet vorhanden sind. Öffne und aktualisiere also deine Webseite. Da ist sie!



Sie sieht jetzt schon viel schöner aus!

## Statische Dateien in Django

Endlich werden wir einen genaueren Blick auf die Dinge werfen, die wir bisher **statische Dateien** genannt haben. Statische Dateien sind alle deine CSS- und Bilddateien. Ihr Inhalt hängt nicht vom Requestkontext ab, sondern gilt für alle Benutzer gleichermaßen.

### Wohin kommen die statischen Dateien für Django

Django weiss schon, wo die statischen Dateien für die integrierte "admin"-App zu finden sind. Wir müssen noch die statischen Dateien für unsere `blog`-App hinzufügen.

Dies tun wir, indem wir einen Ordner namens `static` in der Blog-App erstellen:

```
djangogirls
└── blog
    ├── migrations
    └── static
        └── templates
└── mysite
```

Django findet automatisch alle Ordner mit dem Namen "static" in all unseren App-Ordnern. So ist es in der Lage, ihre Inhalte als statische Dateien zu nutzen.

## Deine erste CSS-Datei!

Erstellen wir nun eine CSS-Datei, um deiner Website deinen eigenen Stil zu verleihen. Erstelle ein neues Verzeichnis namens `css` in deinem `static`-Verzeichnis. Dann erstelle eine neue Datei namens `blog.css` in diesem `css`-Verzeichnis. Fertig?

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

Zeit, ein wenig CSS zu schreiben! Öffne die `blog/static/css/blog.css` Datei in Deinem Code-Editor.

Wir gehen nicht zu sehr auf die Details von CSS ein. Für diejenigen, die mehr über CSS lernen möchten, haben wir am Ende des Kapitels einen Link auf eine Empfehlung für einen kostenlosen CSS-Kurs angefügt.

Aber lass uns wenigstens etwas Kleines probieren. Beispielsweise könnten wir die Farbe unserer Kopfzeile ändern. Computer benutzen spezielle Codes, um Farben zu verstehen. Diese Codes starten immer mit `#`, danach folgen sechs Buchstaben (A-F) und Zahlen (0-9). Blau zum Beispiel ist `#0000FF`. Beispiele für solche Farbcodes findest du hier: <http://www.colorpicker.com/>. Du kannst auch vordefinierte Farben wie `red` und `green` benutzen.

In deiner `blog/static/css/blog.css` Datei änderst du den folgenden Code:

`blog/static/css/blog.css`

```
h1 a, h2 a {  
    color: #C25100;  
}
```

`h1 a` ist ein CSS-Selektor. Das bedeutet, dass wir für ein `a`-Element innerhalb eines `h1`-Elements einen Style hinzufügen; der `h2 a`-Selektor macht das gleiche für `h2`-Elemente. Wenn wir also etwas haben wie: `<h1><a href="">link</a></h1>` wird der `h1 a` Style angewandt. In diesem Fall sagen wir, dass die Farbe in `#C25100` geändert werden soll. Das ist ein dunkles Orange. Du kannst hier auch deine eigene Farbe verwenden, aber stelle sicher, dass sie einen guten Kontrast zum weißen Hintergrund hat!

In einer CSS-Datei werden Stile für Elemente der HTML-Datei festgelegt. Ein Weg, HTML-Elemente zu identifizieren, ist der Name des Elements. Du erinnerst dich vielleicht an diese Namen, die wir als 'Tags' im HTML Kapitel bezeichnet haben. Zum Beispiel sind `a`, `h1` und `body` solche Elementnamen. Wir identifizieren Elemente auch über die Attribute `class` oder `id`. Klassen (`class`) und IDs (`id`) sind Namen, die du den Elementen selbst gibst. Klassen definieren dabei Gruppen von Elementen und IDs verweisen auf bestimmte Elemente. Du könntest zum Beispiel den folgenden Tag anhand des Elementnamens `a`, der Klasse `external_link` oder der ID `link_to_wiki_page` identifizieren:

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

Auf [w3schools](#) erfährst du mehr über CSS-Selektoren.

Wir müssen der HTML-Vorlage noch sagen, dass wir CSS eingefügt haben. Öffne die Datei `blog/templates/blog/post_list.html` im Code-Editor und füge diese Zeile ganz oben ein:

`blog/templates/blog/post_list.html`

```
{% load static %}
```

Wir laden hier die statischen Dateien. :) Füge zwischen den Tags `<head>` und `</head>`, direkt nach den Links zu den Bootstrap-Dateien, noch diese Zeile ein:

`blog/templates/blog/post_list.html`

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

Der Browser liest die Dateien in der Reihenfolge, in der sie aufgeschrieben wurden. Darum müssen wir sicherstellen, dass die Zeile am richtigen Ort steht. Sonst könnte der Code der Bootstrap-Dateien den Code aus unserer Datei überschreiben. Wir haben also unserem Template gerade gesagt, wo sich die CSS-Datei befindet.

Deine Datei sollte jetzt so aussehen:

blog/templates/blog/post\_list.html

```
{% 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 }}</a></h2>
                <p>{{ post.text|linebreaksbr }}</p>
            </div>
        {% endfor %}
    </body>
</html>
```

OK, speichere die Datei und lade die Seite neu!



Gut gemacht! Vielleicht wollen wir unserer Webseite etwas mehr Luft geben, indem wir den Abstand auf der linken Seite vergrößern? Probieren wir es aus!

blog/static/css/blog.css

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

Füge dies zu deinem CSS hinzu, speichere die Datei und schau dir an, was passiert.



The screenshot shows a Mac OS X desktop with a browser window titled "Django Girls blog". The address bar displays "127.0.0.1:8000". The page content includes the title "Django Girls Blog" in a large orange font, a timestamp "published: June 30, 2014, 10:58 p.m.", and a post titled "My second post" with some placeholder text.

Velleicht können wir auch die Schrift in unserem HTML-Kopf anpassen? Füge dies zu `<head>` in `blog/templates/blog/post_list.html` hinzu:

`blog/templates/blog/post_list.html`

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

Wie eben bereits gemacht, prüfe die Reihenfolge und platziere die Anweisung vor dem Link `blog/static/css/blog.css`. Sie importiert eine Schriftart (engl. "Font") namens *Lobster* von Google Fonts (<https://www.google.com/fonts>).

Suche den Anweisungsblock: `h1 a` (der Code zwischen den geschweiften Klammern `{` und `}`) in der CSS Datei `blog/static/css/blog.css`. Nun füge die Zeile `font-family: 'Lobster';` zwischen den geschweiften Klammern hinzu und aktualisiere die Seite:

`blog/static/css/blog.css`

```
h1 a, h2 a {  
    color: #C25100;  
    font-family: 'Lobster';  
}
```



Super!

Wie oben erwähnt, basiert CSS auf dem Konzept von Klassen. Diese erlauben dir, einen Teil des HTML-Codes mit einem Namen zu versehen und die Darstellung dieses Teils separat von anderen Teilen mit einem Stil zu steuern. Das kann sehr hilfreich sein! Eventuell hast Du zwei 'div's die etwas vollkommen Verschiedenes auszeichnen (wie einen Seitentitel oder Post Beitrag). Die Klasse hilft dir, sie unterschiedlich aussehen zu lassen.

Geben wir also einigen Teilen deines HTML-Codes solche Namen. Füge eine Klasse ( `class` ) namens `page-header` dem `div` hinzu, das die Kopfzeilen (header) enthalten soll:

blog/templates/blog/post\_list.html

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

Jetzt fügen wir dem `div` für den Blog-Inhalt (Post) noch eine Klasse `post` hinzu.

blog/templates/blog/post\_list

```
<div class="post">
  <p>published: {{ post.published_date }}</p>
  <h2><a href="#">{{ post.title }}</a></h2>
  <p>{{ post.text|linebreaksbr }}</p>
</div>
```

Wir erweitern jetzt unser CSS mit entsprechenden Selektoren. Selektoren, die mit `. .` anfangen, beziehen sich auf Klassen im HTML. Es gibt im Internet viele gute Tutorials und Informationen über CSS, die dir helfen können, den folgenden Code besser zu verstehen. Kopiere zunächst folgenden Text in deine `blog/static/css/blog.css`-Datei:

blog/static/css/blog.css

```
.page-header {  
    background-color: #C25100;  
    margin-top: 0;  
    padding: 20px 20px 20px 40px;  
}  
  
.page-header h1, .page-header h1 a, .page-header h1 a:visited, .page-header h1 a:active {  
    color: #ffffff;  
    font-size: 36pt;  
    text-decoration: none;  
}  
  
.content {  
    margin-left: 40px;  
}  
  
h1, h2, h3, h4 {  
    font-family: 'Lobster', cursive;  
}  
  
.date {  
    color: #828282;  
}  
  
.save {  
    float: right;  
}  
  
.post-form textarea, .post-form input {  
    width: 100%;  
}  
  
.top-menu, .top-menu:hover, .top-menu:visited {  
    color: #ffffff;  
    float: right;  
    font-size: 26pt;  
    margin-right: 20px;  
}  
  
.post {  
    margin-bottom: 70px;  
}  
  
.post h2 a, .post h2 a:visited {  
    color: #000000;  
}
```

Der HTML-Code, der für die Anzeige der Blogposts verantwortlich ist, soll durch Klassen erweitert werden. Ersetze den folgenden Code:

blog/templates/blog/post\_list.html

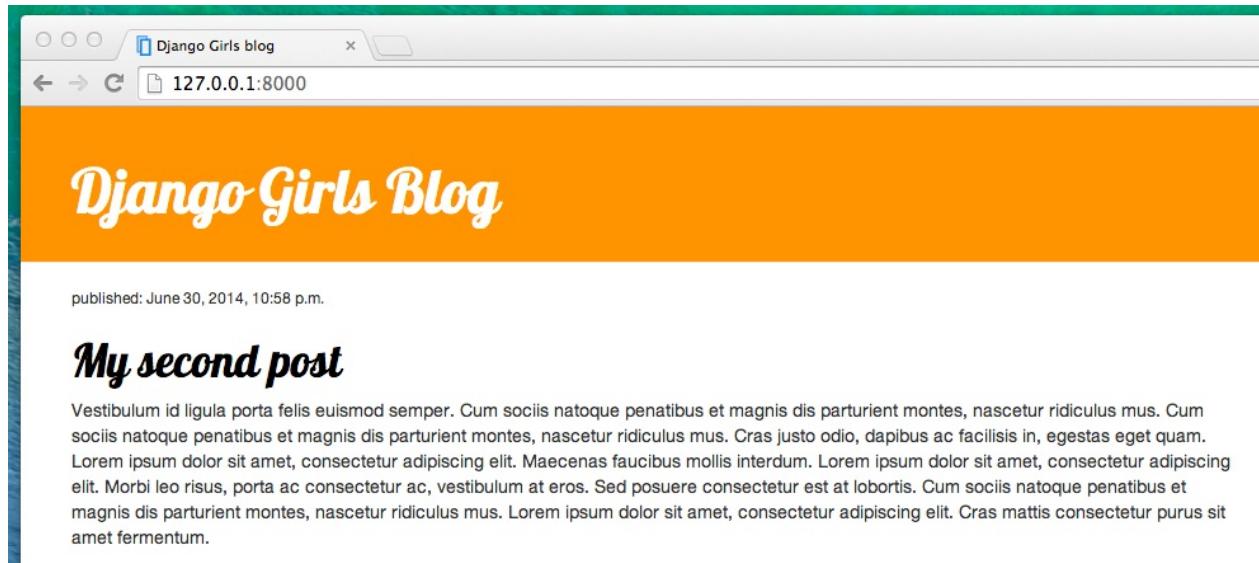
```
{% for post in posts %}  
    <div class="post">  
        <p>published: {{ post.published_date }}</p>  
        <h2><a href="">{{ post.title }}</a></h2>  
        <p>{{ post.text|linebreaksbr }}</p>  
    </div>  
{% endfor %}
```

in `blog/templates/blog/post_list.html` durch diesen:

`blog/templates/blog/post_list.html`

```
<div class="content container">
    <div class="row">
        <div class="col-md-8">
            {% for post in posts %}
                <div class="post">
                    <div class="date">
                        <p>published: {{ post.published_date }}</p>
                    </div>
                    <h2><a href="">{{ post.title }}</a></h2>
                    <p>{{ post.text|linebreaksbr }}</p>
                </div>
            {% endfor %}
        </div>
    </div>
</div>
```

Speichere die geänderten Dateien und aktualisiere die Webseite.



Juhuu! Sieht super aus, oder? Schau dir den Code an, den wir gerade eingefügt haben. Da siehst du, wo wir überall Klassen zu den HTML-Objekten hinzugefügt haben, um sie in CSS zu referenzieren. Wo würdest du eine Änderung machen, um das Datum in Türkis anzuseigen?

Hab keine Angst, etwas mit dieser CSS-Datei herumzuspielen, und versuche, ein paar Dinge zu ändern. Mit CSS herumzuspielen, kann dir helfen zu verstehen, was die verschiedenen Dinge genau machen. Mach dir keine Sorgen, wenn etwas kaputt geht, du kannst deine Änderungen immer rückgängig machen!

Wir empfehlen diesen kostenlosen [Codecademy HTML & CSS Kurs](#). Er wird dir helfen, deine Webseiten mit CSS schöner zu gestalten.

Bereit für das nächste Kapitel? :)

# Erweiterung der Templates

Eine weitere praktische Sache von Django ist das **template extending**, Erweiterungen des Templates. Was bedeutet das? Damit kannst du Teile deines HTML-Codes für andere Seiten deiner Website nutzen.

Templates helfen, wenn du die selben Informationen oder das selbe Layout an mehreren Stellen verwenden willst. Du musst dich so nicht in jeder Datei wiederholen. Und wenn du etwas ändern willst, dann musst du es nicht in jedem einzelnen Template tun, sondern nur in einem!

## Ein Basis-Template erstellen

Ein Basis-Template ist das grundlegende Template, welches dann auf jeder einzelnen Seite deiner Website erweitert wird.

Wir erstellen jetzt eine `base.html` -Datei in `blog/templates/blog/` :

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

Öffne sie im Code-Editor, kopiere alles von `post_list.html` und füge es wie folgt in die `base.html` -Datei ein:

`blog/templates/blog/base.html`

```
{% 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>
              <h2><a href="">{{ post.title }}</a></h2>
              <p>{{ post.text|linebreaksbr }}</p>
            </div>
          {% endfor %}
        </div>
      </div>
    </body>
  </html>
```

Dann ersetze in `base.html` den gesamten `<body>` (alles zwischen `<body>` und `</body>`) hiermit:

`blog/templates/blog/base.html`

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

Vielleicht hast du bemerkt, dass das alles von `{% for post in posts %}` bis `{% endfor %}` mit dem Folgenden ersetzt:

`blog/templates/blog/base.html`

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

Aber warum? Du hast gerade einen `block` erstellt! Du hast den Template-Tag `{% block %}` benutzt, um einen Bereich zu schaffen, der HTML aufnehmen kann. Dieses HTML kommt aus einem anderen Template, welches das Template hier (`base.html`) erweitert. Wir zeigen dir gleich, wie das geht.

Speichere nun die Datei `base.html` und öffne wieder `blog/templates/blog/post_list.html` im Code-Editor. Du löschest alles oberhalb von `{% for post in posts %}` und unterhalb von `{% endfor %}`. Wenn du das gemacht hast, sieht die Datei so aus:

`blog/templates/blog/post_list.html`

```
{% for post in posts %}
    <div class="post">
        <div class="date">
            {{ post.published_date }}
        </div>
        <h2><a href="">{{ post.title }}</a></h2>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endfor %}
```

Dieser Teil unseres Templates soll nun das Basis-Template erweitern. Wir müssen daher Block-Tags ergänzen!

Deine Block-Tags müssen mit den Tags in der `base.html`-Datei übereinstimmen. Stelle sicher, dass die Block-Tags den ganzen Code deines content-Blocks umschließen. Um das zu erreichen, umschließt du ihn mit `{% block content %}` und `{% endblock %}`. So hier:

`blog/templates/blog/post_list.html`

```
{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h2><a href="">{{ post.title }}</a></h2>
            <p>{{ post.text|linebreaksbr }}</p>
        </div>
    {% endfor %}
    {% endblock %}
```

Nur eine Sache noch. Wir müssen die beiden Templates miteinander verknüpfen. Darum geht es ja bei der Template-Erweiterung! Dafür ergänzen wir einen Tag für Erweiterung (`extends tag`) ganz oben in der Datei. Und zwar so:

`blog/templates/blog/post_list.html`

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

{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h2><a href="">{{ post.title }}</a></h2>
            <p>{{ post.text|linebreaksbr }}</p>
        </div>
    {% endfor %}
    {% endblock %}
```

Das war's! Speichere die Datei und probier aus, ob deine Website noch richtig funktioniert. :)

Wenn du den Fehler `TemplateDoesNotExist` erhältst, dann heißt das, dass es die Datei `blog/base.html` nicht gibt und dass `runserver` in der Konsole läuft. Halte ihn an (durch Drücken von `Ctrl+C` bzw. `Strg+C` – Control- und die C-Taste gleichzeitig) und starte ihn neu, indem du den Befehl `python manage.py runserver` ausführst.

# Erweitere deine Anwendung

Wir haben bereits die verschiedenen Schritte für die Erstellung unserer Website abgeschlossen: Wir wissen, wie man Models, URLs, Views und Templates schreibt. Wir wissen auch, wie wir unsere Website verschönern.

Zeit zu üben!

Das erste, was unser Blog braucht, ist eine Seite, auf der ein einzelner Blogpost dargestellt wird, oder?

Wir haben bereits ein `Post`-Model, deshalb brauchen wir nichts zur `models.py` hinzufügen.

## Erstelle eine Template-Verknüpfung

Wir beginnen damit, einen Link in der `blog/templates/blog/post_list.html`-Datei zu erstellen. Öffne sie im Code-Editor, und bisher sollte sie etwa so aussehen:

`blog/templates/blog/post_list.html`

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

{% block content %}
    {% for post in posts %}
        <div class="post">
            <div class="date">
                {{ post.published_date }}
            </div>
            <h2><a href="">{{ post.title }}</a></h2>
            <p>{{ post.text|linebreaksbr }}</p>
        </div>
    {% endfor %}
    {% endblock %}
```

Wir wollen einen Link vom Titel eines Posts in der Post-Liste zur Detailseite des jeweiligen Posts haben. Ändern wir `<h1><a href="">{{ post.title }}</a></h1>`, so dass es zu der Detailseite verlinkt:

`blog/templates/blog/post_list.html`

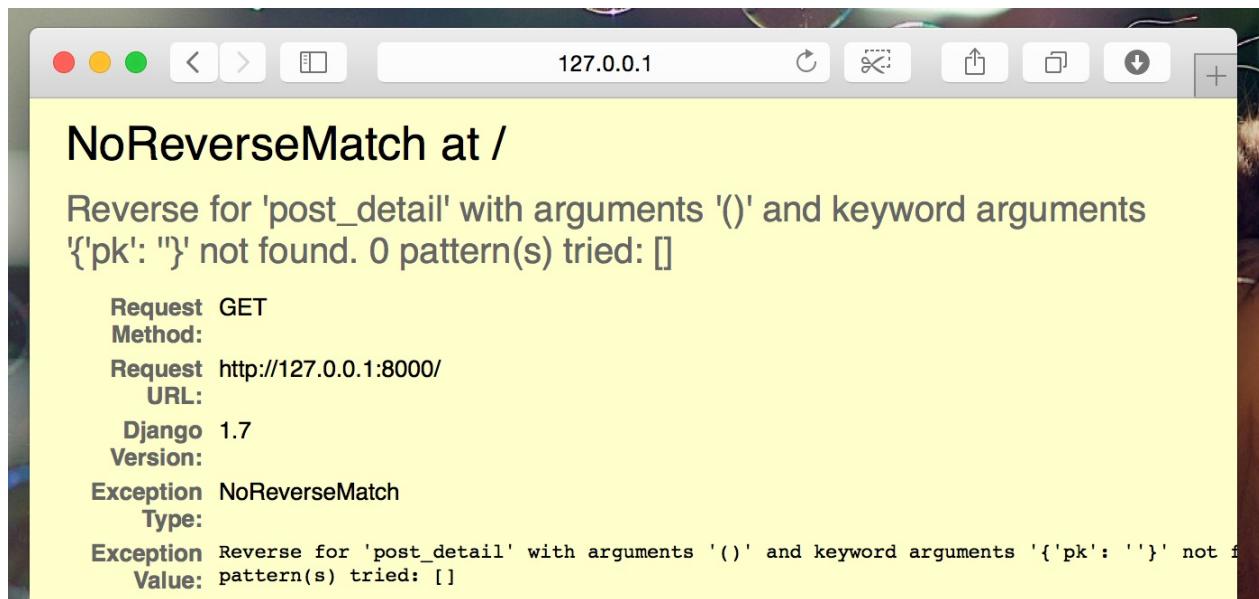
```
<h1><a href="{% url 'post_detail' pk=post.pk %}">{{ post.title }}</a></h1>
```

Es ist an der Zeit, das mysteriöse `{% url 'post_detail' pk=post.pk %}` zu erklären. Wie du dir wahrscheinlich schon denkst, bedeutet `{% %}`, dass wir Django-Template-Tags verwenden. Dieses Mal verwenden wir eines, das eine URL für uns erzeugen wird!

Der `post_detail`-Teil bedeutet, dass Django eine URL in `blog/urls.py` mit dem Namen `name=post_detail` erwartet.

Und was ist mit `pk=post.pk`? `pk` ist die Abkürzung für primary key (Primär-Schlüssel), ein eindeutiger Name für jeden Datensatz in einer Datenbank. Da wir keinen Primärschlüssel in unserem `Post`-Model angelegt haben, erstellt Django einen für uns (standardmäßig) ist das eine Zahl, die mit jedem Eintrag nach oben gezählt wird, z.B. 1, 2, 3) und fügt den Schlüssel in einem Feld namens `pk` zu jedem unserer Blogposts hinzu. Wir können auf den Primärschlüssel durch `post.pk` zugreifen, genauso wie auf andere Felder (`title`, `author`, usw.) in unserem `Post`-Objekt!

Wenn wir jetzt <http://127.0.0.1:8000/> aufrufen, erhalten wir einen Fehler (wie erwartet, da wir ja noch keine URL oder View für `post_detail` erstellt haben). Er wird so aussehen:



## Erstelle eine URL für die Post-Detailseite

Lass uns eine URL in `urls.py` für unsere `post_detail`-View erstellen!

Wir wollen, dass unsere erste Blogpost-Detailseite unter dieser **URL** angezeigt wird: <http://127.0.0.1:8000/post/1/>

Lass uns eine URL in der Datei `blog/urls.py` anlegen, um Django auf die View `post_detail` zu verweisen, welche dann den ganzen Blogpost anzeigen wird. Öffne die Datei `blog/urls.py` im Code-Editor und füge die Zeile `path('post/<int:pk>/', views.post_detail, name='post_detail'),` hinzu, so dass die Datei wie folgt aussieht:

`blog/urls.py`

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.post_list, name='post_list'),
    path('post/<int:pk>', views.post_detail, name='post_detail'),
]
```

Der Teil `post/<int:pk>` definiert ein URL-Muster – wir erklären es dir:

- `post/` heißt, dass die URL mit dem Wort **post** beginnen sollte, gefolgt von einem `/`. So weit, so gut.
- `<int:pk>` – Dieser Teil ist schwieriger. Er bedeutet, dass Django eine Ganzzahl (einen Integer-Wert) erwartet und diese in Form einer Variablen namens `pk` einer View weitergibt.
- `/` – dann brauchen wir vor dem Abschluss der URL wieder einen `/`.

Wenn du also `http://127.0.0.1:8000/post/5/` in deinen Browser eingibst, wird Django verstehen, dass du nach einer View suchst, die `post_detail` heißt, und wird der View die Information weitergeben, dass `pk` dabei `5` sein soll.

So, jetzt haben wir der `blog/urls.py` ein neues URL-Muster hinzugefügt! Lass uns die Seite <http://127.0.0.1:8000/> neu laden - Bumm! Der Server läuft wieder nicht mehr. Schau in die Konsole - wie erwartet gibt es noch einen anderen Fehler!

```

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

```

Erinnerst du dich, was der nächste Schritt ist? Eine View hinzufügen!

## Füge eine View hinzu

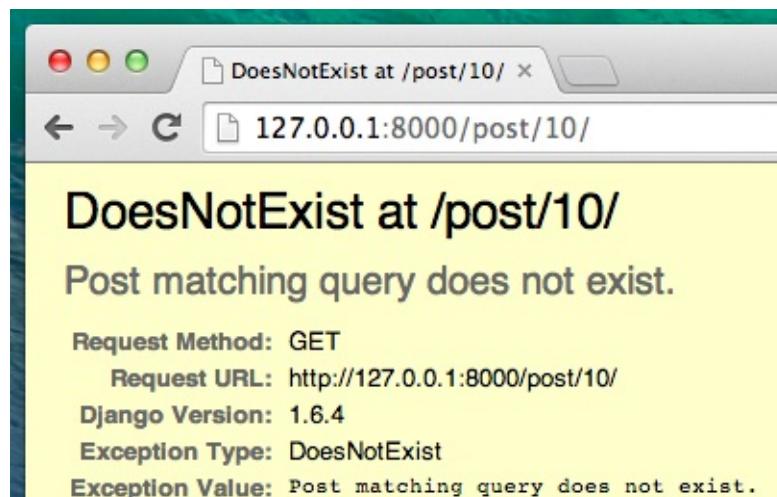
Dieses Mal bekommt unsere View den extra Parameter `pk`. Unsere View muss diesen entgegennehmen, richtig? Also definieren wir unsere Funktion als `def post_detail(request, pk)`. Beachte, dass wir genau den gleichen Variablennamen benutzen müssen, wie in den URLs festgelegt (`pk`). Diese Variable wegzulassen ist falsch und führt zu einem Fehler!

Jetzt benötigen wir also genau einen bestimmten Blogpost. Diesen finden wir, indem wir ein QuerySet folgendermaßen schreiben:

`blog/views.py`

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

Aber bei diesem Code gibt es ein Problem. Wenn es kein `Post`-Objekt mit diesem primary key (`pk`) gibt, bekommen wir einen super-hässlichen Fehler!



Das wollen wir nicht! Zum Glück stellt uns Django etwas zur Verfügung, das uns dieses Problem abnimmt:

`get_object_or_404`. Wenn es kein `Post`-Objekt mit einem gegebenen `pk` gibt, wird eine schöne Seite angezeigt, die sogenannte `Page Not Found 404`-Seite ("Seite nicht gefunden"-Seite).



Die gute Neuigkeit ist, dass du auch deine eigene `Page not found`-Seite erstellen und diese so hübsch gestalten kannst, wie du willst. Aber das ist jetzt gerade nicht so wichtig, deshalb überspringen wir das.

Okay, es wird Zeit, die `View` zu unserer `views.py`-Datei hinzuzufügen!

In `blog/urls.py` haben wir eine URL-Regel namens `post_detail` erstellt, die auf eine View namens `views.post_detail` verweist. Das heißt, dass Django eine View-Funktion erwartet, die `post_detail` heißt und in `blog/views.py` angelegt wurde.

Wir sollten also `blog/views.py` im Code-Editor öffnen und den folgenden Code zu den anderen `from` Zeilen hinzufügen:

`blog/views.py`

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

Und am Ende der Datei werden wir unsere View-Funktion ergänzen:

`blog/views.py`

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

Super. Lass uns nun <http://127.0.0.1:8000/> neu laden.

The screenshot shows a Mac OS X desktop environment with a browser window titled "Django Girls Blog". The URL in the address bar is "127.0.0.1:8000". The main content area has a yellow header with the text "Django Girls". Below it, there is a post with the title "Nulla facilisi" in a large, stylized font. The post content starts with "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras bibendum sapien i" followed by several ellipses (...). Another post below it has the title "Fusce vehicula feugiat augue eget consectetur" in a similar large, stylized font.

Es hat funktioniert! Aber was passiert, wenn du auf den Link im Blog-Titel klickst?

The screenshot shows a Mac OS X desktop environment with a browser window titled "TemplateDoesNotExist at /". The URL in the address bar is "127.0.0.1:8000/post/3/". The main content area displays an error message: "TemplateDoesNotExist at /post/3/" followed by "blog/post\_detail.html". Below this, there is a detailed error report:  
Request Method: GET  
Request URL: http://127.0.0.1:8000/post/3/  
Django Version: 1.6.4  
Exception Type: TemplateDoesNotExist  
Exception Value: blog/post\_detail.html

Oh nein! Ein anderer Fehler! Aber wir wissen ja schon, wie wir mit diesem umgehen, oder? Wir müssen ein Template hinzufügen!

## Erstelle ein Template für die Post-Detailseite

Wir erstellen eine Datei in `blog/templates/blog` mit dem Namen `post_detail.html` und öffnen sie im Code-Editor.

Das sieht dann so aus:

blog/templates/blog/post\_detail.html

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

{% block content %}
    <div class="post">
        {% if post.published_date %}
            <div class="date">
                {{ post.published_date }}
            </div>
        {% endif %}
        <h2>{{ post.title }}</h2>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endblock %}
```

Wir erweitern wieder `base.html`. Im `content`-Block wollen wir das Publikationsdatum eines Posts (`published_date`), falls es existiert, anzeigen und auch den Titel und den Text. Aber wir müssen noch ein paar wichtige Dinge klären, oder?

`{% if ... %} ... {% endif %}` ist ein Template-Tag, das wir benutzen können, wenn wir etwas überprüfen möchten. (Erinnerst du dich an `if ... else ...` vom Kapitel **Einführung in Python**?) In diesem Fall hier wollen wir prüfen, ob das `published_date`-Feld eines Post-Objektes nicht leer ist.

OK, aktualisieren wir unsere Seite und sehen, ob `TemplateDoesNotExist` jetzt weg ist.



Yay! Es funktioniert!

## Veröffentlichen!

Es wäre schön zu sehen, ob deine Website noch auf PythonAnywhere funktioniert, richtig? Lass sie uns erneut veröffentlichen.

command-line

```
$ git status  
$ git add --all .  
$ git status  
$ git commit -m "View und Template für Blogpost-Details sowie CSS für die Website hinzugefügt"  
$ git push
```

Dann führe Folgendes in der [PythonAnywhere-Bash-Konsole](#) aus:

PythonAnywhere command-line

```
$ cd ~/<your-pythonanywhere-username>.pythonanywhere.com  
$ git pull  
[...]
```

(Denk daran, `<your-pythonanywhere-username>` durch deinen tatsächlichen PythonAnywhere-Benutzernamen zu ersetzen - ohne die spitzen Klammern).

## Die statischen Dateien auf dem Server aktualisieren

Server wie PythonAnywhere behandeln statische Dateien ("static files", z.B. CSS-Dateien) anders als Python-Dateien, weil statische Dateien noch optimiert und dadurch dann schneller geladen werden können. Deswegen müssen wir, nachdem wir Änderungen an den CSS Dateien vorgenommen haben, einen zusätzlichen Befehl auf dem Server ausführen, um diese Dateien zu aktualisieren. Der Befehl heißt `collectstatic`.

Aktiviere also deine virtuelle Umgebung, wenn sie nicht vom letzten Mal noch aktiv ist (PythonAnywhere benutzt dazu das Kommando `workon`, das ist ähnlich wie `source myenv/bin/activate`, das du auf deinem Computer verwendest):

PythonAnywhere command-line

```
$ workon <your-pythonanywhere-username>.pythonanywhere.com  
(ola.pythonanywhere.com)$ python manage.py collectstatic  
[...]
```

Der `manage.py collectstatic` Befehl ist ein bisschen wie `manage.py migrate`. Wir machen ein paar Änderungen in unserem Code und dann sagen wir Django, dass es diese übernehmen (*apply*) soll, entweder zu der servereigenen Sammlung von statischen Dateien oder in die Datenbank.

Auf jeden Fall sind wir nun soweit, dass wir [auf die Seite "Web"](#) wechseln können (mittels dem Menü-Knopf in der Ecke oben rechts) und **Reload** klicken können. Schau dir dann die Seite <https://yourname.pythonanywhere.com> an, um das Ergebnis zu sehen.

Und das sollte es sein! Herzlichen Glückwunsch :)

# Django-Formulare

Als Letztes möchten wir auf unserer Website noch die Möglichkeit haben, Blogposts hinzuzufügen und zu editieren. Die Django `admin`-Oberfläche ist cool, aber eher schwierig anzupassen und hübsch zu machen. Mit Formularen, `forms`, haben wir die absolute Kontrolle über unser Interface - wir können fast alles machen, was man sich vorstellen kann!

Das Gute an Django-Forms ist, dass man sie entweder vollständig selbst definieren oder eine `ModelForm` erstellen kann, welche den Inhalt des Formulars in das Model speichert.

Genau das wollen wir jetzt machen: Wir erstellen ein Formular für unser `Post`-Model.

So wie die anderen wichtigen Django-Komponenten haben auch die Forms ihre eigene Datei: `forms.py`.

Wir erstellen nun eine Datei mit diesem Namen im `blog`-Verzeichnis.

```
blog
└── forms.py
```

So, jetzt lass uns diese im Code-Editor öffnen und folgenden Code hinzufügen:

`blog/forms.py`

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

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

Zuerst müssen wir die Django-Forms importieren (`from django import forms`) und auch unser `Post`-Model (`from .models import Post`).

Wie du wahrscheinlich schon vermutet hast, `PostForm` ist der Name unseres Formulars. Wir müssen Django mitteilen, dass unser Formular ein `ModelForm` ist (so kann Django ein bisschen für uns zaubern) - `forms.ModelForm` ist dafür verantwortlich.

Als Nächstes sehen wir uns `class Meta` an, damit sagen wir Django, welches Model benutzt werden soll, um das Formular zu erstellen (`model = Post`).

Nun können wir bestimmen, welche(s) Feld(er) unser Formular besitzen soll. Wir wollen hier nur den `title` und `text` sichtbar machen - der `author` sollte die Person sein, die gerade eingeloggt ist (Du!) und `created_date` sollte automatisch generiert werden, wenn der Post erstellt wird (also im Code). Stimmt's?

Und das war's schon! Jetzt müssen wir das Formular nur noch in einem `view` benutzen und im Template darstellen.

Also erstellen wir hier auch wieder einen Link auf die Seite, eine URL, eine View und ein Template.

## Link auf eine Seite mit dem Formular

Jetzt ist es an der Zeit, `blog/templates/blog/base.html` im Code-Editor zu öffnen. Wir fügen einen Link in `div` hinzu mit dem Namen `page-header`:

blog/templates/blog/base.html

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

Beachte, dass wir unsere neue View `post_new` nennen wollen. Die Klasse `"glyphicon glyphicon-plus"` wird durch das verwendete Bootstrap-Theme zur Verfügung gestellt und wird ein Pluszeichen anzeigen.

Nach dem Hinzufügen der Zeile sieht deine HTML-Datei so aus:

blog/templates/blog/base.html

```
{% 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>
```

Nach dem Speichern und Neuladen von <http://127.0.0.1:8000> solltest du den bereits bekannten `NoReverseMatch`-Fehler sehen. Ist dem so? Gut!

## URL

Wir öffnen `blog/urls.py` im Code-Editor und fügen eine Zeile hinzu:

blog/urls.py

```
path('post/new', views.post_new, name='post_new'),
```

Der finale Code sieht dann so aus:

blog/urls.py

```
from django.urls import path
from . import views
urlpatterns = [
    path('', views.post_list, name='post_list'),
    path('post/<int:pk>', views.post_detail, name='post_detail'),
    path('post/new/', views.post_new, name='post_new'),
]
```

Nach dem Neuladen der Site sehen wir einen `AttributeError`, weil wir noch keine `post_new`-View eingefügt haben. Fügen wir sie gleich hinzu!

## Die post\_new-View

Jetzt wird es Zeit, die Datei `blog/views.py` im Code-Editor zu öffnen und die folgenden Zeilen zu den anderen `from`-Zeilen hinzuzufügen:

`blog/views.py`

```
from .forms import PostForm
```

Und dann unsere View:

`blog/views.py`

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

Um ein neues `PostForm` zu erstellen, rufen wir `PostForm()` auf und übergeben es an das Template. Wir kommen gleich nochmal zu dem View zurück, aber jetzt erstellen wir schnell ein Template für das Form.

## Template

Wir müssen eine Datei `post_edit.html` im Verzeichnis `blog/templates/blog` erstellen und im Code-Editor öffnen. Damit ein Formular funktioniert, benötigen wir einige Dinge:

- Wir müssen das Formular anzeigen. Wir können das zum Beispiel mit einem einfachen `` tun.
- Die Zeile oben muss von einem HTML-Formular-Tag eingeschlossen werden `<form method="POST">...</form>`.
- Wir benötigen einen `Save`-Button. Wir erstellen diesen mit einem HTML-Button: `<button type="submit">Save</button>`.
- Und schließlich fügen wir nach dem öffnenden `<form ...>` Tag `{% csrf_token %}` hinzu. Das ist sehr wichtig, da es deine Formulare sicher macht! Wenn du diesen Teil vergisst, wird sich Django beim Speichern des Formulars beschweren.



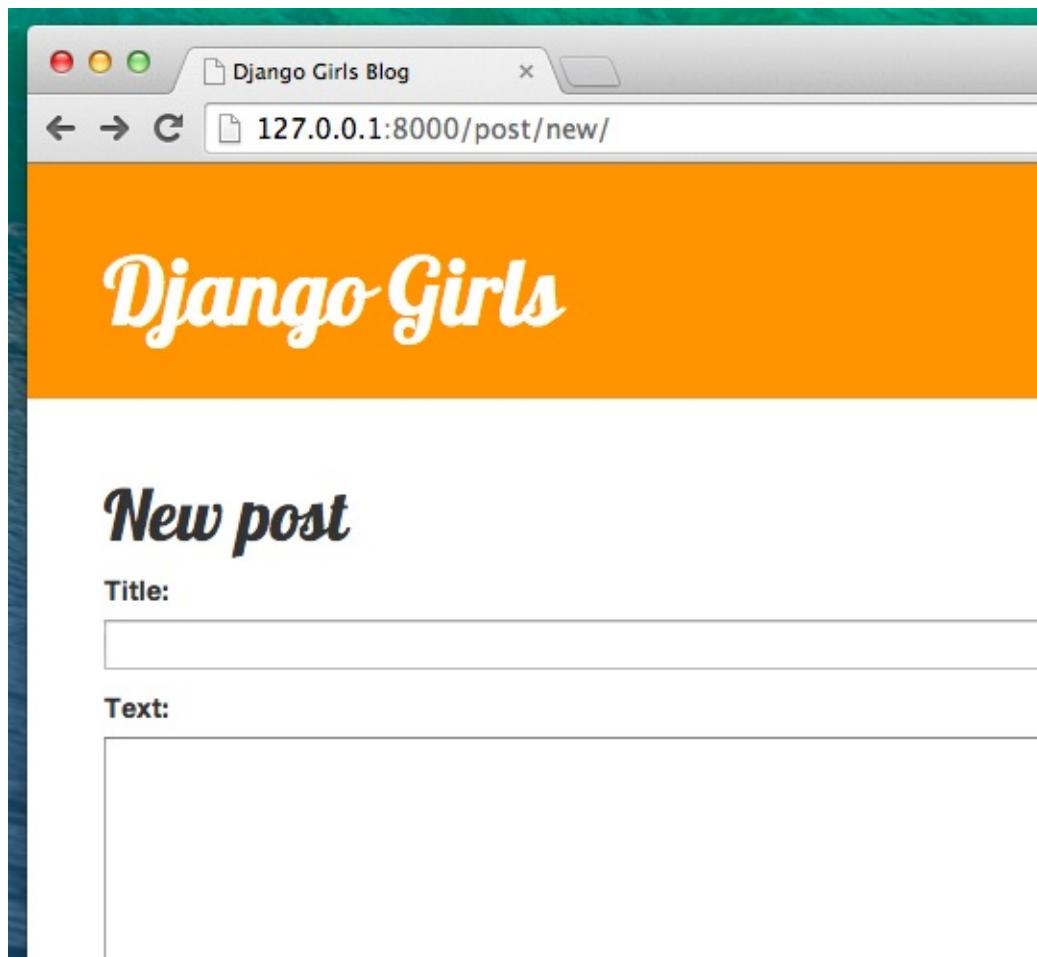
Ok, also schauen wir mal, wie der HTML-Code in `post_edit.html` aussehen sollte:

`blog/templates/blog/post_edit.html`

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

{% block content %}
    <h2>New post</h2>
    <form method="POST" class="post-form">{% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
    </form>
{% endblock %}
```

So, jetzt aktualisieren wir die Seite! Yay! Das Formular wird angezeigt!



Aber Moment! Wenn du in das `title` - oder `text` -Feld etwas eintippst und versuchst es zu speichern - was wird wohl passieren?

Nichts! Wir landen wieder auf der selben Seite und unser Text ist verschwunden... und kein neuer Post wurde hinzugefügt. Was lief denn hier schief?

Die Antwort ist: nichts. Wir müssen einfach noch etwas mehr Arbeit in unsere View stecken.

## Speichern des Formulars

Öffne `blog/views.py` erneut im Code-Editor. Derzeit ist alles, was wir in der View `post_new` haben, das hier:

`blog/views.py`

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

Wenn wir das Formular übermitteln, werden wir zur selben Ansicht weitergeleitet, aber dieses Mal haben wir mehr Daten in `request`, genauer in `request.POST` (der Name hat nichts zu tun mit einem "Blogpost", sondern damit, dass wir Daten "posten"). Erinnerst du dich daran, dass in der HTML-Datei unsere `<form>` Definition die Variable `method="POST"` hatte? Alle Felder aus dem Formular sind jetzt in `request.POST`. Du solltest `POST` nicht umbenennen (der einzige andere gültige Wert für `method` ist `GET`, wir wollen hier jetzt aber nicht auf den Unterschied eingehen).

Somit müssen wir uns in unserer View mit zwei verschiedenen Situationen befassen: erstens, wenn wir das erste Mal auf die Seite zugreifen und ein leeres Formular wollen und zweitens, wenn wir zur View mit allen soeben ausgefüllten Formular-Daten zurück wollen. Wir müssen also eine Bedingung hinzufügen (dafür verwenden wir `if`):

blog/views.py

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

Es wird Zeit, die Lücken zu füllen `[...]`. Falls die Methode `POST` ist, wollen wir das `PostForm` mit Daten vom Formular erstellen. Oder? Das machen wir folgendermaßen:

blog/views.py

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

Als Nächstes müssen wir testen, ob das Formular korrekt ist (alle benötigten Felder sind ausgefüllt und keine ungültigen Werte werden gespeichert). Wir tun das mit `form.is_valid()`.

Wir überprüfen also, ob das Formular gültig ist und wenn ja, können wir es speichern!

blog/views.py

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

Im Grunde passieren hier zwei Dinge: Wir speichern das Formular mit `form.save` und wir fügen einen Autor hinzu (da es bislang kein `author` Feld in der `PostForm` gab und dieses Feld notwendig ist). `commit=False` bedeutet, dass wir das `Post` Model noch nicht speichern wollen - wir wollen erst noch den Autor hinzufügen. Meistens wirst du `form.save()` ohne `commit=False` benutzen, aber in diesem Fall müssen wir es so tun. `post.save()` wird die Änderungen sichern (den Autor hinzufügen) und ein neuer Blogpost wurde erstellt!

Wäre es nicht grossartig, wenn wir direkt zu der `post_detail` Seite des neu erzeugten Blogposts gehen könnten? Um dies zu tun, benötigen wir noch einen zusätzlichen Import:

blog/views.py

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

Füge dies direkt am Anfang der Datei hinzu. Jetzt können wir endlich sagen: "Gehe zu der `post_detail` Seite unseres neu erstellten Posts":

blog/views.py

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

`post_detail` ist der Name unserer View, zu der wir springen wollen. Erinnerst du dich, dass diese View einen `pk` benötigt? Um diesen an die View weiterzugeben, benutzen wir `pk=post.pk`, wobei `post` unser neu erstellter Blogpost ist!

Ok, wir haben jetzt eine ganze Menge geredet, aber du willst bestimmt sehen, wie die gesamte View aussieht, richtig?

blog/views.py

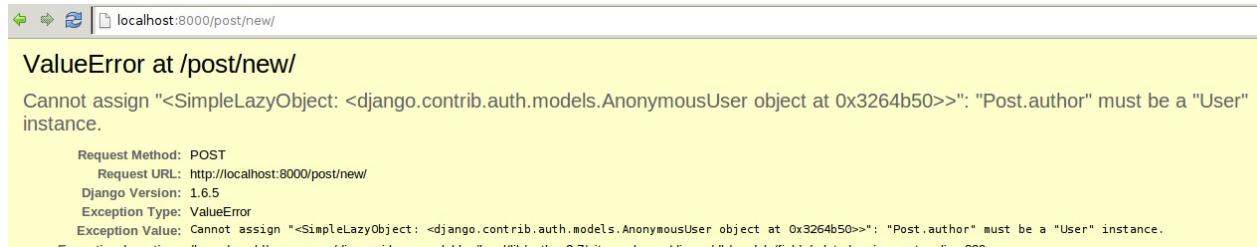
```
def post_new(request):
    if request.method == "POST":
        form = PostForm(request.POST)
        if form.is_valid():
            post = form.save(commit=False)
            post.author = request.user
            post.published_date = timezone.now()
            post.save()
            return redirect('post_detail',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

Schauen wir mal, ob es funktioniert. Gehe zur Seite <http://127.0.0.1:8000/post/new/>, füge einen title und text hinzu und speichere es...voilà! Der neue Blogpost wird hinzugefügt und wir werden auf die post\_detail Seite umgeleitet!

Du hast vielleicht bemerkt, dass wir das Veröffentlichungsdatum festlegen, bevor wir den Post veröffentlichen. Später werden wir einen *publish button* in **Django Girls Tutorial: Extensions** einführen.

Das ist genial!

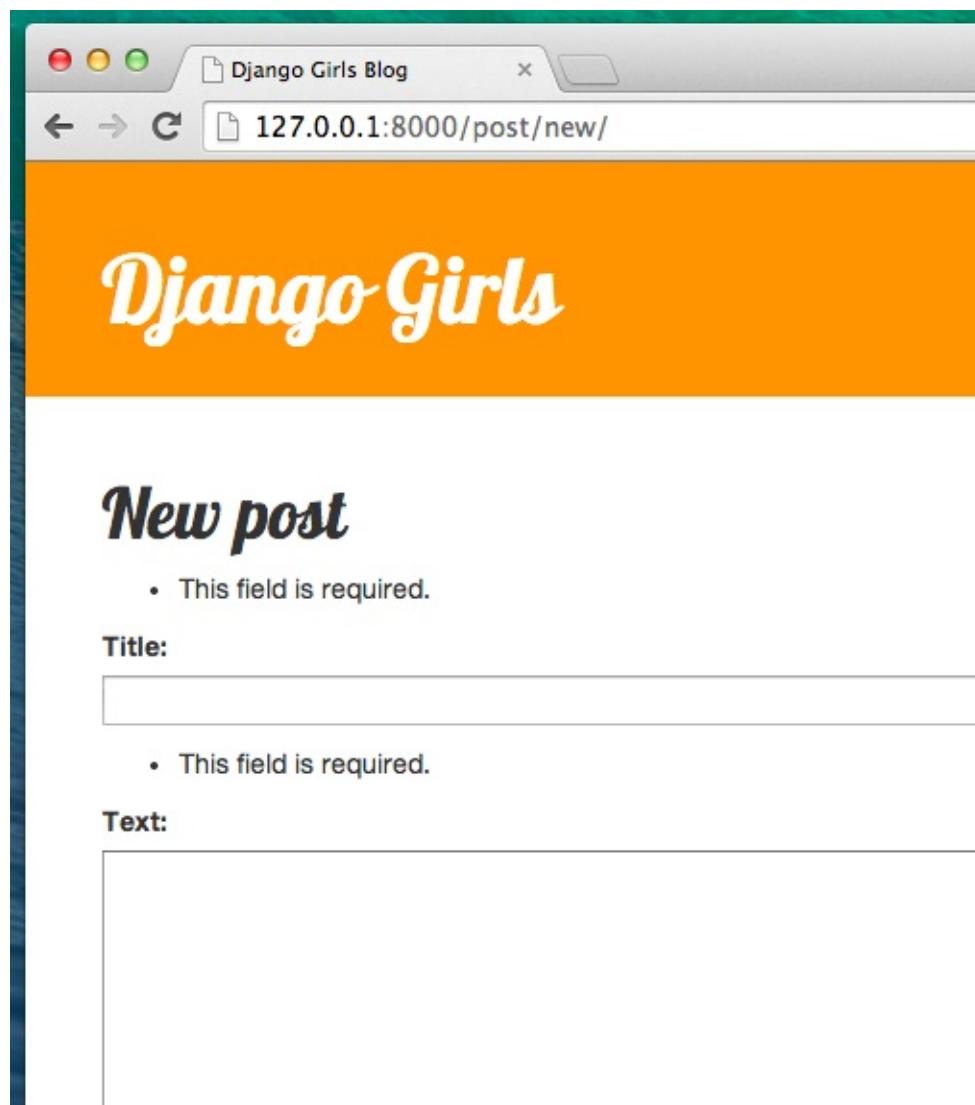
Da wir vor Kurzem das Django-Admin-Interface benutzt haben, denkt das System, dass wir noch angemeldet sind. Es gibt einige Situationen, welche dazu führen können, dass wir ausgeloggt werden (Schließen des Browsers, Neustarten der Datenbank etc). Wenn du feststellst, dass du bei dem Erstellen von Posts Fehlermeldungen bekommst, die auf nicht angemeldete Nutzer zurückzuführen sind, dann gehe zur Admin Seite <http://127.0.0.1:8000/admin> und logge dich erneut ein. Dies wird das Problem vorübergehend lösen. Es gibt eine permanente Lösung dafür, die im Kapitel **Homework: add security to your website!** nach dem Haupttutorial auf dich wartet.



## Formularvalidierung

Jetzt zeigen wir dir, wie cool Django-Formulare sind. Ein Blogpost muss title - und text -Felder besitzen. In unserem Post -Model haben wir (im Gegensatz zu dem published\_date ) nicht festgelegt, dass diese Felder nicht benötigt werden, also nimmt Django standardmäßig an, dass sie definiert werden.

Versuch, das Formular ohne title und text zu speichern. Rate, was passieren wird!



Django kümmert sich darum sicherzustellen, dass alle Felder in unserem Formular richtig sind. Ist das nicht großartig?

## Formular bearbeiten

Jetzt wissen wir, wie ein neues Formular hinzugefügt wird. Aber was ist, wenn wir ein bereits bestehendes bearbeiten wollen? Das funktioniert so ähnlich wie das, was wir gerade getan haben. Lass uns schnell ein paar wichtige Dinge kreieren. (Falls du etwas nicht verstehst, solltest du deinen Coach fragen oder in den vorherigen Kapiteln nachschlagen, da wir all die Schritte bereits behandelt haben.)

Öffne `blog/templates/blog/post_detail.html` im Code-Editor und füge folgende Zeile hinzu:

`blog/templates/blog/post_detail.html`

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

damit die Vorlage so aussieht:

`blog/templates/blog/post_detail.html`

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

{% block content %}
    <div class="post">
        {% if post.published_date %}
            <div class="date">
                {{ post.published_date }}
            </div>
        {% endif %}
        <a class="btn btn-default" href="{% url 'post_edit' pk=post.pk %}"><span class="glyphicon glyphicon-pencil"></span></a>
        <h2>{{ post.title }}</h2>
        <p>{{ post.text|linebreaksbr }}</p>
    </div>
{% endblock %}
```

Öffne `blog/urls.py` im Code-Editor und fügen diese Zeile hinzu:

`blog/views.py`

```
path('post/<int:pk>/edit/', views.post_edit, name='post_edit'),
```

Wir werden die Vorlage `blog/templates/blog/post_edit.html` wiederverwenden, daher ist das einzig Fehlende eine neue View.

Öffne `blog/views.py` im Code-Editor und füge ganz am Ende der Datei Folgendes hinzu:

`blog/views.py`

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

Sieht genauso aus wie unsere `post_new`-View, oder? Aber nicht ganz. Zum einen übergeben wir zusätzliche `pk` Parameter von urls. Und: Wir bekommen das `Post`-Model, welches wir bearbeiten wollen, mit `get_object_or_404(Post, pk=pk)` und wenn wir dann ein Formular erstellen, übergeben wir `post` als `instance`, wenn wir das Formular speichern...

`blog/views.py`

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

als auch, wenn wir ein Formular mit `post` zum Editieren öffnen:

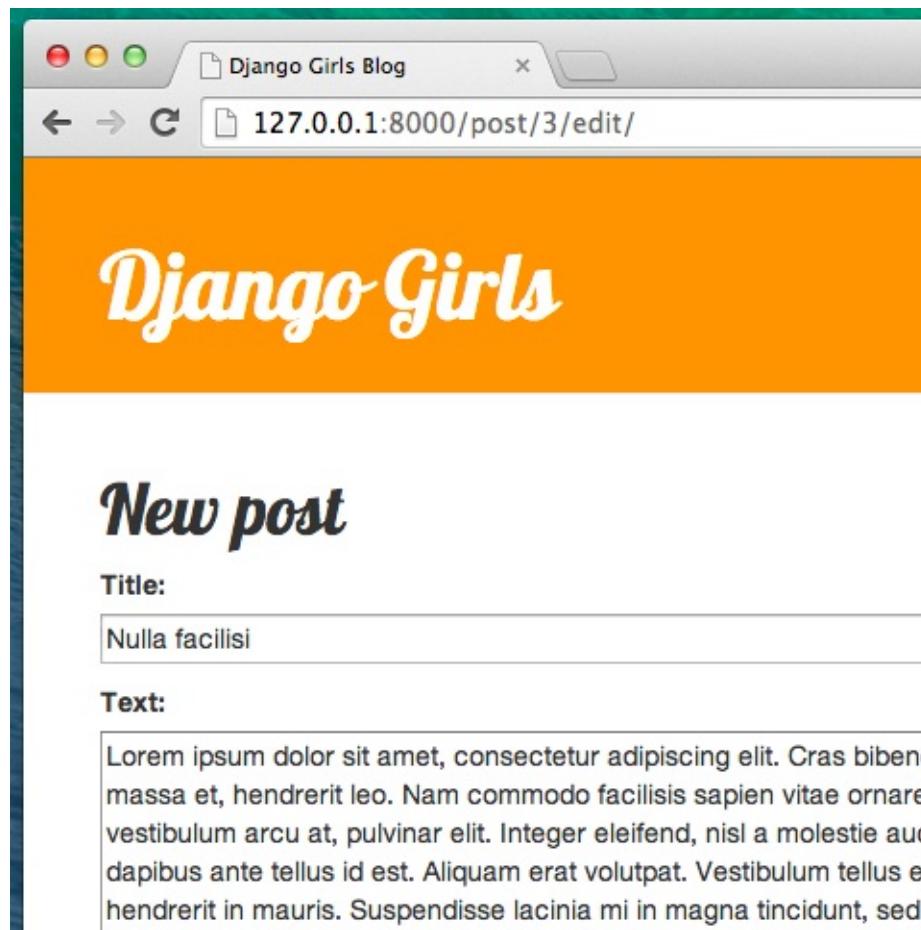
`blog/views.py`

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

Ok, lass uns mal schauen, ob das funktioniert! Geh auf die `post_detail`-Seite. Dort sollte sich ein Editier-Button in der oberen rechten Ecke befinden:



Wenn du darauf klickst, siehst du das Formular mit unserem Blogpost:



Probier doch einmal, den Titel oder den Text zu ändern und die Änderungen zu speichern!

Herzlichen Glückwunsch! Deine Anwendung nimmt immer mehr Gestalt an!

Falls du mehr Informationen über Django-Formulare benötigst, solltest du die offizielle Dokumentation lesen:

<https://docs.djangoproject.com/en/2.0/topics/forms/>.

## Sicherheit

Neue Posts mit einem Linkklick zu erstellen, ist großartig! Aber im Moment ist jeder, der deine Seite besucht in der Lage, einen neuen Blogpost zu veröffentlichen und das ist etwas, was du garantiert nicht willst. Lass es uns so machen, dass der Button für dich angezeigt wird, aber für niemanden sonst.

Öffne die Datei `blog/templates/blog/base.html` im Code-Editor, finde darin unseren `page-header` `div` und das Anchor-Tag, welches du zuvor eingefügt hast. Es sollte so aussehen:

`blog/templates/blog/base.html`

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

Wir fügen ein weiteres `{% if %}`-Tag ein, was dafür sorgt, dass der Link nur für angemeldete Nutzer angezeigt wird. Im Moment bist das also nur du! Ändere den `<a>`-Tag zu Folgendem:

`blog/templates/blog/base.html`

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

Dieses `{% if %}` sorgt dafür, dass der Link nur zu dem Browser geschickt wird, wenn der anfragende Nutzer auch angemeldet ist. Das verhindert das Erzeugen neuer Posts nicht komplett, ist aber ein sehr guter erster Schritt. In der Erweiterungslektion kümmern wir uns ausgiebiger um Sicherheit.

Erinnerst du dich an den Editier-Button, den wir gerade zu unserer Seite hinzugefügt haben? Wir wollen dort dieselbe Anpassung machen, damit andere Leute keine existierenden Posts verändern können.

Öffne `blog/templates/blog/post_detail.html` im Code-Editor und finde folgende Zeile:

`blog/templates/blog/post_detail.html`

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

Ändere es wie folgt:

`blog/templates/blog/post_detail.html`

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

Da du wahrscheinlich angemeldet bist, wirst du beim Refresh der Seite keinen Veränderung feststellen. Lade jedoch die Seite in einem anderen Browser oder einem Inkognito-Fenster ("In Private" im Windows Edge) und du wirst sehen, dass dieser Link nicht auftaucht und das Icon ebenfalls nicht angezeigt wird!

## Eins noch: Zeit für das Deployment!

Mal sehen, ob das alles auch auf PythonAnywhere funktioniert. Zeit für ein weiteres Deployment!

- Commite als Erstes deinen neuen Code und schiebe ihn auf GitHub:

command-line

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

- Dann führe Folgendes in der [PythonAnywhere Bash-Konsole](#) aus:

PythonAnywhere command-line

```
$ cd ~/<your-pythonanywhere-username>.pythonanywhere.com  
$ git pull  
[...]
```

(Denke daran, `<your-pythonanywhere-username>` durch deinen PythonAnywhere-Benutzernamen zu ersetzen - ohne spitze Klammern).

- Gehe schliesslich noch rüber [auf die Seite "Web"](#) (benutze den Menü-Knopf in der rechten oberen Ecke der Konsole) und klicke **Reload**. Lade deinen Blog <https://yourname.pythonanywhere.com> neu, um die Änderungen zu sehen.

Und das war's! Glückwunsch :)

# Wie geht es weiter?

Herzlichen Glückwunsch! **Du bist der Hammer!** Wir sind echt stolz! <3

## Was jetzt?

Mach eine Pause und entspanne! Du hast gerade etwas wirklich Großes geleistet.

Und dann folge Django Girls doch auf [Facebook](#) oder [Twitter](#), um auf dem Laufenden zu bleiben.

## Könnt ihr weiteres Lernmaterial empfehlen?

Ja! Es gibt *sehr* viele Online-Ressourcen zum Erlernen aller möglichen Programmierfähigkeiten – es kann ziemlich entmutigend sein, herauszufinden, was man als Nächstes machen sollte, aber wir helfen dir dabei. Was auch immer du für Interessen hattest, bevor du zu Django Girls gekommen bist und was auch immer du für Interessen während des Tutorials entwickelt hast, hier sind einige kostenfreie Ressourcen (oder solche mit größeren kostenlosen Teilen), die du nutzen kannst, um auf deinem Weg voranzukommen.

## Django

- Unser anderes Buch, [Django Girls Tutorial: Erweiterungen](#)
- [Das offizielle Django-Tutorial](#)
- [Video-Lektionen "Getting Started With Django"](#)

## HTML, CSS und JavaScript

- [Web-Development-Kurs auf Codecademy](#)
- [freeCodeCamp](#)

## Python

- [Python-Kurs auf Codecademy](#)
- [Googles Python-Kurs](#)
- [Learn Python The Hard Way](#) – die ersten Übungen sind kostenlos
- [New Coder Tutorials](#) – eine Vielzahl von praktischen Beispielen, wie Python verwendet werden kann
- [edX](#) – die meisten Kurse kannst du kostenfrei testen, aber wenn du ein Zertifikat oder Credits für eine Weiterbildung erhalten willst, dann kostet das etwas
- [Courseras Python-Spezialisierung](#) – einige Video-Lektionen können kostenlos getestet werden und du kannst ein Coursera-Zertifikat erlangen, wenn du diesen Kurs belegst
- [Python for Everybody](#) - eine kostenlose und offene Version der Coursera-Python-Spezialisierung

## Mit Daten arbeiten

- [Der Data-Science-Kurs von Codecademy](#)
- [edX](#) – die meisten Kurse kannst du kostenfrei testen, aber wenn du ein Zertifikat oder Credits für eine Weiterbildung erhalten willst, dann kostet das etwas
- [Dataquest](#) – die ersten 30 "Missionen" sind frei

Wir sind gespannt darauf, was du als Nächstes baust!