

Tutorial

django *girls*

Tabela de conteúdos

Introdução	1.1
Como funciona a internet?	1.2
Introdução à linha de comando	1.3
Instalação do Python	1.4
Editor de Código	1.5
Introdução ao Python	1.6
O que é Django?	1.7
Instalação do Django	1.8
Criando um projeto	1.9
Modelos do Django	1.10
Administração	1.11
Implantação!	1.12
Urls	1.13
Views - hora de criar!	1.14
Introdução a HTML	1.15
Django ORM (Querysets)	1.16
Dados dinâmicos nos templates	1.17
Templates	1.18
CSS - Deixe mais bonito	1.19
Estendendo os templates	1.20
Amplie sua aplicação	1.21
Formulários	1.22
Domínio	1.23
O que vem depois?	1.24

Tutorial Django Girls

gitter [join chat](#)

Este trabalho é licenciado sob a licença Creative Commons Attribution-ShareAlike 4.0. Para ver uma cópia desta licença, visite <https://creativecommons.org/licenses/by-sa/4.0/>

Translation

This tutorial has been translated from English to Portuguese by a group of awesome volunteers.

Special thanks for help goes out to:

Adailton do Nascimento Adam Victor Nazareth Brandizzi *Antonio Luis* Annanda Sousa *Adjamilton Junior* Bernardo Fontes *Camilla Achutti* Carla Suarez *Cleiton Lima* clemente.jnr *danieltex* Eric Hideki *Flavio Barros* Fabio C. Barrionuevo da Luz *ffabiorj* Gabriela Cavalcante da Silva *Leandro Silva Araujo* Lucas Magnum *1pedro* Joao Luiz Lorencetti *Katyanna Moura* Kleber C Pinheiro *Leandro Barbosa* Leonardo Alves dos Santos *Marcel Ribeiro Dantas* joepreludian *Rafael Biagioni de Fazio* Pablo Palacios *Paulo Alem* Raony Guimares Correo do Carmo Lisboa Cardenas *Vivian Macedo* Willie Lawrence *Ricardo Manhães Savii Wow! <3 <3

Introdução

Você já sentiu como se o mundo fosse cada vez mais sobre tecnologia e que de alguma forma você tenha ficado pra trás? Você já imaginou como seria criar um website mas nunca teve motivação suficiente para começar? Você já pensou que o mundo do software é complicado demais até para tentar fazer alguma coisa sozinho?

Bem, nós temos boas notícias para você! Programação não é tão difícil quanto parece e nós queremos te mostrar o quão divertido pode ser.

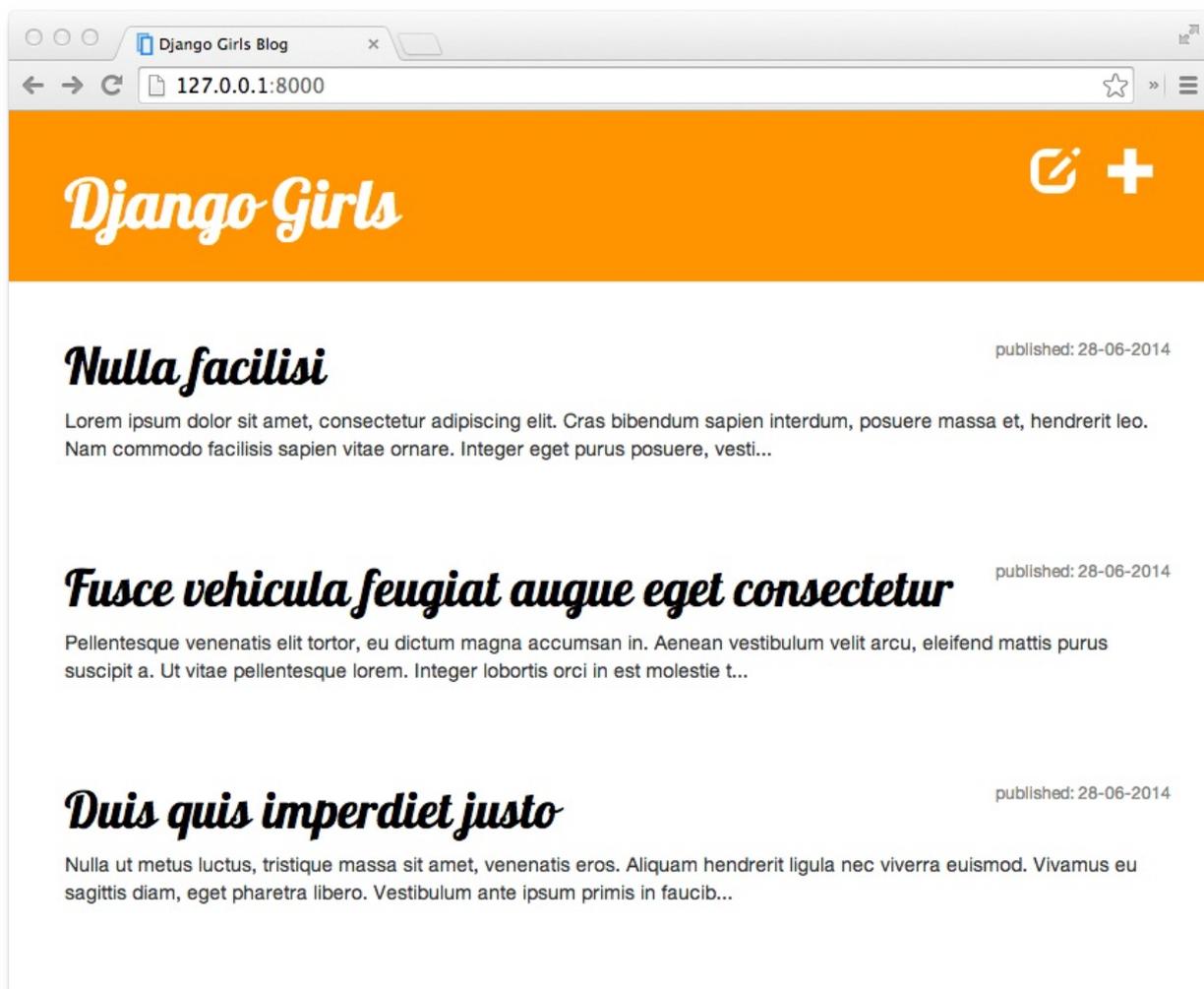
Este tutorial não irá te transformar magicamente em um programador. Se você quer ser bom nisso precisa de meses ou até mesmo anos de treino e prática. Mas nós queremos te mostrar que a programação ou criação de websites não é tão complicada quanto parece. Nós tentaremos explicar os diferentes pedaços tão bem quanto pudermos, tal que você não se sinta intimidado pela tecnologia.

Nós esperamos conseguir fazer você amar a tecnologia tanto quanto nós amamos!

O que você irá aprender durante o tutorial?

Quando você tiver terminado o tutorial você terá uma aplicação web simples e funcional: seu próprio blog. Nós vamos mostrar como colocá-lo online, para que outros vejam seu trabalho!

Ele se parecerá (mais ou menos) com isso:



Se você seguir o tutorial por conta própria e não tiver um treinador para ajudar em caso de qualquer problema, nós temos um chat para você: [gitter](#) [join chat](#) 1. Nós pedimos aos nossos treinadores e participantes anteriores para acessarem lá de tempos em tempos e ajudarem outros com o tutorial! Não tenha medo de fazer sua pergunta lá!

OK, [vamos pelo começo...](#)

Sobre o tutorial e contribuições

Este tutorial é mantido por [DjangoGirls](#). Se você encontrar quaisquer erros ou quiser atualizar o tutorial, por favor [siga as orientações de contribuição](#).

Gostaria de nos ajudar a traduzir o tutorial para outros idiomas?

no momento, as traduções estão sendo mantidas na plataforma [crowdin.com](#) em:

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

Se o seu idioma não está listado no [crowdin](#), por favor [open a new issue](#) informando o idioma para que possamos adicioná-lo.

Como funciona a internet

Este capítulo é inspirado na palestra "Como a Internet funciona" de Jessica McKellar (<http://web.mit.edu/jesstess/www/>).

Apostamos que você usa a Internet todos os dias. Mas você sabe realmente o que acontece quando você digita um endereço como <https://djangogirls.org> em seu navegador e pressiona 'Enter'?

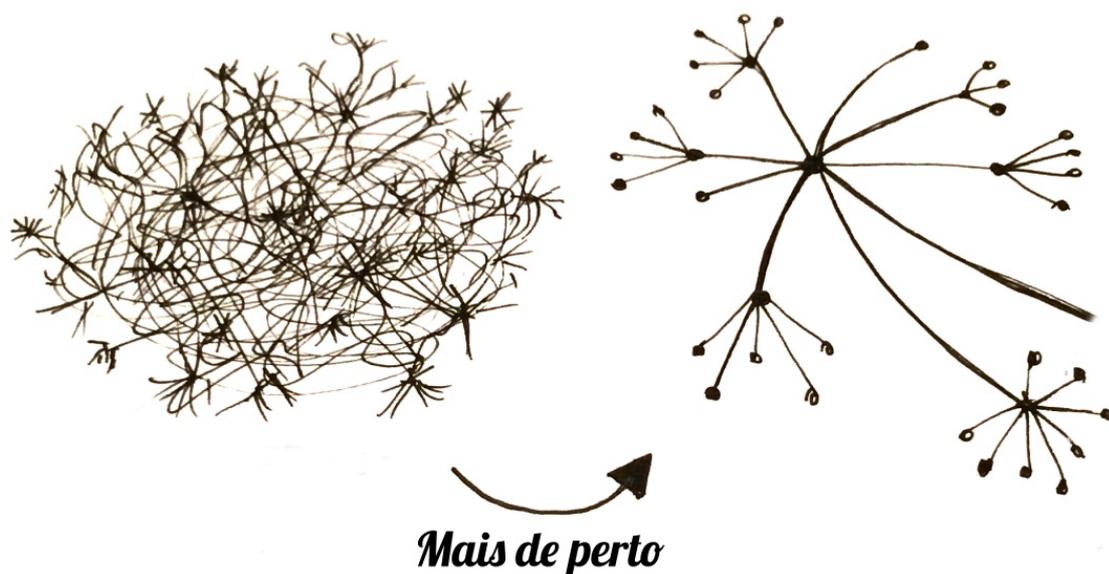
A primeira coisa que você precisa entender é que um site é só um monte de arquivos salvos em um disco rígido. Assim como seus filmes, músicas ou fotos. No entanto, existe uma parte que é exclusiva para sites: essa parte inclui código de computador chamado HTML.

Se você não estiver familiarizada com programação, pode ser difícil compreender o HTML no começo, mas seu navegador web (como o Chrome, Safari, Firefox, etc) ama ele. Navegadores Web são projetados para entender esse código, seguir suas instruções e apresentar todos esses arquivos de que seu site é feito, exatamente do jeito que você quer que eles sejam apresentados.

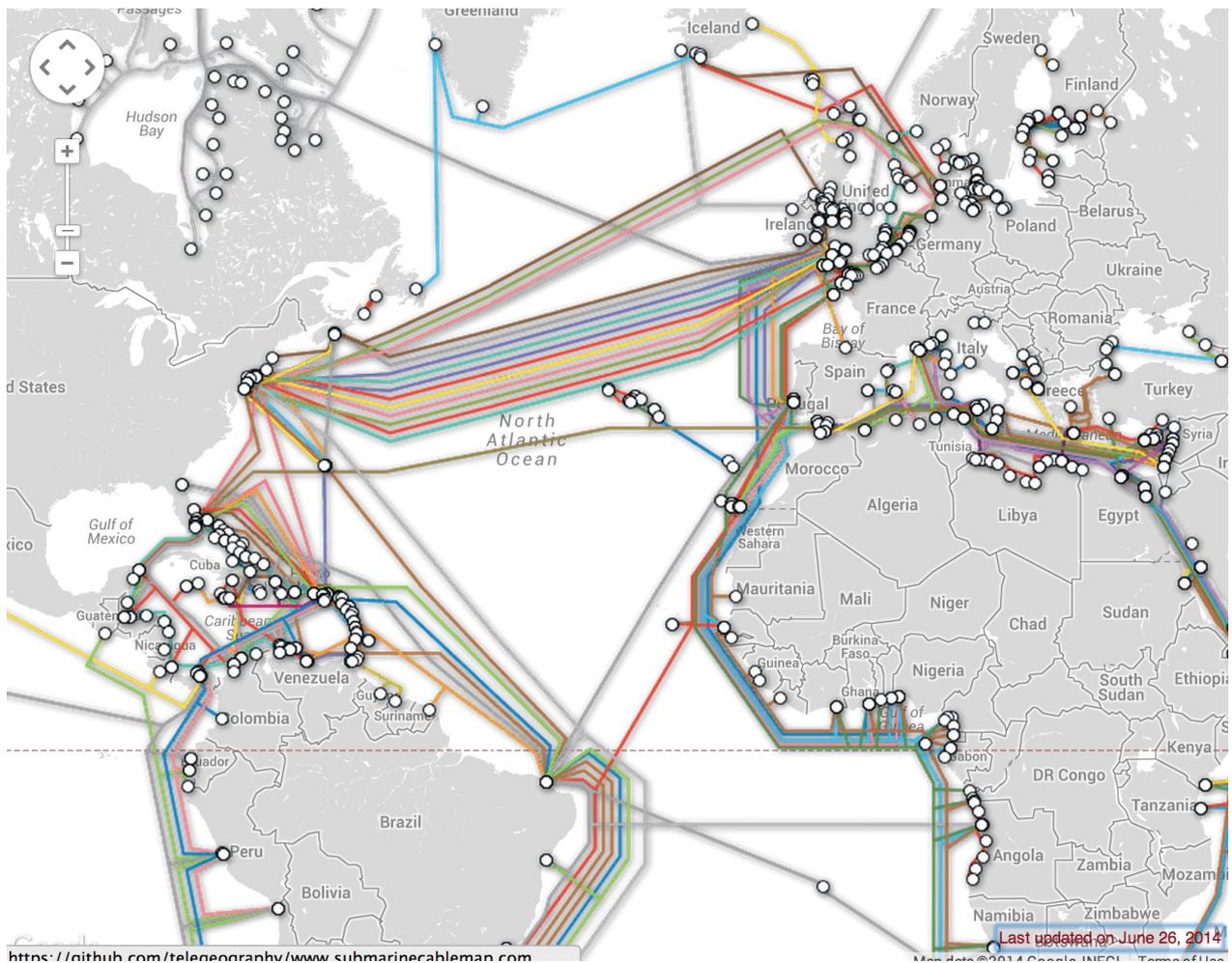
Igual à todos os arquivos, os arquivos HTML precisam ser armazenados em um disco rígido. Para a internet, nós usamos especiais e poderosos computadores chamados de *servidores*. Eles não têm tela, mouse ou teclado, porque sua finalidade principal é armazenar dados e servi-los. É por isso que eles são chamados de *servidores*...—porque eles *servem* a você, dados.

OK, mas você quer saber com o quê a internet se parece, certo?

Fizemos um desenho pra você! Veja:

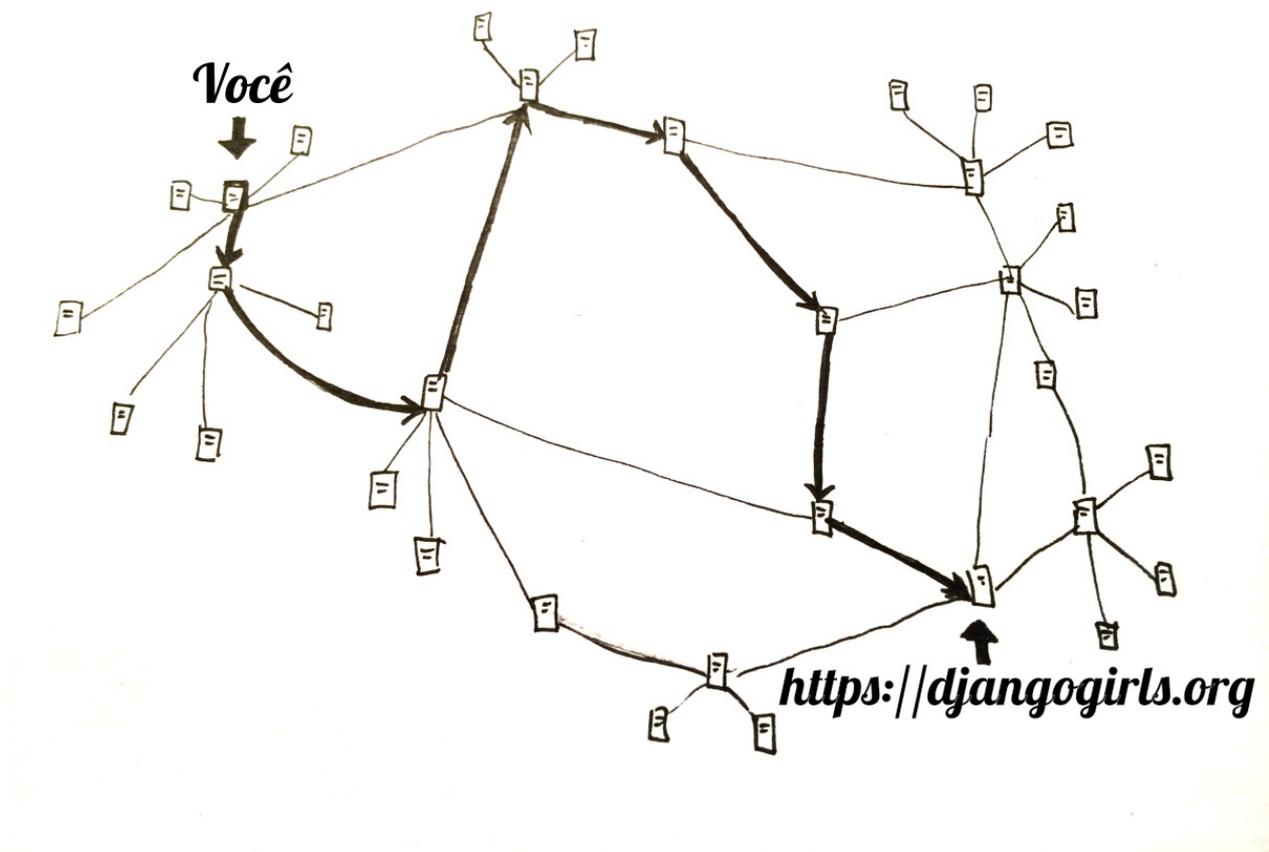


Parece uma bagunça, não é? Na verdade é uma rede de máquinas conectadas (os *servidores* mencionados acima). Centenas de milhares de máquinas! Muitos, muitos quilômetros de cabos por todo o mundo! Para ver o quão complicada a internet é, você pode visitar um site (<http://submarinecablemap.com/>) que mostra um mapa com os cabos submarinos. Aqui está um screenshot do site:



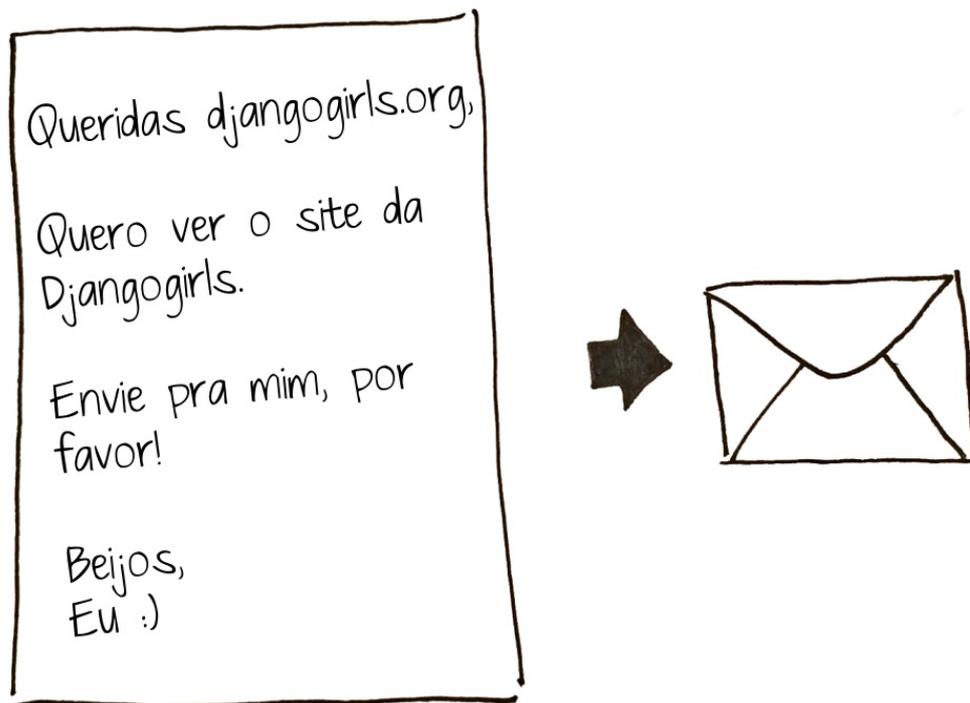
Fascinante, não? Mas, obviamente, não é possível ter um fio conectado a toda máquina ligada na internet. Logo, para chegar em uma máquina (por exemplo aquela onde <https://djangogirls.org> está salva) nós precisamos passar uma requisição através de muitas e muitas máquinas diferentes.

Se parece com isso:



Imagine que quando você digita <https://djangogirls.org>, você envia uma carta que diz: "Querido Django Girls, eu desejo ver o site djangogirls.org. Envie ele pra mim, por favor!"

Sua carta vai para a agência dos correios mais próxima de você. Depois vai para outra que é um pouco mais perto de seu destinatário, depois para outra e outra, até que ela seja entregue ao seu destino. O único diferencial é que se você enviar cartas (*pacotes de dados*) com frequência para o mesmo lugar, cada carta pode passar por diferentes agências de correios (*roteadores*), dependendo de como elas são distribuídas em cada agência.



Sim, é simples assim. Você envia mensagens e espera alguma resposta. Claro, ao invés de papel e caneta você usa bytes de dados, mas a ideia é a mesma!

Ao invés de endereços com o nome da rua, cidade, código postal e nome do país, nós usamos endereços IP. Primeiro seu computador pede ao DNS (Domain Name System - Sistema de Nome de Domínio) para traduzir djangogirls.org para um endereço IP. O funcionamento dele se parece um pouco com as antigas listas telefônicas onde você pode olhar para o nome da pessoa que quer entrar em contato e achar o seu número de telefone e endereço.

Quando você envia uma carta, ela precisa ter certas características para ser entregue corretamente: um endereço, selo, etc. Você também usa uma linguagem que o receptor compreende, certo? O mesmo acontece com *pacotes de dados* que você envia para ver um site: você usa um protocolo chamado HTTP (Hypertext Transfer Protocol - Protocolo de Transferência de Hipertexto).

Então, basicamente, quando você tem um site, você precisa ter um *servidor* (máquina) onde ele fica hospedado. O *servidor* está à espera de quaisquer *requisições* recebidas (cartas que solicitam ao servidor o envio do seu site) e ele envia de volta seu site (em outra carta).

Como este é um tutorial de Django você vai perguntar o que o Django faz. Quando você envia uma resposta nem sempre você quer enviar a mesma coisa para todo mundo. Será muito melhor se suas cartas forem personalizadas, diferenciada para a pessoa que acabou de escrever para você, certo? O Django ajuda você a criar essas personalizadas e interessantes cartas :).

Chega de falar, é hora de criar!

Introdução à linha de comando

É emocionante, não?! Você vai escrever sua primeira linha de código em poucos minutos :)

Deixe-nos apresentá-lo ao seu primeiro novo amigo: a linha de comando!

As etapas a seguir mostrarão a você como usar a janela preta que todos os hackers usam. Pode parecer um pouco assustador no começo, mas realmente é apenas um prompt esperando por comandos de você.

Qual é a linha de comando?

A janela, que normalmente é chamada de **linha de comando** ou **interface de linha de comando**, é um aplicativo baseado em texto para visualização, manipulação e manuseio de arquivos em seu computador (como por exemplo, o Windows Explorer ou o Finder no Mac, mas sem interface gráfica). Outros nomes para a linha de comando são: *cmd*, *CLI*, *prompt*, *console* ou *terminal*.

Abra a interface de linha de comando

Para começar alguns experimentos, precisamos abrir a nossa interface de linha de comando primeiro.

Windows

Vá em Iniciar → Todos os Programas → Acessórios → Prompt de comando.

Mac OS X

Applications → Utilities → Terminal.

Linux

Provavelmente você vai achar em Applications → Accessories → Terminal, mas isso depende do seu sistema operacional. Qualquer coisa é só procurar no Google :)

Prompt

Agora você deve ver uma janela branca ou preta que está à espera de seus comandos.

Se você estiver em Mac ou num Linux, você provavelmente verá um `` \$, como este:

```
$
```

No Windows, é um sinal de >, como este:

```
>
```

Cada comando será antecedido por este sinal e um espaço, mas você não precisa digitá-lo. Seu computador fará isso por você :)

Apenas uma pequena nota: no seu caso, talvez há algo como `C:\Users\ola>` OU `01as-MacBook Air: ~ ola$` antes do sinal do prompt isto estará 100% correto. Neste tutorial nós apenas simplificaremos ele para o mínimo.

Seu primeiro comando (YAY!)

Vamos começar com algo simples. Digite o seguinte comando:

```
$ whoami
```

ou

```
> whoami
```

Depois tecla Enter. Essa é nossa saída:

```
$ whoami  
olasitarska
```

Como você pode ver, o computador só apresentou seu nome de usuário. Elegante, né?:)

Tente digitar cada comando, não copiar e colar. Você vai se lembrar mais dessa forma!

O Básico

Cada sistema operacional tem o seu próprio conjunto de instruções para a linha de comando, então se certifique que você está seguindo as instruções do seu sistema operacional. Vamos tentar, certo?

Pasta atual

Seria legal saber em que pasta estamos agora, certo? Vamos ver. Digite o seguinte comando seguido de um enter:

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

Se você estiver no Windows:

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

Provavelmente você vai ver algo parecido na sua máquina. Um vez que você abre a linha de comando você já começa na pasta Home.

Nota: 'pwd' significa 'print working directory'.

Listando arquivos e pastas

Então o que tem nele? Seria legal descobrir. Vamos ver:

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

Windows:

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

Entrar em outra pasta

Talvez a gente queira entrar na nossa pasta Desktop?

```
$ cd Desktop
```

Windows:

```
> cd Desktop
```

Veja se realmente entramos na pasta:

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

Windows:

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

Aqui está!

Dica de profissional: se você digitar `cd D` e apertar a tecla `tab` no seu teclado, a linha de comando irá preencher automaticamente o resto do nome para que você possa navegar rapidamente. Se houver mais de uma pasta que comece com "D", aperte a tecla `tab` duas vezes para obter uma lista de opções.

Criando uma pasta

Que tal criar um diretório Django Girls na sua área de trabalho? Você pode fazer assim:

```
$ mkdir.djangogirls
```

Windows:

```
> mkdir.djangogirls
```

Este comando vai criar uma pasta com o nome `djangogirls` no nosso desktop. Você pode verificar se ele está lá, só de olhar na sua área de trabalho ou executando um comando `ls` (Mac ou Linux) ou `dir` (Windows)! Experimente :)

Dica de profissional: Se você não quiser digitar o mesmo comando várias vezes, tente pressionar `seta para cima` e `seta para baixo` no teclado para percorrer comandos usados recentemente.

Exercite-se!

Um pequeno desafio para você: na sua mais nova pasta criada `djangogirls` crie uma outra pasta chamada `teste`. Use os comandos `cd` e `mkdir`.

Solução:

```
$ cd djangogirls
$ mkdir teste
$ ls
teste
```

Windows:

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

Parabéns! :)

Limpando

Não queremos deixar uma bagunça, então vamos remover tudo o que fizemos até agora.

Primeiro precisamos voltar para a pasta Desktop:

```
$ cd ..
```

Windows:

```
> cd ..
```

Fazendo `cd` para `..` nós mudaremos do diretório atual para o diretório pai (que significa o diretório que contém o diretório atual).

Veja onde você está:

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

Windows:

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

Agora é hora de excluir o diretório `djangogirls`.

Atenção: A exclusão de arquivos usando `del`, `rmdir` ou `rm` é irreversível, significando *Arquivos excluídos vão embora para sempre!* Então, tenha cuidado com este comando.

```
$ rm -r djangogirls
```

Windows:

```
> rmdir /S djangogirls
djangogirls, Tem certeza <S/N>? S
```

Pronto! Para ter certeza que a pasta foi excluída, vamos checar:

```
$ ls
```

Windows:

```
> dir
```

Saindo

Por enquanto é isso! Agora você fechar a linha de comando com segurança. Vamos fazer do jeito hacker, certo?..)

```
$ exit
```

Windows:

```
> exit
```

Legal, né?:)

Sumário

Aqui vai uma lista de alguns comandos úteis:

Comando (Windows)	Comando (Mac OS / Linux)	Descrição	Exemplo
exit	exit	Fecha a janela	exit
cd	cd	Muda a pasta	cd test
dir	ls	Lista as pastas e os arquivos	dir
copy	cp	Copia um arquivo	copy c:\test\test.txt c:\windows\test.txt
move	mv	Move um arquivo	move c:\test\test.txt c:\windows\test.txt
mkdir	mkdir	Cria uma pasta	mkdir testdirectory
del	rm	Deleta uma pasta e/ou arquivo	del c:\test\test.txt

Estes são apenas alguns dos poucos comandos que você pode executar em sua linha de comando, mas você não vai usar mais nada do que isto hoje.

Se você estiver curioso, ss64.com contém uma referência completa de comandos para todos os sistemas operacionais.

Pronto?

Vamos mergulhar no Python!

Vamos começar com Python

Finalmente chegamos aqui!

Mas primeiro, vamos falar um pouco sobre o que o Python é. Python é uma linguagem de programação muito popular que pode ser usada para criar sites, jogos, softwares científicos, gráficos e muito, muito mais.

O Python é originário da década de 1980 e seu principal objetivo é ser legível por seres humanos (não apenas máquinas!), por isso ele parece muito mais simples do que outras linguagens de programação. Isso faz ele mais fácil de aprender, mas não se engane, Python também é muito poderoso!

Instalação do Python

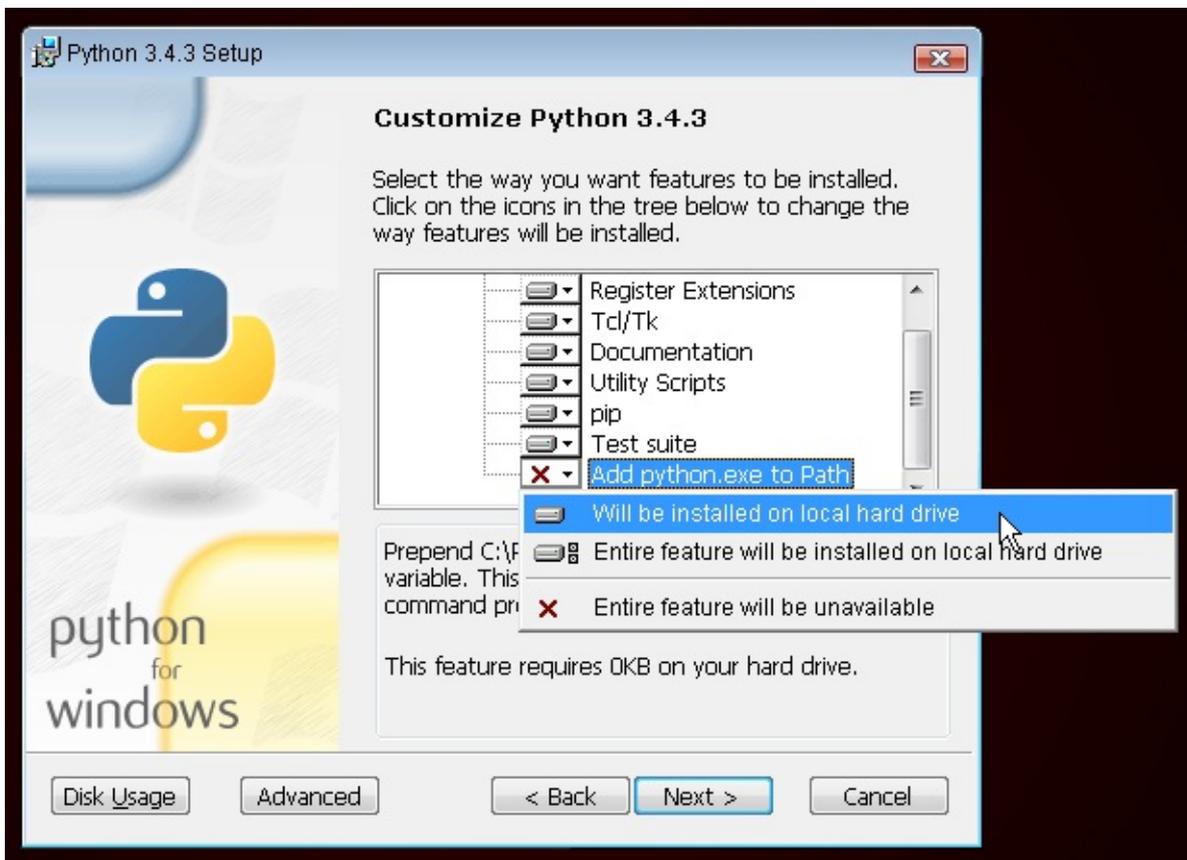
Este subcapítulo é baseado em um tutorial criado por Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>)

Django é escrito em Python. Nós precisamos dele para fazer qualquer coisa em Django. Vamos começar com sua instalação! Nós queremos que você instale o Python 3.4, então se você tem qualquer versão anterior, você vai precisar atualizá-la.

Windows

Você pode baixar o Python para Windows no website <https://www.python.org/downloads/release/python-343/>. Depois de fazer o download do arquivo *.msi, você precisa executá-lo (dando um duplo-clique nele) e seguir as instruções. É importante lembrar o caminho (a pasta) onde você instalou o Python. Ela será útil depois!

Cuidado com uma coisa: na segunda tela do assistente de instalação, marcado "Customize", certifique-se você rolar para baixo e escolha a opção "Adicionar python.exe para o caminho", como em



Linux

É muito provável que você já tenha o Python instalado e configurado. Para ter certeza se ele está instalado (e qual a sua versão), abra um terminal e digite o seguinte comando:

```
$ python3 --version
Python 3.4.2
```

Se você não tem o Python instalado ou quer uma versão diferente, você pode instalá-lo da seguinte maneira:

Ubuntu

Digite o seguinte comando no terminal:

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

Fedora

Use o seguinte comando no terminal:

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

openSUSE

Use o seguinte comando no terminal:

```
sudo zypper install python3
```

OS X

Você precisa acessar o site <https://www.python.org/downloads/release/python-342/> e baixar o instalador do Python:

- download do *instalador Mac OS X 64-bit/32-bit DMG*,
- Dê um duplo-clique para abri-lo,
- Dê um duplo-clique no *Python.mpkg* para executar o instalador.

Verifique se a instalação foi bem sucedida abrindo o *Terminal* e digitando o comando `python3` :

```
$ python3 --version
Python 3.4.2
```

Se você tiver qualquer dúvida ou se alguma coisa deu errado e você não sabe o que fazer - por favor pergunte ao seu instrutor! Às vezes, as coisas não estão indo bem e é melhor pedir ajuda a alguém com mais experiência.

Editor de Código

Você está prestes a escrever sua primeira linha de código, então é hora de baixar um editor de código!

Existem muitos editores diferentes e em grande parte se resume a preferência pessoal. A maioria dos programadores Python usa as complexas, mas extremamente poderosas IDEs (Integrated Development Environments, ou em português, Ambiente de desenvolvimento Integrado), tais como PyCharm. Para um iniciante, entretanto, estas IDE's são menos convenientes; nossas recomendações são bem mais simples, porém, igualmente poderosas.

Nossas sugestões estão abaixo, mas sinta-se livre para perguntar ao seu coach quais são as suas preferências - assim vai ser mais fácil obter a ajuda deles.

Gedit

Gedit é um editor open-source, gratuito, disponível para todos os sistemas operacionais.

[Baixe aqui](#)

Sublime Text 3

Sublime Text 3 é um editor muito popular com um período de avaliação gratuita. É fácil de instalar e usar, e está disponível para todos os sistemas operacionais.

[Baixe aqui](#)

Atom

Atom é um novo editor de código criado pelo [GitHub](#). Ele é gratuito, open-source, fácil de instalar e fácil de usar. Está disponível para Windows, OSX e Linux.

[Baixe aqui](#)

Por que estamos instalando um editor de código?

Você deve estar se perguntando porque estamos instalando esse software editor de código especial ao invés de usar algo como Word ou Bloco de Notas.

Primeiro é que código precisa ser **texto sem formatação**, e o problema com programas como Word e Textedit é que eles na verdade não produzem texto sem formatação, eles fazem "rich text" (com fontes e formatação), usando formatos personalizados como RTF.

A segunda razão é que editores de código são especializados em editar código, então eles podem oferecer recursos úteis, como realce de código com cores de acordo com o seu significado, ou automaticamente fechar citações para você.

Nós vamos ver isso em ação depois. Em breve você terá seu editor de código como uma das suas ferramentas favoritas. :)

Introdução ao Python

Parte deste capítulo é baseado nos Tutoriais de Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Vamos escrever um pouco de código!

Interpretador Python

Para começar a brincar com Python nós precisamos abrir uma *linha de comando* no seu computador. Você já deve saber como fazer isso -- você aprendeu isso no capítulo [Introdução à Linha de Comando](#). Assim que estiver pronto, siga as instruções abaixo.

Nós queremos abrir o Python num terminal, então digite `python3` e tecele Enter.

```
$ python3
Python 3.4.2 (...)
Type "copyright", "credits" or "license" for more information.
>>>
```

Seu primeiro comando Python!

Depois de executar o comando Python, o prompt mudou para `>>>`. Para nós, isso significa que por enquanto só utilizaremos comandos na linguagem Python. Você não precisa digitar `>>>` - O Python fará isso por você.

Se você deseja sair do console do Python, apenas digite `exit()` ou use o atalho `ctrl + z` no Windows e `ctrl + d` no Mac/Linux. Então você não vai ver mais o `>>>`.

Mas agora não queremos sair da linha de comando do Python. Queremos aprender mais sobre ele. Vamos, então, fazer algo muito simples. Por exemplo, tente digitar alguma operação matemática, como `2 + 3` e aperte Enter.

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

Incrível! Vê como a resposta simplesmente aparece? O Python conhece matemática! Você pode tentar outros comandos como: `- 4 * 5 - 5 - 1 - 40 / 2`

Divirta-se com isso por um tempo e depois volte aqui :).

Como você pode ver, o Python é uma ótima calculadora. Se você está se perguntando o que mais você pode fazer...

Strings

Que tal o seu nome? Digite seu primeiro nome entre aspas, desse jeito:

```
>>> "01a"
'01a'
```

Você acabou de criar sua primeira string! String é um sequência de caracteres que podem ser processada pelo computador. A string sempre precisa iniciar e terminar com o mesmo caractere. Este pode ser aspas duplas(`"`) ou simples(`'`) - elas dizem ao Python que o que está dentro delas é uma string.

Strings podem ser juntadas. Tente isto:

```
>>> "oi " + "ola"
'oi ola'
```

Você também pode multiplicar strings por um número:

```
>>> "ola" * 3
'olaolaola'
```

Se você precisa colocar uma apóstrofe dentro de sua string, existem duas maneiras de fazer.

Usando aspas duplas:

```
>>> "Correndo' ladeira abaixo"
"Correndo' ladeira abaixo"
```

ou escapando apóstrofo com uma barra invertida (\):

```
>>> "Correndo\' ladeira abaixo"
"Correndo' ladeira abaixo"
```

Legal, hein? Para ver seu nome em letras maiúsculas, basta digitar:

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

Você acabou de usar a **função** `upper` na sua string! Uma função (como `upper()`) é um conjunto de instruções que o Python realiza em um determinado objeto (`"ola"`), sempre que você chamar por ele.

Se você quer saber o número de letras do seu nome, existe uma função para isso também!

```
>>> len("ola")
3
```

Se perguntando porque algumas vezes você chama funções com um `.` no fim de uma string (como `"ola".upper()`) e algumas vezes você primeiro chama a função colocando a string nos parênteses? Bem, em alguns casos, funções pertencem a objetos, como `upper()`, que só pode ser utilizada em strings. Nesse caso, nós chamamos a função de **método**. Outras vezes, funções não pertencem a nada específico e podem ser usadas em diferentes tipos de objetos, assim como `len()`. É por isso que nós estamos fornecendo `"ola"` como um parâmetro para a função `len`.

Sumário

OK, chega de strings. Até agora você aprendeu sobre:

- **o prompt** - digitar comandos (códigos) no interpretador Python resulta em respostas em Python
- **números e strings** - no Python, números são usados para matemática e strings para objetos de texto
- **operadores** - como + e *, combinam valores para produzir um novo valor
- **funções** - como `upper()` e `len()`, executam ações nos objetos.

Isso é o básico sobre todas as linguagens de programação que você aprende. Pronto para algo mais difícil? Apostamos que sim!

Erros

Vamos tentar algo novo. Podemos obter o tamanho de um número da mesma forma que podemos encontrar o tamanho do nosso nome? Digite `len(304023)` e pressione Enter:

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

Temos nosso primeiro erro! Ele diz que objetos do tipo "int" (inteiros, apenas números) não têm nenhum comprimento. Então o que podemos fazer agora? Talvez possamos escrever nosso número como uma string? Strings têm um comprimento, certo?

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

Funcionou! Usamos a função `str` dentro da função `len`. `str()` converte tudo para strings.

- A função `str` converte as coisas em **strings**
- A função `int` converte as coisas em **números inteiros**

Importante: podemos converter números em texto, mas nós não podemos, necessariamente, converter texto em números - o que `int('hello')` quer dizer?

Variáveis

Um conceito importante na programação é o conceito de variáveis. Uma variável não é nada mais do que um nome para alguma coisa, de tal forma que você possa usá-la mais tarde. Os programadores usam essas variáveis para guardar dados, para fazer seus códigos mais legíveis e para não ter que se lembrar sempre o que algumas coisas significam.

Digamos que queremos criar uma nova variável chamada `nome`:

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

Vê? É fácil! É só fazer: nome igual a Ola.

Como você percebeu, seu programa não retornou nada como fez anteriormente. Então como sabemos que a variável realmente existe? Simplesmente digite `nome` e tecla Enter:

```
>>> nome
'Ola'
```

Yippe! Sua primeira variável! :) Você sempre pode mudar o seu valor:

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

Você pode usá-la também em funções:

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

Incrível não? Claro, variáveis podem ser qualquer coisa, então podem ser números também! Tente isso:

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

Mas, e se digitarmos o nome errado? Você consegue adivinhar o que aconteceria? Vamos tentar!

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

Um erro! Como você pode ver, Python tem diferentes tipos de erros e este é chamado **NameError**. Python dará este erro se você tentar usar uma variável que não foi definida ainda. Se você encontrar esse erro depois, veja se no seu código se você não digitou o nome de uma variável errado.

Brinque com isso por um tempo e veja o que você consegue fazer!

A função print

Tente isso:

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

Quando você apenas digita `nome`, o interpretador Python responde com a *representação* como string da variável 'name', que são as letras M-a-r-i-a, entre aspas simples. Quando você diz `print(nome)`, Python vai "imprimir" o conteúdo da variável na tela, sem as aspas, o que é mais puro.

Como veremos mais tarde, `print()` também é útil quando queremos imprimir algo dentro de funções, ou quando queremos imprimir algo em várias linhas.

Listas

Além de strings e inteiros, o Python tem todos os tipos diferentes de objetos. Vamos apresentar um chamado **lista**. Listas são exatamente o que você acha que elas são: elas são objetos que são listas de outros objetos :)

Vá em frente e crie uma lista:

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

Sim, esta é uma lista vazia. Não é muito, não é? Vamos criar uma lista dos números da loteria. Como não queremos ficar repetindo o código todo o tempo vamos criar uma variável para ela:

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

Tudo certo, nós temos uma lista! O que podemos fazer com isso? Vamos ver quantos números de loteria existem nesta lista. Você tem ideia de qual função deve usar para isso? Você já sabe disso!

```
>>> len(loteria)
6
```

Sim! `len()` pode te dar o número de objetos que fazem parte de uma lista. Uma mão na roda, não? Vamos organizar isso agora:

```
>>> loteria.sort()
```

Isso não retorna nada, apenas troca a ordem em que os números aparecem na lista. Vamos imprimir isso outra vez e ver o que acontece:

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

Como você pode ver, os números na nossa lista estão ordenados do menor para o maior. Parabéns!

Talvez a gente queira inverter essa ordem? Vamos fazer isso!

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

Moleza né? Se você quiser adicionar alguma coisa à sua lista, você pode fazer isto digitando o seguinte comando:

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

Se você quiser mostrar apenas o primeiro número você pode usar **índices**. Um índice é um número que diz onde um item da lista está. Os computadores gostam de iniciar a contagem por 0, então o primeiro objeto tem índice 0, o próximo tem índice 1 e por aí vai. Tente isso:

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

Como você pode ver, você pode acessar diferentes objetos na sua lista usando o nome da lista e o índice do objeto dentro dos colchetes.

Por diversão extra, tente alguns outros índices: 6, 7, 1000, -1, -6 ou -1000. Veja se você consegue prever o resultado antes de tentar o comando. Os resultados fazem sentido?

Você pode encontrar uma lista de todos os métodos disponíveis neste capítulo na documentação do Python:

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

Dicionários

Um dicionário é semelhante a uma lista, mas você pode acessar valores através de uma chave ao invés de um índice. Uma chave pode ser qualquer string ou número. A sintaxe para definir um dicionário vazio é:

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

Isso mostra que você acabou de criar um dicionário vazio. Hurra!

Agora, tente escrever o seguinte comando (tente substituir com as suas próprias informações também):

```
>>> participante = {'nome': 'Ola', 'pais': 'Polonia', 'numeros_favoritos': [7, 42, 92]}
```

Com esse comando, você acabou de criar uma variável chamada `participante` com três pares de chave-valor:

- A chave `nome` aponta para o valor `'ola'` (um objeto `string`),
- `pais` aponta para `'Polonia'` (outra `string`),
- e `numeros_favoritos` apontam para `[7, 42, 92]` (uma `list` com três números nela).

Você pode checar o conteúdo de chaves individuais com a sintaxe:

```
>>> print(participante['nome'])
ola
```

Veja, é similar a uma lista. Mas você não precisa lembrar o índice - apenas o nome.

O que acontece se pedirmos ao Python o valor de uma chave que não existe? Você consegue adivinhar? Vamos tentar e descobrir!

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

Olha, outro erro! Esse é um **KeyError**. Python é bastante prestativo e te diz que a chave `'idade'` não existe nesse dicionário.

Quando usar um dicionario ou uma lista? Bem, um bom ponto para refletir. Pense em uma solução antes de olhar a resposta na próxima linha.

- Você precisa de uma sequência ordenada de itens? Use uma `list`.
- Você precisa associar valores com chaves, assim você pode procurá-los eficientemente (pela chave) mais tarde? Use um `dictionary`.

Dicionários, como listas, são *mutáveis*, ou seja, que podem ser mudados depois que são criados. Você pode adicionar novos pares de chave/valor para o dicionário após sua criação, como:

```
>>> participante['linguagem_favorita'] = 'Python'
```

Como as `lists`, usar o método `len()` em dicionários retorna o número de pares chave-valor no dicionario. Vá em frente e digite o comando:

```
>>> len(participante)
4
```

Espero que agora faça sentido até agora. :) Pronta para mais diversão com dicionários? Pule na próxima linha para coisas incríveis.

Você pode usar o comando `pop()` para deletar um item no dicionario. Digamos, se você quer excluir a entrada correspondente a chave `'numeros_favoritos'`, basta digitar o seguinte comando:

```
>>> participante.pop('numeros_favoritos')
>>> participante
{'pais': 'Polonia', 'linguagem_favorita': 'Python', 'nome': 'ola'}
```

Como você pode ver no retorno, o par chave-valor correspondente à chave `'numeros_favoritos'` foi excluído.

Além disso você pode mudar o valor associado com uma chave já criada no dicionário. Digite:

```
>>> participante['pais'] = 'Alemanha'
>>> participante
{'pais': 'Alemanha', 'linguagem_favorita': 'Python', 'nome': 'ola'}
```

Como você pode ver, o valor da chave `'país'` foi alterado de `'Polônia'` para `'Alemanha'` . :) Emocionante? Hurra! Você acabou de aprender outra coisa incrível.

Sumário

Incrível! Agora você sabe muito sobre programação. Nesta última parte você aprendeu sobre:

- **erros** - agora você sabe como ler e entender erros que aparecem se o Python não entender um comando que você deu
- **variáveis** - nomes para objetos que permitem você programar facilmente e deixar seu código mais legível
- **listas** - listas de objetos armazenados em uma ordem específica
- **dicionários** - objetos armazenados como pares chave-valor

Empolgado(a) para o próximo passo? :)

Compare coisas

Grande parte da programação consiste em comparar coisas. O que é mais fácil de comparar? Números, é claro. Vamos ver como isso funciona:

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

Demos ao Python alguns números para comparar. Como você pode ver, Python pode comparar não só números mas também resultados de métodos. Legal, hein?

Você está se perguntando por que colocamos dois sinais de igual `==` lado a lado para comparar se os números são iguais? Nós usamos um único `=` para atribuir valores a variáveis. Você sempre, **sempre** precisa colocar dois `==` se quiser verificar se as coisas são iguais. Também é possível afirmar que as coisas são desiguais entre si. Para isso, usamos o símbolo `!=`, conforme mostrado no exemplo acima.

Dê ao Python mais duas tarefas:

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

`>` e `<` são fáceis, mas o que `>=` e `<=` significam? Leia eles da seguinte forma:

- `x > y` significa: x é maior que y
- `x < y` significa: x é menor que y
- `x <= y` significa: x é menor ou igual a y
- `x >= y` significa: x é maior ou igual a y

Fantástico! Quer mais? Tente isto:

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

Você pode dar ao Python quantos números para comparar quanto você quiser, e ele vai te dar uma resposta! Espertinho, certo?

- **and** - se você usar o operador `and`, ambas as comparações terão que ser verdadeiras para que todo o comando seja verdadeiro
- **or** - se você usar o operador `or`, apenas uma das comparações precisa ser verdadeira para que o comando todo seja verdadeiro

Já ouviu a expressão "comparar maçãs com laranjas"? Vamos tentar o equivalente em Python:

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

Aqui vemos que assim como na expressão, Python não é capaz de comparar um número (`int`) e uma string (`str`). Em vez disso, ele mostrou um **TypeError** e nos disse que os dois tipos não podem ser comparados juntos.

Booleano

Acidentalmente, você aprendeu sobre um novo tipo de objeto em Python. É chamado de **booleano** -- e provavelmente o tipo mais fácil que existe.

Existem apenas dois objetos booleanos: - True (verdadeiro) - False (falso)

Mas para o Python entender isso, você precisa sempre escrever True (primeira letra maiúscula, com o resto das letras em minúsculo). **true, TRUE, tRUE não vai funcionar -- só True é correto.** (O mesmo se aplica ao False, claro.)

Booleanos podem ser variáveis também! Veja:

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

Você também pode fazer desse jeito:

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

Pratique e divirta-se com os valores booleanos, tentando executar os seguintes comandos:

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

Parabéns! Booleanos são um dos recursos mais interessantes na programação, e você acabou de aprender como usá-los!

Salvá-lo!

Até agora nós escrevemos todo nosso código em um interpretador python, que nos limita a uma linha de código em um momento. Programas normais são salvos em arquivos e executados pelo nosso **interpretador** de linguagem de programação ou **compilador**. Até agora já corremos nossos programas de uma linha de cada vez no **interpretador** Python. Nós vamos precisar de mais de uma linha de código para as próximas tarefas, então precisaremos rapidamente:

- Sair do interpretador Python
- Abrir o editor de código de sua escolha
- Salvar algum código em um novo arquivo python
- Executá-lo!

Para sair do interpretador Python que estamos usando, simplesmente digite a função `exit()` :

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

Isso vai colocá-la no prompt de comando.

Anteriormente, nós escolhemos um editor de código da seção do [editor de código](#). Nós precisamos abrir o editor agora e escrever algum código em um novo arquivo:

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

Nota Você deve observar que uma das coisas mais legais sobre editores de código: cores! No console do Python, tudo era da mesma cor, mas agora você deve ver que a função de `Imprimir` é uma cor diferente da sequência de caracteres no seu interior. Isso é chamado de "realce de sintaxe", e é uma ajuda muito útil quando está programando. Perceba a cor das coisas e você vai obter uma dica para quando você esquecer de fechar uma sequência de caracteres, ou fazer um erro de digitação em um nome de palavra-chave (como `def` em uma função, que veremos abaixo). Esta é uma das razões pelas quais que nós usamos um editor de código :)

Obviamente, você é um desenvolvedor python bastante experiente agora, então sintá-se livre para escrever um código que você aprendeu hoje.

Agora temos de salvar o arquivo e dê um nome descritivo. Vamos chamar o arquivo **python_intro.py** e salve-o em seu desktop. Podemos nomear o arquivo tudo o que quisermos, o importante aqui é ter certeza que o arquivo termina no **py**, isto diz nosso computador, que é um **arquivo executável de python** e Python pode executá-lo.

Com o arquivo salvo, é hora de executá-lo! Usando as habilidades que você aprendeu na seção de linha de comando, use o terminal **altere os diretórios** para o desktop.

Em um Mac, o comando será parecido com isto:

```
cd ~/Desktop
```

No Linux, será assim (a palavra "Desktop" pode ser traduzido para seu idioma):

```
cd ~/Desktop
```

E no windows, vai ser assim:

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

Se você ficar preso, só pedir ajuda.

Em seguida, usar o Python para executar o código no arquivo assim:

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

Tudo bem! Você acabou de executar seu primeiro programa em python que foi salvo em um arquivo. Se sente ótima?

Você pode agora passar para uma ferramenta essencial na programação:

if...elif...else

Muitas coisas no código só podem ser executadas se determinadas condições forem atendidas. É por isso que o Python tem alguma coisa chamada **declaração if**.

Substitua o código no arquivo **python_intro.py** para isto:

```
if 3 > 2:
```

Se salvou este e ele foi executado, nós veríamos um erro como este:

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

Python espera que nós forneçamos mais instruções que serão supostamente executadas caso a condição `3 > 2` venha a ser verdadeira (ou `True` nesse caso). Vamos tentar fazer o Python imprimir "It works!". Altere o seu código no seu arquivo **python_intro.py** para isto:

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

Você percebeu que indentamos a próxima linha com 4 espaços? Precisamos fazer isso para que o Python saiba qual código será executado se o resultado for True. Você pode fazer com 1 espaço, mas quase todos os programadores Python fazem com 4 para deixar as coisas arrumadas. Um único tab também vai contar como 4 espaços.

Salvá-lo e execute novamente:

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

E se não?

Nos exemplos anteriores, o código foi executado somente quando as condições eram verdade. Mas o Python também tem instruções `elif` e `else`:

```
if 5 > 2:
    print('5 é realmente maior que 2')
else:
    print('5 não é maior que 2')
```

Quando for executado irá imprimir:

```
$ python3 python_intro.py
5 é realmente maior que 2
```

Se 2 for um número maior do que 5, então o segundo comando será executado. Fácil, né? Vamos ver como funciona o `elif`:

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

e executado:

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

Viu o que aconteceu?

Sumário

Nos últimos três exercícios você aprendeu:

- **comparar as coisas** - em Python, você pode comparar as coisas usando os operadores `>`, `>=`, `==`, `<=`, `<` e o `and`, `or`
- **Booleano** - um tipo de objeto que só tem um dos dois valores: `True` ou `False`
- **Salvando arquivos** - armazenamento de código em arquivos assim você pode executar programas maiores.
- **if... elif... else** - instruções que permitem que você execute o código somente se determinadas condições forem atendidas.

É hora da última parte deste capítulo!

Suas próprias funções!

Se lembra de funções como `len()` que você pode executar no Python? Bem, boas notícias, agora você vai aprender a escrever suas próprias funções!

Uma função é uma sequência de instruções que o Python deve executar. Cada função em Python começa com a palavra-chave `def`, seguido de um nome para a função e opcionalmente uma lista de parâmetros. Vamos começar com uma função simples. Substitua o código no `python_intro.py` com o seguinte:

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

hi()
```

Ok, nossa primeira função está pronta!

Você pode se perguntar por que escrevemos o nome da função na parte inferior do arquivo. Isto é porque Python lê o arquivo e executa de cima para baixo. Então, para usar a nossa função, temos de escrevê-lo na parte inferior.

Vamos executa-lo agora e ver o que acontece:

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

Isso foi fácil! Vamos construir nossa primeira função com parâmetros. Usaremos o exemplo anterior - uma função que diz 'hi' para quem o executa - com um nome:

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

Como você pode ver, agora demos um parâmetro chamado `name` para nossa função:

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

Como você pode ver, nós precisamos colocar dois espaços antes da função `print`, porque `if` precisa saber o que deve acontecer quando a condição for atendida. Vamos ver como isso funciona agora:

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

Oops, um erro. Felizmente, Python nos fornece uma mensagem de erro bastante útil. Ela diz que a função `hi()` (aquela que declaramos) tem um argumento obrigatório (chamado `name`) e que nós esquecemos de passá-lo ao chamar a função. Vamos corrigi-lo na parte inferior do arquivo:

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

e execute novamente:

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

E se mudarmos o nome?

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

e executá-lo:

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

Agora, o que acha que vai acontecer se você escrever outro nome lá? (Sem ser Ola ou Sonja) Experimentá-lo e ver se você está certo. Ele deve imprimir isto:

```
Hi anonymous!
```

Isto é incrível, não? Dessa maneira você não precisa se repetir (DRY - don't repeat yourself, ou em português, não se repita) cada vez que for mudar o nome da pessoa que a função pretende cumprimentar. E é exatamente por isso que precisamos de funções - você nunca quer repetir seu código!

Vamos fazer algo mais inteligente...--existem mais que dois nomes, e escrever uma condição para cada um seria difícil, certo?

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

Vamos chamar o código agora:

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

Parabéns! Você acabou de aprender a criar funções :)

Laços

Já é a última parte. Foi rápido, não? :)

Como mencionamos, os programadores são preguiçosos, não gostam de repetir as mesmas coisas. Programação fala sobre como automatizar as coisas, então não queremos cumprimentar cada pessoa pelo seu nome manualmente, certo? É aí onde os laços vem a calhar.

Ainda se lembra das listas? Vamos fazer uma lista de garotas:

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

Queremos cumprimentar todas elas pelos seus nomes. Temos a função `hi` para fazer isso, então vamos usá-la em um loop:

```
for name in girls:
```

O `for` se comporta da mesma forma que o `if`, o código abaixo esses dois precisam ser recuados quatro espaços.

Aqui está o código completo que será salvo no arquivo:

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

e quando executá-lo:

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

Como você pode ver, tudo o que você vai colocar dentro de uma instrução `for` com espaço será repetido para cada elemento da lista `girls`.

Você também pode usar o `for` em números usando a função `range`:

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

Que iria imprimir:

```
1  
2  
3  
4  
5
```

`range` é uma função que cria uma lista de números que se seguem um após o outro (esses números são dados por você como parâmetros).

Note que o segundo desses dois números não está incluído na lista que o Python mostrou (em `range(1, 6)`, conta de 1 a 5, mas o 6 não é incluído).

Sumário

É isso. **Você é totalmente demais!** Não é tão fácil, então você deve se sentir orgulhosa de si mesma. Estamos definitivamente orgulhosas de você por ter chegado até aqui!

Talvez você queira brevemente fazer algo mais - espreguiçar, andar um pouco, descansar os olhos - antes de ir para o próximo capítulo. :)



O que é Django?

Django é um framework gratuito e de código aberto para a criação de aplicações web, escrito em Python. É um framework web, ou seja, é um conjunto de componentes que ajuda a desenvolver sites de forma mais rápida e mais fácil.

Veja, quando você está construindo um site, você sempre precisa um conjunto similar de componentes: uma maneira de lidar com a autenticação do usuário (inscrever-se, realizar login, realizar logout), painel de gerenciamento para o seu site, formulários, upload de arquivos, etc.

Felizmente para você, há muito tempo, outras pessoas notaram várias semelhanças nos problemas enfrentados pelos desenvolvedores web quando estão criando um novo site, então eles uniram-se e criaram os frameworks (Django é um deles) que lhe dão componentes prontos, que você pode usar.

Frameworks existem para salvá-lo de ter que reinventar a roda e ajudam a aliviar a sobrecarga quando você está construindo um novo site.

Por que você precisa de um framework?

Para entender o que Django é na verdade, precisamos olhar mais de perto os servidores. A primeira coisa é que o servidor precisa saber o que você quer para servi-lo uma página da Web.

Imagine uma caixa de correio (porta) que é monitorada por cartas recebidas (requisição). Isso é feito por um servidor web. O servidor web lê a carta e envia uma resposta com uma página web. Mas, quando você quer enviar alguma coisa você precisa ter um conteúdo. E o Django é aquilo que vai lhe ajudar a criar esse conteúdo.

O que acontece quando alguém solicita um site do seu servidor?

Quando chega uma requisição para o servidor web ela é passada para o Django que tenta descobrir do que ela se trata. Primeiro ele pega um endereço web e tenta descobrir o que fazer. Essa parte é feita pelo **urlresolver** do Django. (Note que o endereço de um site se chama URL - Uniform Resource Locator, em português Localizador de Recursos Uniforme, dessa forma o nome *urlresolver*, ou resolvidor de urls, faz sentido). Isso não é muito inteligente - leva à uma lista de padrões e tenta corresponder a URL. O Django verifica padrões de cima para baixo e se algo é correspondido, passa a solicitação para a função associada (que é chamada *view*).

Imagine um carteiro com uma carta. Ela está andando pela rua e verifica cada número de casa com a que está na carta. Se ele corresponder, ela coloca a carta lá. É assim que funciona o urlresolver!

Todas as coisas interessantes são feitas dentro da *view*: podemos dar uma olhada no banco de dados para procurar algumas informações. Talvez o usuário queira mudar algo nos dados? Como uma carta dizendo: "Por favor mude a descrição do meu emprego." - a *view* checa se você tem permissão para fazer isso e então atualiza a descrição do emprego pra você, enviando em seguida uma mensagem: "Feito!". Então a *view* gera uma resposta e o Django pode enviá-la para o navegador do cliente.

Claro, a descrição acima é muito simplificada, mas você não precisa saber detalhes técnicos ainda. Ter uma ideia geral já é suficiente.

Então em vez de mergulhar em muitos detalhes, nós simplesmente vamos começar criando algo com o Django e aprenderemos todas as partes importantes ao longo do caminho!

Instalação do Django

Parte deste capítulo é baseado nos tutoriais do Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Parte deste capítulo é baseado no [django-marcador tutorial](#) licenciado sobre Creative Commons Attribution-ShareAlike 4.0 International License. O tutorial do django-marcador é protegido por direitos autorais por Markus Zapke-Gründemann et al.

Ambiente virtual

Antes de instalarmos o Django, nós iremos instalar uma ferramenta extremamente útil que irá ajudar a manter seu ambiente de desenvolvimento arrumado em seu computador. É possível ignorar este passo, mas ele é altamente recomendado. Começar com a melhor configuração possível te salvará de muitos problemas no futuro!

Então, vamos criar um **ambiente virtual**(também chamado um *virtualenv*). Isso isolará sua configuração Python/Django em uma base por projeto, significa que quaisquer mudanças que fizer em um website não afetará quaisquer outras aplicações que estiver desenvolvendo a parte. Arrumado, certo?

Tudo o que você precisa fazer é encontrar um diretório no qual você deseja criar o `virtualenv` ; Seu diretório Home, por exemplo. No Windows pode parecer como `C:\Usuário\Nome` (onde `Nome` é o nome do seu login).

Para este tutorial usaremos um novo diretório `djangogirls` do seu diretório home:

```
mkdir djangogirls
cd djangogirls
```

Nós vamos fazer um `virtualenv` chamado `myvenv` . O formato geral desse comando é:

```
python3 -m venv myvenv
```

Windows

Para criar um novo `virtualenv` , você precisa abrir o console(Nós falamos sobre isso alguns capítulos atrás, lembra-se?) e executar `C:\Python34\python -m venv myvenv` . Será algo parecido com isto:

```
C:\Usuário\Nome\djangogirls> C:\Python34\python -m venv myvenv
```

onde `C:\Python34\python` é o diretório em que você previamente instalou Python e `myvenv` é o nome da sua `virtualenv` . Você pode usar qualquer outro nome, mas sempre use minúsculas e sem espaços, acentos ou caracteres especiais. Também é uma boa ideia manter o nome curto - você irá referenciar muito a ele!

Linux e OS X

Criar um `virtualenv` tanto no Linux como OS X é simples como executar `python3 -m venv myvenv` . Será algo parecido com isto:

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

`myvenv` é o nome da sua `virtualenv` . Você pode usar qualquer outro nome, mas permaneça em caixa baixa(minúsculas) e não use espaços entre os nomes. Também é uma boa ideia manter o nome curto - você irá referenciar muito a ele!

NOTA: Iniciar o ambiente virtual no Ubuntu 14.04 assim retornará o seguinte erro:

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

Para contornar esse problema, use o comando `virtualenv .`

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

Trabalhando com o virtualenv

O comando acima criará um diretório chamado `myvenv` (ou seja o nome que você escolheu) que contém o nosso ambiente virtual (basicamente um conjunto de diretórios e arquivos). Tudo o que queremos por enquanto é iniciá-lo executando:

```
C:\Usuário\Nome\djangogirls> myvenv\Scripts\activate
```

no Windows, ou:

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

no OS X e no Linux.

Lembre-se de substituir `myvenv` com seu nome escolhido do `virtualenv` !

NOTE: às vezes `source(fonte)` pode não estar disponível. Nesses casos, tente fazer isso:

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

Você vai saber que tem um `virtualenv` funcionando quando ver o prompt no seu console se parecer com:

```
(myvenv) C:\Usuário\Nome\djangogirls>
```

ou:

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

Perceba que o prefixo `(myvenv)` aparece!

Ao trabalhar dentro de um ambiente virtual, `python` irá automaticamente se referir a versão correta para que possa utilizar `python` em vez de `python3` .

Ok, nós temos todas as dependências importantes no lugar. Finalmente podemos instalar o Django!

Instalando o Django

Agora que você tem a sua `virtualenv` iniciado, você pode instalar Django usando `pip` . No console, execute `pip install django==1.8.5` (Perceba que usamos um duplo sinal de igual: `==`).

```
(myvenv) ~$ pip install django==1.8.5
Downloading/unpacking django==1.8.5
Installing collected packages: django
Successfully installed django
Cleaning up...
```

no Windows

Se você receber um erro ao chamar o pip na plataforma Windows por favor, verifique se o caminho do projeto contém espaços, acentos ou caracteres especiais (exemplo, `C:\Users\User Name\djangogirls`). Se sim por favor mova para outro lugar sem espaços, acentos ou caracteres especiais (sugestão é: `C:\djangogirls`). Após a mudança, por favor tente novamente o comando acima.

No Linux

Se você receber um erro ao chamar pip no Ubuntu 12.04 por favor execute `python -m pip install -U --force-reinstall pip` para corrigir a instalação do pip no virtualenv.

É isso! Agora você está (finalmente) pronto para criar uma aplicação Django!

Seu primeiro projeto Django!

Parte deste capítulo é baseado nos tutoriais do Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Parte deste capítulo é baseado no [django-marcador tutorial](#) licenciado sobre Creative Commons Attribution-ShareAlike 4.0 International License. O tutorial do django-marcador é protegido por direitos autorais por Markus Zapke-Gründemann et al.

Nós vamos criar um blog simples!

O primeiro passo para criá-lo é começar um novo projeto de Django. Basicamente, isto significa que vamos executar alguns scripts fornecidos pelo Django que irá criar o esqueleto de um projeto Django para nós: um bando de diretórios e arquivos que usaremos mais tarde.

Os nomes de alguns arquivos e diretórios são muito importantes para o Django. Não renomeie os arquivos que estamos prestes a criar. Mover para um lugar diferente também não é uma boa idéia. Django precisa manter uma determinada estrutura para ser capaz de encontrar coisas importantes.

Lembre-se que tudo no virtualenv. Se você não vê um prefixo `(myvenv)` em seu console, você precisa ativar o virtualenv. Nós explicamos como fazer isso no capítulo [instalação do Django](#) na parte [trabalhando com virtualenv](#). Você pode fazer isso digitando o seguinte comando: `myvenv\Scripts\activate` no Windows ou `myvenv/bin/activate` no Mac OS / Linux.

Nota Verifique que você incluiu o ponto (`.`) no final do comando, é importante porque diz o script para instalar o Django em seu diretório atual.

No console, você deve executar (Lembre-se de que você não pode digitar `~/djangogirls$ (myvenv)`, OK?):

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

no Windows:

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

`Django-admin` é um script que irá criar os diretórios e arquivos para você. Agora, você deve ter um diretório estrutura que se parece com isso:

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

`manage.py` é um script que ajuda com a gestão do site. Com isso seremos capazes de iniciar um servidor de web no nosso computador sem instalar nada, entre outras coisas.

O arquivo `settings.py` contém a configuração do seu site.

Lembra quando falamos sobre um carteiro verificando onde entregar uma carta? arquivo `urls.py` contém uma lista dos padrões usados por `urlresolver`.

Vamos ignorar os outros arquivos por agora - nós não vamos mudá-los. A única coisa a lembrar é não excluí-los por acidente!

Configurando

Vamos fazer algumas alterações no `mysite/settings.py`. Abra o arquivo usando o editor de código que você instalou anteriormente.

Seria bom ter a hora correta no nosso site. Vá para a <[wikipedia timezones list](#) e copie seu fuso horário. (por exemplo, `America/Sao_Paulo`)

Em `settings.py`, localize a linha que contém `TIME_ZONE` e modifique para escolher seu próprio fuso horário:

```
TIME_ZONE = 'America/Sao_Paulo'
```

Modifique "America/Sao_Paulo", conforme o caso

Nós também precisaremos adicionar um caminho para arquivos estáticos (nós vamos descobrir tudo sobre arquivos estáticos e CSS mais tarde no tutorial). Desça até o *final* do arquivo e logo abaixo da entrada `STATIC_URL`, adicione um novo um chamado `STATIC_ROOT`:

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

Instalação de um banco de dados

Há vários softwares de banco de dados diferentes que pode armazenar dados para o seu site. Nós vamos usar o padrão, `sqlite3`.

Isto já está configurado nesta parte do seu arquivo `mysite/settings.py`:

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

Para criar um banco de dados para o nosso blog, vamos fazer o seguinte no console. Digite: `python manage.py migrate` (precisamos estar no diretório que contém o arquivo `manage.py` `djangogirls`). Se isso der certo, você deve ver algo como isto:

```
(myenv) ~/djangogirls$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying sessions.0001_initial... OK
```

E está pronto! Hora de iniciar o servidor web e ver se nosso site está funcionando!

Você precisa estar no diretório que contém o arquivo `manage.py` (o diretório `djangogirls`). No console, nós podemos iniciar o servidor web executando o `python manage.py runserver`:

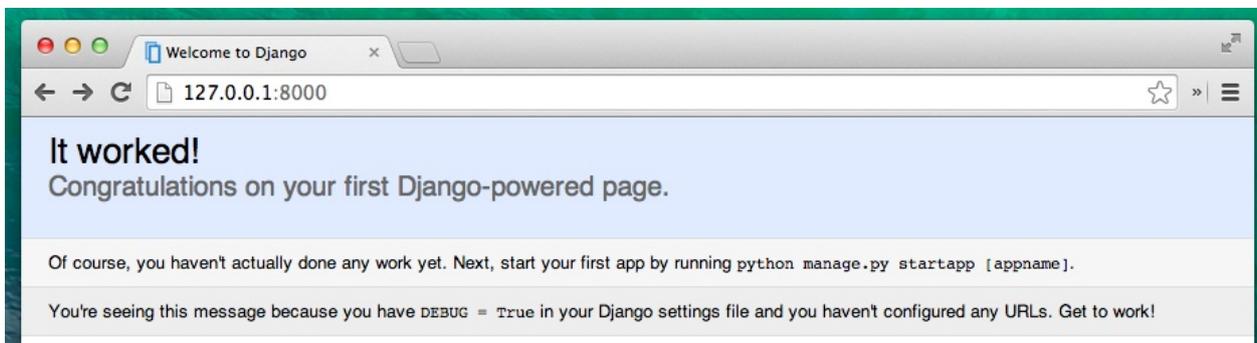
```
(myenv) ~/djangogirls$ python manage.py runserver
```

Agora tudo que você precisa fazer é verificar se seu site está sendo executado - abra seu navegador (Firefox, Chrome, Safari, Internet Explorer ou o que você usa) e digite o endereço:

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

O servidor web vai assumir seu prompt de comando até você pará-lo: para digitar mais comandos ou abrir uma nova janela do terminal (e não se esqueça de ativar seu virtualenv nele também), ou parar o servidor de web, alternando de volta para a janela na qual está executando e pressionando CTRL + C - botões de controle e C juntos (no Windows, você pode ter que pressionar Ctrl + Break).

Parabéns! Você criou seu primeiro site e o executou usando um servidor de web! Não é impressionante?



Pronto para o próximo passo? Está na hora de criar algum conteúdo!

Modelos do Django

Agora o que nós queremos criar é algo que armazene todos os posts no nosso blog. Mas para fazer isso precisamos aprender um pouco mais sobre coisas chamadas `objetos`.

Objetos

Existe um conceito na programação chamado `Programação Orientada à Objetos (POO)`. A ideia é que ao invés de escrever tudo como uma chata sequência de instruções de programação podemos modelar as coisas e definir como elas interagem umas com as outras.

Então o que é um objeto? É uma coleção de propriedades e ações. Isto pode parecer estranho, mas vamos lhe dar um exemplo.

Se queremos modelar um gato nós criaremos um objeto `Gato` que possui algumas propriedades, por exemplo `cor`, `idade`, `humor` (bom, mau, sonolento ;)), `dono` (que é um objeto da classe `Pessoa` ou, caso seja um gato de rua, essa propriedade é vazia).

E então o `Gato` tem algumas ações: `ronronar`, `arranhar` ou `comer` (no qual vamos dar ao gato alguma `ComidaDeGato`, que poderia ser um objeto separado com propriedades, como `sabor`).

```
Gato
-----
cor
idade
humor
dono
ronronar()
arranhar()
comer(comida_de_gato)

ComidaDeGato
-----
sabor
```

Então, basicamente, a ideia é descrever coisas reais no código com propriedades(chamadas de `propriedades do objeto`) e ações (chamadas de `métodos`).

Como nós iremos modelar as postagens do blog então? Queremos construir um blog, certo?

Precisamos responder à pergunta: o que é uma postagem de blog? Que propriedades deve ter?

Bem, com certeza nosso blog precisa de alguma postagem com o seu conteúdo e um título, certo? Também seria bom saber quem a escreveu - então precisamos de um autor. Finalmente, queremos saber quando a postagem foi criada e publicada.

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

Que tipo de coisa pode ser feita com uma postagem? Seria legal ter algum `método` que publique a postagem, não é mesmo?

Então precisamos de um método chamado `publicar`.

Como já sabemos o que queremos alcançar, podemos começar a modelagem em Django!

Modelo do Django

Sabendo o que um objeto é, nós criaremos um modelo no Django para a postagem do blog.

Um modelo no Django é um tipo especial de objeto - ele é salvo em um `banco de dados`. Um banco de dados é uma coleção de dados. O banco de dados é um local em que você vai salvar dados sobre usuários, suas postagens, etc. Usaremos um banco de dados chamado SQLite para armazenar as nossas informações. Este é o adaptador de banco de dados padrão Django -- ele vai ser o suficiente para nós neste momento.

Você pode pensar em um modelo de banco de dados como uma planilha com colunas (campos) e linhas (dados).

Criando uma aplicação

Para manter tudo arrumado vamos criar um aplicativo separado dentro do nosso projeto. É muito bom ter tudo organizado desde o início. Para criar um aplicativo precisamos executar o seguinte comando no console (a partir do diretório

```
djangoirls onde está o arquivo manage.py ):
```

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

Você vai notar que um novo diretório `blog` é criado e que ele agora contém um número de arquivos. Nossos diretórios e arquivos no nosso projeto devem se parecer com este:

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

Depois de criar um aplicativo também precisamos dizer ao Django que deve usá-lo. Fazemos isso no arquivo `mysite/settings.py`. Precisamos encontrar o `INSTALLED_APPS` e adicionar uma linha com `'blog'`, logo acima do `)`. É assim que o produto final deve ficar assim:

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

Criando o modelo Post do nosso blog

No arquivo `blog/models.py` definimos todos os objetos chamados `Modelos` - este é um lugar em que vamos definir nossa postagem do blog.

Vamos abrir `blog/models.py`, remova tudo dele e escreva o código como este:

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

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

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

    def __str__(self):
        return self.title
```

Certifique-se de ter usado dois caracteres (`_`) em cada lado do `str`. Aqueles caracteres são usados frequentemente em Python e às vezes os chamamos de "dunder" (abreviação de "double-underscore" ou "duplo sublinhado").

É assustador, não? Mas não se preocupe, vamos explicar o que estas linhas significam!

Todas as linhas começando com `from` ou `import` são linhas que adicionam alguns pedaços de outros arquivos. Então ao invés de copiar e colar as mesmas coisas em cada arquivo, podemos incluir algumas partes com `from... import ...`.

`class Post(models.Model):` - esta linha define o nosso modelo (é um objeto).

- `class` é uma palavra-chave especial que indica que estamos definindo um objeto.
- `Post` é o nome do nosso modelo, podemos lhe dar um nome diferente (mas é preciso evitar os espaços em branco e caracteres especiais). Sempre comece um nome de classe com uma letra maiúscula.
- `models.Model` significa que o `Post` é um modelo de Django, então o Django sabe ele que deve ser salvo no banco de dados.

Agora podemos definir as propriedades que discutimos: `titulo`, `texto`, `data_criacao`, `data_publicacao` e `autor`. Para isso precisamos definir um tipo de campo (é um texto? É um número? Uma data? Uma relação com outro objeto, por exemplo, um usuário?).

- `models.CharField` - assim é como você define um texto com um número limitado de caracteres.
- `models.TextField` - este é para textos longos sem um limite. Será ideal para um conteúdo de post de blog, certo?
- `models.DateTimeField` - este é uma data e hora.
- `models.ForeignKey` - este é um link para outro modelo.

Nós não vamos explicar cada pedaço de código aqui, pois isso levaria muito tempo. Você deve olhar a documentação do Django se você quiser saber mais sobre campos do Model e como definir coisas além destas descritas acima (<https://docs.djangoproject.com/en/1.8/ref/models/fields/#field-types>).

Que tal `def publish(self):`? É exatamente o nosso método de `publish` que falávamos antes. `def`, significa que se trata de um função/método. `publish` é o nome do método. Você pode alterar, se quiser. A regra é que usamos letras minúsculas e sublinhados em vez de espaços em branco (ou seja, se você quer ter um método que calcula o preço médio, você poderia chamá-lo `calculate_average_price`).

Métodos muitas vezes `return` algo. Há um exemplo de que, no método `__str__`. Nesse cenário, quando chamamos `__str__()` teremos um texto (**string**), com um título do Post.

Se algo ainda não está claro sobre modelos, sinta-se livre para pedir o seu treinador! Sabemos que é muito complicado, especialmente quando você aprender o que são objetos e funções ao mesmo tempo. Mas espero que ele se parece um pouco menos mágica para você agora!

Criando tabelas para nossos modelos no banco de dados

O último passo é adicionar nosso novo modelo para nosso banco de dados. Primeiro temos que fazer o Django saber que nós temos algumas mudanças em nosso modelo (só criamos isso), digite `python manage.py makemigrations blog`. Será algo parecido com isto:

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

Django prepara um arquivo de migração que temos de aplicar agora para nosso banco de dados, tipo `python manage.py migrate blog`, a saída deve ser:

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

Viva! Nosso modelo de Post está agora em nosso banco de dados, seria um prazer vê-lo, certo? Saltar para o próximo capítulo para ver o aspecto do seu Post!

Administração

Para adicionar, editar e remover postagens que nós criamos usaremos o Django admin.

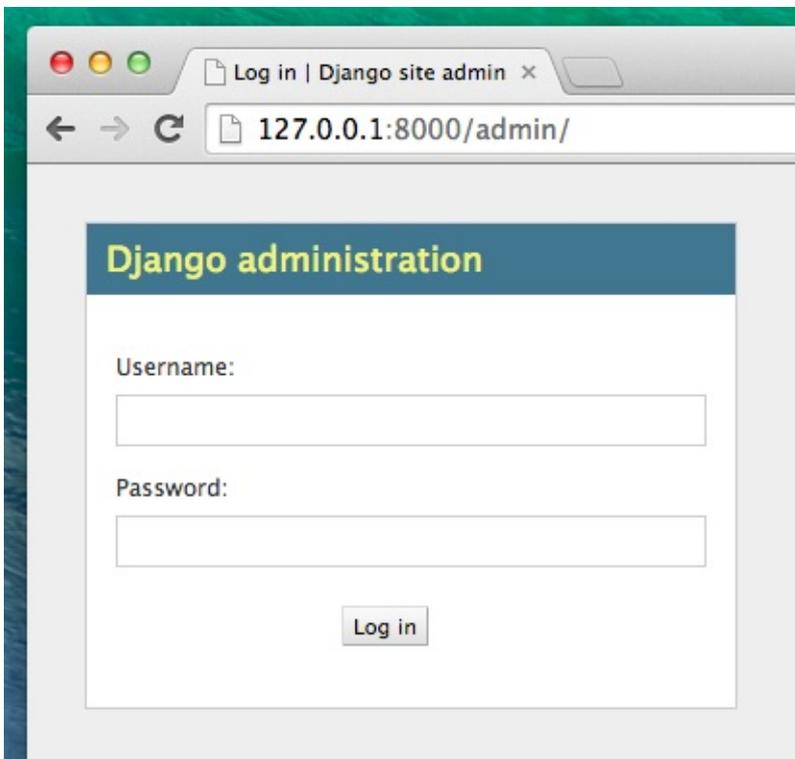
Vamos abrir o arquivo `blog/admin.py` e substituir seu conteúdo por:

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

admin.site.register(Post)
```

Como você pode ver, nós importamos (incluímos) o modelo `Post` definido no capítulo anterior. Para tornar nosso modelo visível na página de administração, nós precisamos registrá-lo com: `admin.site.register(Post)`.

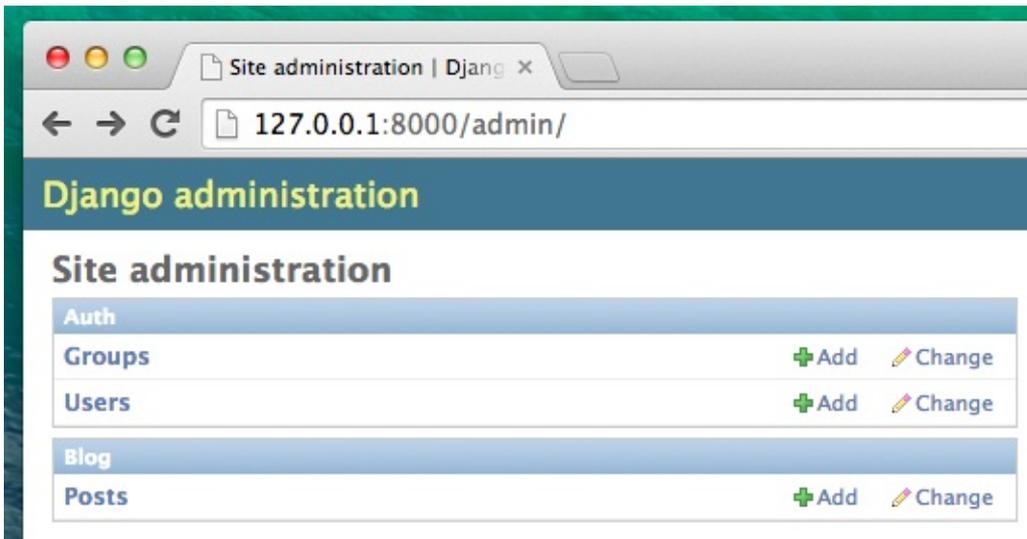
OK, hora de olhar para o nosso modelo de `Post`. Lembre-se de executar `python manage.py runserver` no console para executar o servidor web. Vá para o navegador e digite o endereço <http://127.0.0.1:8000/admin/> Você verá uma página de login assim:



Para fazer login você precisa criar um *superuser* - um usuário que possui controle sobre tudo do site. Volte para o terminal e digite `python manage.py createsuperuser`, pressione enter e digite seu nome de usuário (caixa baixa, sem espaço), endereço de e-mail e password quando eles forem requisitados. Não se preocupe que você não pode ver a senha que você está digitando - é assim que deve ser. Só digitá-la e pressione 'Enter' para continuar. A saída deve parecer com essa (onde Username e Email devem ser os seus):

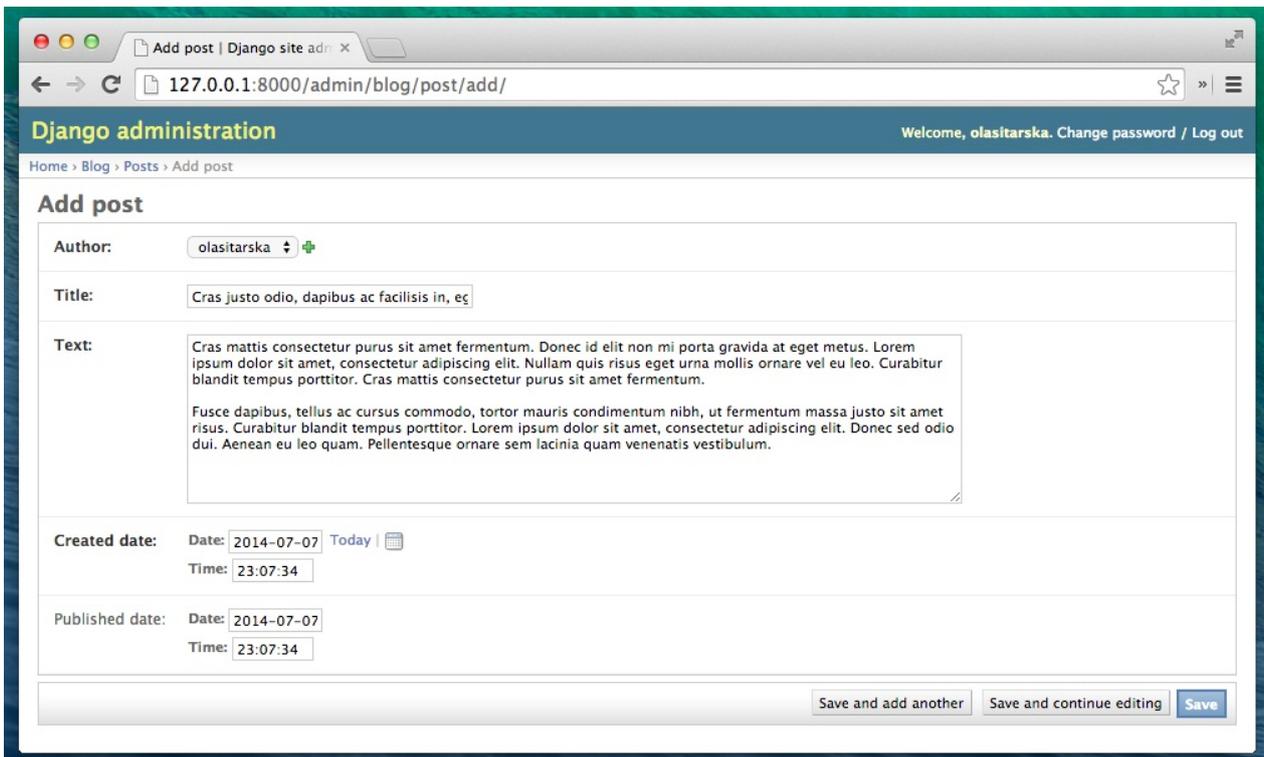
```
(myvenv) ~/djangogirls$ python manage.py createsuperuser
Username: admin
Email address: admin@admin.com
Password:
Password (again):
Superuser created successfully.
```

Volte para a o navegador e faça login com as credenciais de superuser que você escolheu, você deve visualizar o painel de controle do Django admin.



Vá para as postagens e experimente um pouco com elas. Adicione cinco ou seis postagens. Não se preocupe com o conteúdo - você pode copiar e colar algum texto deste tutorial para o conteúdo para economizar tempo :).

Certifique-se que pelo menos duas ou três postagens (mas não todas) têm a data de publicação definida. Isso será útil depois.



Se você quiser saber mais sobre o Django admin, você deve conferir a documentação do Django:

<https://docs.djangoproject.com/en/1.8/ref/contrib/admin/>

Este é provavelmente um bom momento para tomar um café (ou chocolate) ou algo para comer para repor as energias. Você criou seu primeiro modelo de Django - você merece um pouco de descanso!

Implantação!

Nota O capítulo seguinte pode ser às vezes um pouco difícil de passar. Persista e termine-o; Implantação é uma parte importante do processo de desenvolvimento de website. Este capítulo está localizado no meio do tutorial para que seu tutor possa lhe ajudar com o processo ligeiramente complexo de colocar seu site online. Isto significa que você ainda pode terminar o tutorial por conta própria se você continuar em outro momento.

Até agora nosso site só estava disponível no seu computador, agora você vai aprender como publicar ele na internet! A implantação é o processo de publicação do seu aplicativo na Internet de tal forma que as pessoas possam, finalmente, ver seu aplicativo :).

Como você aprendeu, um website precisa estar localizado num servidor. Existem muitos provedores, mas iremos utilizar o que tem um processo de deploy relativamente simples: [PythonAnywhere](#). PythonAnywhere é gratuito para aplicações pequenas que não possuem muitos visitantes, então será suficiente para você por enquanto.

O outro serviço externo que usaremos é [GitHub](#), que é um serviço de hospedagem de código. Existem outros, mas quase todos os programadores possuem uma conta no GitHub atualmente e agora você também!

Usaremos o GitHub como um trampolim para transportar nosso código para o PythonAnywhere.

Git

Git é "sistema de controle de versão" usado por muitos programadores - um software que controla mudanças nos arquivos ao longo do tempo para que você possa recuperar versões específicas depois. Um pouco como "controlar mudanças" no Microsoft Word, mas muito mais poderoso.

Instalando o Git

Windows

Você pode baixar Git em [git-scm.com](#). Você pode apertar "next next next" em todos os passos exceto um; no quinto passo chamado "Adjusting your PATH environment", escolha "Run Git and associated Unix tools from the Windows command-line" (a opção de baixo). Além disso, o padrão está ótimo. Checkout estilo Windows, commit Unix-style linhas de confirmação está bom.

MacOS

Baixar Git [git-scm.com](#) e siga as instruções.

Linux

Se ele já não estiver instalado, Git deve estar disponível através de seu gerenciador de pacotes, então tente:

```
sudo apt-get install git
# or
sudo yum install git
```

Começando nosso repositório no Git

Git controla as alterações para um determinado conjunto de arquivos no que chamamos de repositório de código (ou "repo"). Vamos começar um para nosso projeto. Abra o console e execute esses comandos, no diretório `djangogirls` :

Nota: Verifique o seu diretório de trabalho atual com um `pwd` (OSX/Linux) ou o comando `cd` (Windows) antes de inicializar o repositório. Você deve estar na pasta `djangoirls`.

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

Inicializar o repositório git é algo que só precisamos fazer uma vez por projeto (e você não terá que re-introduzir o nome de usuário e e-mail nunca mais)

Git irá controlar as alterações para todos os arquivos e pastas neste diretório, mas existem alguns arquivos que queremos ignorar. Fazemos isso através da criação de um arquivo chamado `.gitignore` no diretório base. Abra seu editor e crie um novo arquivo com o seguinte conteúdo:

```
*.pyc
__pycache__
myenv
db.sqlite3
.DS_Store
```

E salve como `.gitignore` na pasta de nível superior "djangoirls".

Nota: O ponto no início do nome do arquivo é importante! Se você está tendo alguma dificuldade em criá-la (Macs não gostam de criar arquivos que começam com um ponto através do Finder, por exemplo), use o recurso "Save As" no seu editor que sempre funciona.

É uma boa idéia para usar um comando de `git status` antes de `git add` ou sempre que você não tiver certeza de que será feito, para evitar surpresas (por exemplo, serão adicionados arquivos errados ou commitados). O comando `git status` retorna informações sobre todos os arquivos controlado/modificado/encenado, status de ramo e muito mais. O output deve ser semelhante a:

```
$ git status
On branch master

Initial commit

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

.gitignore
blog/
manage.py
mysite/

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

E finalmente nós salvamos nossas alterações, Vá para o seu console e execute estes comandos:

```
$ 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
```

Empurrando o nosso código para GitHub

Vá para [GitHub.com](https://github.com) e cadastre uma nova e gratuita conta de usuário. Em seguida, crie um novo repositório, e dê o nome "my-first-blog". Deixe o "initialise with a README" desmarcado, deixe a opção .gitignore em branco (já fizemos isso manualmente) e a licença como None.

Owner hjwp / **Repository name** my-first-blog ✓

Great repository names are short and memorable. Need inspiration? How about **ducking-octo-tyrion**.

Description (optional)

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

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

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

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

Create repository

Nota O nome `my-first-blog` é importante --você poderia escolher outra coisa, mas vamos usá-lo muitas vezes nas instruções abaixo e você teria que substituí-lo cada vez. É provavelmente mais fácil ficar com o nome `my-first-blog`.

Na tela seguinte, você será mostrada a clone URL do seu repo. Escolha a versão "HTTPS", copie, e vamos colá-lo no terminal em breve:

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
```

Agora precisamos ligar o repositório Git no seu computador com o no GitHub.

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

Digite seu GitHub username e senha, e você deve ver algo como isto:

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

Seu código agora está no GitHub. Vá e confira! Você saberá que está em boa companhia - [Django](#), o [Django Girls Tutorial](#) e muitos outros grandes projetos de software de fonte aberta também hospedam seu código no GitHub :)

Criação de nosso blog em PythonAnywhere

Em seguida, é hora de se inscrever para uma conta gratuita de "Beginner" na PythonAnywhere.

- www.pythonanywhere.com

Nota: ao escolher seu nome de utilizador aqui, tenha em mente que a URL do seu blog terá o formulário `yourusername.pythonanywhere.com`, então escolha seu nickname ou o nome do que é o blog.

Pulling our code down on PythonAnywhere

Quando você se inscreve para PythonAnywhere, você é levado ao seu painel de controle ou página "Consoles". Escolha a opção iniciar o console "Bash"--que é a versão PythonAnywhere de um console, como aquela no seu PC

Nota: PythonAnywhere é baseado em Linux, assim se você estiver no Windows o console vai parecer um pouco diferente do que está no seu computador.

Vamos puxar nosso código de GitHub em PythonAnywhere através da criação de um "clone" do repo. Digite o seguinte para o console na PythonAnywhere:

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

Isto puxará uma cópia do seu código para PythonAnywhere. Confira digitando:

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

Criando um virtualenv na PythonAnywhere

Assim como fez em seu próprio computador, você pode criar um virtualenv na PythonAnywhere. No console Bash, digite:

```
20:20 ~ $ cd my-first-blog

20:20 ~ $ virtualenv --python=python3.4 myvenv
Running virtualenv with interpreter /usr/bin/python3.4
[...]
Installing setuptools, pip...done.

20:20 ~ $ source myvenv/bin/activate

(mvenv)20:20 ~ $ pip install django whitenoise
Collecting django
[...]
Successfully installed django-1.8.5 whitenoise-2.0
```

Coleta de arquivos estáticos.

Você estava imaginando o que é "whitenoise"? É uma ferramenta para servir os chamados "arquivos estáticos". Arquivos estáticos funcionam de forma diferente nos servidores em comparação com nosso próprio computador, e precisamos de uma ferramenta como o "whitenoise" para atendê-los.

Vamos descobrir um pouco mais sobre arquivos estáticos mais tarde no tutorial, quando vamos editar o CSS para o nosso site.

Por enquanto só precisamos executar um comando extra chamado "collectstatic" no servidor. Isso diz pro Django reunir todos os arquivos estáticos que ele precisa no servidor. Em sua maioria, estes são os arquivos estáticos que fazem o site do admin bonito no momento.

```
20:20 ~ $ python manage.py collectstatic

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

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

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

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

Digite "yes" (Sim) e vai embora! Você não adora fazer computadores imprimir páginas e páginas de texto? Sempre faça pequenos ruídos para acompanhá-lo. Brp, brp brp...

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

Criando o banco de dados em PythonAnywhere

Aqui está outra coisa que é diferente entre seu computador e o servidor -- ele usa um banco de dados diferente. Então as contas de usuário e mensagens podem ser diferentes no servidor e no seu computador.

Então nós vamos inicializar o banco de dados no servidor tal como fizemos no seu próprio computador, com `migrate` e `createsuperuser` :

```
(mvenv)20:20 ~ $ python manage.py migrate
Operations to perform:
[...]
Applying sessions.0001_initial... OK

(mvenv)20:20 ~ $ python manage.py createsuperuser
```

Publicação do nosso blog como um aplicativo web

Agora nosso código está na PythonAnywhere, nossa virtualenv está pronta, os arquivos estáticos estão recolhidos e o banco de dados está inicializado, estamos prontos para publicá-lo como um aplicativo da web.

Clique em voltar para o PythonAnywhere dashboard clicando no seu logotipo e clique na guia **Web** e vá em **Add a new web app**.

Na caixa de diálogo, após a confirmação de seu nome de domínio, escolha **manual configuration** (NB *não* a opção "Django"). Em seguida, escolha **Python 3.4** e clique em Next para concluir o assistente.

Nota certifique-se você escolheu a opção "Manual configuration", não a "Django". Nós somos demais para o padrão de configuração Django da PythonAnywhere ;-)

Definindo o virtualenv

Você será levado para a tela de configuração de PythonAnywhere para seu webapp que é onde você precisará de ir quando quiser fazer alterações para o aplicativo no servidor.

The screenshot shows the PythonAnywhere dashboard for the domain `edith.pythonanywhere.com`. The 'Web' tab is selected. Under 'Actions', there are buttons for 'Reload edith.pythonanywhere.com' and 'Delete edith.pythonanywhere.com'. The 'Code' section shows the source code path, WSGI configuration file, and Python version (3.4). The 'Virtualenv' section is highlighted with a red arrow, and the path `/home/edith/my-first-blog/myvenv` is entered in the input field, with another red arrow pointing to the checkmark button.

Na seção "Virtualenv", clique no texto vermelho que diz "Enter the path to a virtualenv" e digite: `/home/<your-username>/my-first-blog/myvenv/`

Nota: substitua seu próprio nome de usuário conforme apropriado. Se você cometer um erro, PythonAnywhere irá mostrar um pequeno aviso.

Configurando o arquivo WSGI

Django funciona usando o WSGI protocol, um padrão para servir sites usando Python, que oferece suporte a PythonAnywhere. A maneira que configuramos PythonAnywhere para reconhecer nosso blog Django é editando um arquivo de configuração do WSGI.

Clique no link "WSGI configuration file" (na seção "Code" perto do topo da página - -ele vai ser nomeado algo como `/var/www/<your-username>_pythonanywhere_com_wsgi.py`), e você será levado para um editor.

Exclua todo o conteúdo atual e substitua com algo parecido com isto:

```
import os
import sys

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

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

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

Nota não se esqueça de substituir em seu próprio nome de usuário onde diz `<your-username>`

O que esse arquivo faz é dizer PythonAnywhere onde mora a nossa aplicação web e qual o nome do arquivo de configurações Django. Ele também define a ferramenta de arquivos estáticos "whitenoise".

Aperte **Save** e então volte para a guia **Web**.

Já terminamos! Aperte o botão grande verde de **Reload** e você será capaz de ir ver os seu aplicativo. Você encontrará um link para ele no topo da página.

Dicas de debugging

Se você vir um erro quando você tenta visitar o seu site, o primeiro lugar para procurar alguma informação de debugging é no seu **error log** -- você encontrará um link para isso na guia web PythonAnywhere. Ver se há mensagens de erro lá. As mais recentes estão na parte inferior. Problemas comuns incluem

- esquecer um dos passos que fizemos no console: criando o virtualenv, ativá-lo, instalando o Django, collectstatic, inicializando o banco de dados
- cometer um erro no caminho do virtualenv na guia web -- haverá geralmente uma pequena mensagem de erro vermelho lá, se há um problema
- cometer um erro no arquivo de configuração WSGI...--você usou o caminho para a pasta do my-first-blog certinho?

O treinador está aqui para ajudar!

Você está live!

A página padrão para seu site deve dizer "Bem-vindo ao Django", como acontece no seu PC local. Tente adicionar `/admin/` para o final da URL, e você será levado ao site admin. Fazer login com o nome de usuário e senha, e você verá que você pode adicionar novas mensagens no servidor.

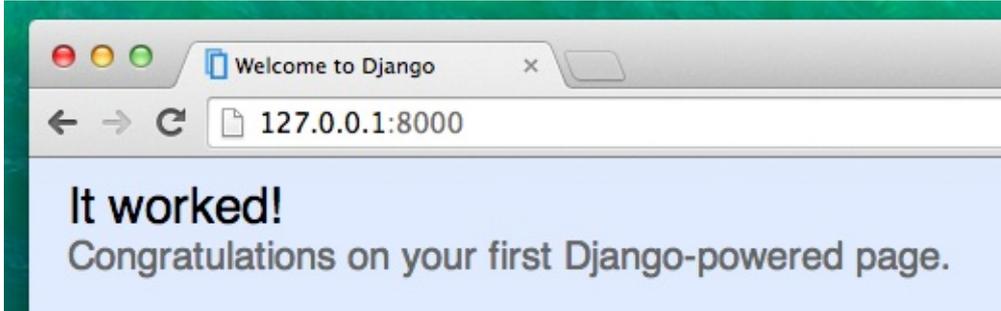
Dê em você mesma um *enorme* tapinha nas costas - implantações de servidor são uma das partes mais difíceis do desenvolvimento web, e muitas vezes leva as pessoas vários dias antes de fazer funcionar. Mas você tem seu site publicado, na internet, assim!

Urls

Estamos prestes a construir nossa primeira Web page - uma página inicial para o seu blog! Mas primeiro, vamos aprender um pouco mais sobre Django urls.

O que é uma URL?

Uma URL é simplesmente um endereço da web, você pode ver uma URL toda vez que você visita qualquer site - é visível na barra de endereços do seu navegador (Sim! `127.0.0.1:8000` é uma URL! E <http://djangogirls.org> também é uma URL):



Cada página na Internet precisa de sua própria URL. Desta forma seu aplicativo sabe o que deve mostrar a um usuário que abre uma URL. Em Django, nós usamos algo chamado `URLCONF` (configuração de URL), que é um conjunto de padrões que Django vai tentar coincidir com a URL recebida para encontrar a visão correta.

Como funcionam as URLs em Django?

Vamos abrir o arquivo `mysite/urls.py` e ver com que ele se parece:

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

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

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

Como você pode ver, o Django já colocou alguma coisa lá pra nós.

As linhas que começam com `#` são comentários - isso significa que essas linhas não serão executadas pelo Python. Muito útil, não?

A URL do admin, que você visitou no capítulo anterior já está aqui:

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

Isso significa que para cada URL que começa com `admin /` o Django irá encontrar um correspondente *modo de exibição*. Neste caso nós estamos incluindo um monte de admin URLs para que isso não fique tudo embalado neste pequeno arquivo...--é mais legível e mais limpo.

Regex

Você quer saber como o Django coincide com URLs para views? Bem, esta parte é complicada. o Django usa o `regex` -- expressões regulares. Regex tem muito (muito!) de normas que formam um padrão de pesquisa. Como regexes são um tópico avançado, nós veremos em detalhes como elas funcionam.

Se você ainda quiser entender como criamos os padrões, aqui está um exemplo do processo - só precisamos um subconjunto limitado de regras para expressar o padrão que procuramos, ou seja:

```
^ para o início do texto
$ para o final do texto
\d para um dígito
+ para indicar que o item anterior deve ser repetido pelo menos uma vez
() para capturar parte do padrão
```

Qualquer outra coisa na definição de url será levada literalmente.

Agora imagine que você tem um site com o endereço assim: `http://www.mysite.com/post/12345/`, onde `12345` é o número do seu post.

Escrever views separadas para todos os números de post seria muito chato. Com expressões regulares podemos criar um padrão que irá coincidir com a url e extrair o número para nós: `^ post/(\d+) / $`. Vamos aos poucos ver o que estamos fazendo aqui:

- `^ post /` está dizendo ao Django para pegar tudo que tenha `post /` no início da url (logo após o `^`)
- `(\d+)` significa que haverá um número (um ou mais dígitos) e que queremos o número capturado e extraído
- `/` diz para o Django que deve seguir outro `/`
- `$` indica o final da URL significando que apenas sequências terminando com o `/` irão corresponder a esse padrão

Sua primeira url Django!

É hora de criar nossa primeira URL! Queremos `http://127.0.0.1:8000 /` para ser uma página inicial do nosso blog e exibir uma lista de posts.

Também queremos manter o arquivo de `mysite/urls.py` limpo, aí nós importaremos `urls` da nossa aplicação `blog` para o arquivo principal `mysite/urls.py`.

Vá em frente, apague as linhas comentadas (as linhas que começam com `#`) e adicione uma linha que vai importar `blog.urls` para a url principal (`'`).

O seu arquivo `mysite/urls.py` deve agora se parecer com isto:

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

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

O Django agora irá redirecionar tudo o que entra em `'http://127.0.0.1:8000 /` para `blog.urls` e procurar por novas instruções lá.

Ao escrever as expressões regulares em Python é sempre feito com `r` na frente da sequência - isso é só uma dica útil para Python que a sequência pode conter caracteres especiais que não são destinadas para Python em si, mas em vez disso são parte da expressão regular.

blog.urls

Crie um novo arquivo vazio `blog/urls.py`. Tudo bem! Adicione estas duas primeiras linhas:

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

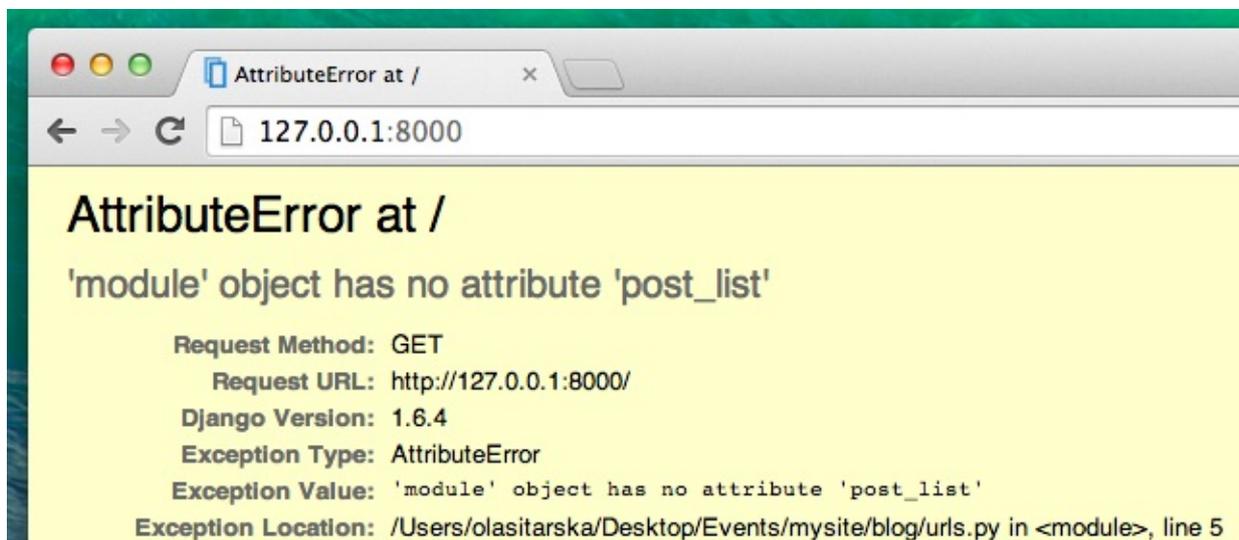
Aqui nós estamos apenas importando métodos do Django e todos os nossos `views` do aplicativo `blog` (ainda não temos nenhuma, mas teremos em um minuto!)

Depois disso nós podemos adicionar nosso primeira URL padrão:

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

Como você pode ver, estamos agora atribuindo uma `view` chamada `post_list` para `^ $` URL. Essa expressão regular corresponderá a `^` (um começo) seguido por `$` (fim) - então somente uma seqüência vazia irá corresponder. E isso é correto, porque em resolvedores de Django url, `'http://127.0.0.1:8000/'` não é uma parte da URL. Este padrão irá mostrar o Django que `views.post_list` é o lugar certo para ir, se alguém entra em seu site no endereço `'http://127.0.0.1:8000/'`.

Tudo certo? Abra <http://127.0.0.1:8000> no seu navegador pra ver o resultado.



Não tem mais "It Works!" mais ein? Não se preocupe, é só uma página de erro, nada a temer! Elas são na verdade muito úteis:

Você pode ler que não há **no attribute 'post_list'**. O `post_list` te lembra alguma coisa? Isto é como chamamos o nosso view! Isso significa que está tudo no lugar, só não criamos nossa `view` ainda. Não se preocupe, nós chegaremos lá.

Se você quer saber mais sobre Django URLconfs, veja a documentação oficial:

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

Views - hora de criar!

É hora de resolver o bug que criamos no capítulo anterior :)

Uma *view* é colocada onde nós colocamos a "lógica" da nossa aplicação. Ele irá solicitar informações a partir do `model` que você criou antes e passá-lo para um `template` que você vai criar no próximo capítulo. Views, no fundo, não passam de métodos escritos em Python que são um pouco mais complicados do que aquilo que fizemos no capítulo **Introdução ao Python**.

As views são postas no arquivo `views.py`. Nós vamos adicionar nossas *views* no arquivo `blog/views.py`.

blog/views.py

OK, vamos abrir o arquivo e ver o que tem nele:

```
from django.shortcuts import render

# Create your views here.
```

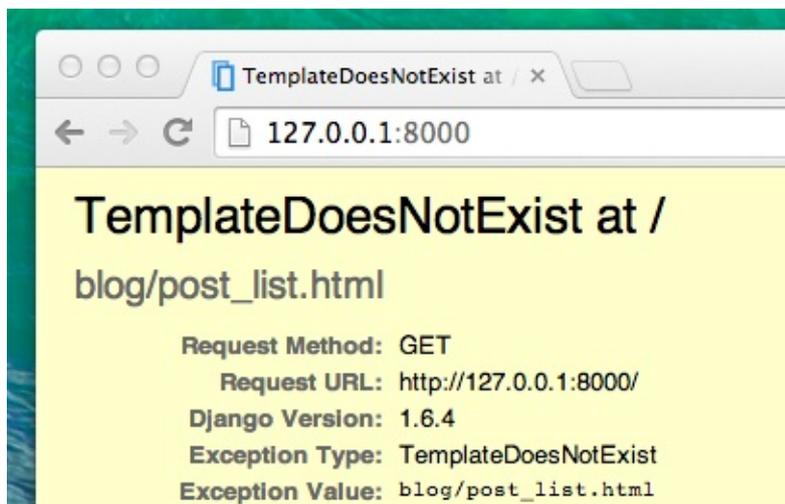
Não tem muita coisa. A *view* mais básica se parece com isto.

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

Como você pode ver, nós criamos um método (`def`) chamado `post_list` que aceita o `pedido` e `retornar` um método `render` será processado (para montar) nosso modelo `blog/post_list.html`.

Salve o arquivo, vá para <http://127.0.0.1:8000/> e veja o que temos agora.

Outro erro! Leia o que está acontecendo agora:



Esta é fácil: *TemplateDoesNotExist*. Vamos corrigir este bug e criar um modelo no próximo capítulo!

Aprenda mais sobre as views do Django lendo a documentação oficial:

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

Introdução a HTML

Você pode se perguntar: e o que é um template?

Um template é um arquivo que nós podemos reutilizar para apresentar diferentes informações de uma forma consistente. Por exemplo, você poderia usar um template para te ajudar a escrever uma carta, pois, embora cada carta possua uma mensagem e um destino diferente, todas terão sempre o mesmo formato.

O formato do template do Django é descrito em uma linguagem chamada HTML (esse é o mesmo HTML que mencionamos no primeiro capítulo **Como a Internet funciona**).

O que é HTML?

HTML é um simples código que é interpretado pelo seu navegador web - como o Chrome, o Firefox ou o Safari - para exibir uma página da web para o usuário.

HTML significa "HyperText Markup Language". **HiperText** significa que é um tipo de texto que suporta hiperlinks entre páginas. **Marcação** nada mais é que marcar um documento com códigos que dizem para alguém (nesse caso, o navegador web) como a página deverá ser interpretada. Código em HTML é feito com **tags**, cada uma começando com `<` e terminando com `>`. Essas tags marcam os **elementos**.

Seu primeiro template!

Criar um template significa criar um arquivo de template. Tudo é um arquivo, certo? Provavelmente você já deve ter notado isso.

Os templates são salvos no diretório `blog/templates`. Logo, crie um diretório chamado `templates` dentro do diretório do seu blog. Em seguida, crie outro diretório chamado `blog` dentro do diretório `templates`:

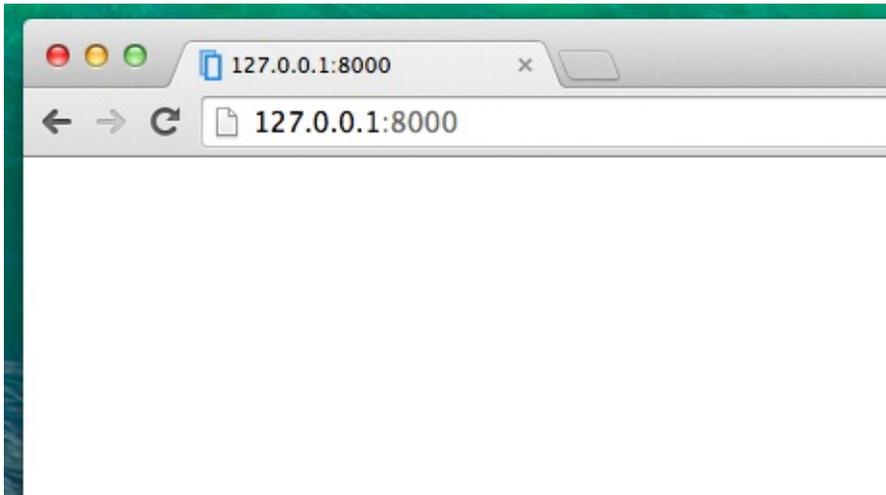
```
blog
├── templates
│   └── blog
```

(Você deve estar se perguntando porque nós precisamos de dois diretórios chamados `blog` - como você descobrirá mais para frente, essa é uma simples e útil convenção que facilita a vida quando as coisas começarem a ficar mais complicadas.)

E agora nós criamos o arquivo `post_list.html` (deixe-o em branco por agora) dentro do diretório `blog/templates/blog`.

Veja como o nosso site está se parecendo agora: <http://127.0.0.1:8000/>

Se ocorrer um erro de `TemplateDoesNotExist` tente reiniciar o seu servidor. Entre na linha de comando, pare o servidor pressionando `Ctrl+C` (Control seguido da tecla C, juntas) e reinicie-o rodando `python manage.py runserver`.

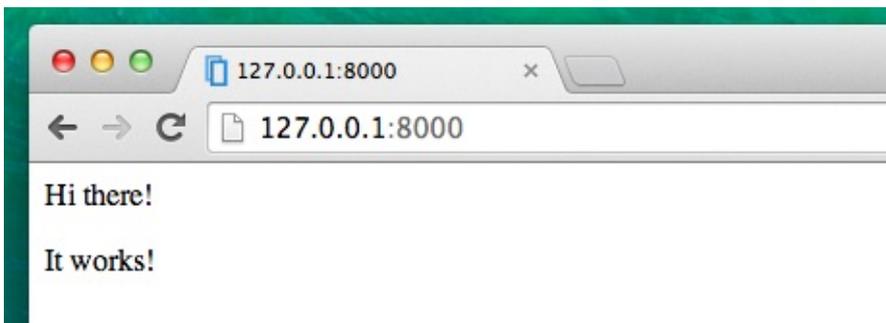


Acabaram-se os erros! Parabéns :) Entretanto, nosso site não mostra nada a não ser uma página em branco. Isso porque o nosso template está vazio. Então precisamos consertar isso.

Adicione a seguinte linha dentro do template:

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

Como nosso site se parece agora? Clique para descobrir: <http://127.0.0.1:8000/>



Funcionou! Bom trabalho :)

- A tag mais básica, `<html>`, estará sempre no começo de qualquer página da web, assim como, `</html>` sempre estará no fim. Como você pode ver, todo o conteúdo de um website se encontra entre a tag de início `<html>` e entre a tag de fim `</html>`
- `<p>` é a tag que denomina parágrafos; `</p>` determina o fim de cada parágrafo

Head & body

Cada página HTML também é dividida em dois elementos: **head** (cabeça) e **body** (corpo).

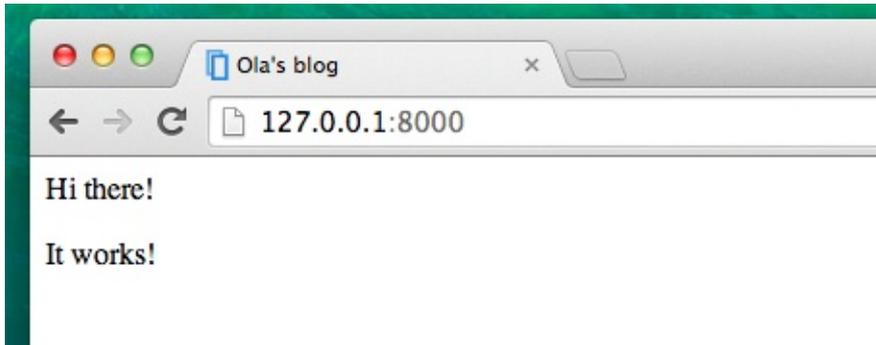
- **head** é um elemento que contém informações sobre o documento que não são mostradas na tela.
- **body** é um elemento que contém tudo o que é exibido como parte de uma página de um site.

Nós usamos a tag `<head>` para dizer ao navegador sobre as configurações da página. Por sua vez, a tag `<body>` diz ao navegador o que há de verdade na página.

Por exemplo, você pode por o elemento título de uma página web dentro da tag `<head>`. Veja:

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

Salve o arquivo e atualize sua página.



Viu como o navegador entendeu que "Ola's blog" é o título da página? Ele interpretou `<title>Ola's blog</title>` e colocou o texto na barra de título do seu navegador (e também será usado para os favoritos e outras coisas mais).

Provavelmente você já deve ter notado que cada tag de abertura casa com uma *tag de fechamento*, com uma `/`, e que os elementos estão *aninhados* (ex.: você não pode fechar um tag em particular antes que todas as outras tags que estiverem dentro dela já estejam fechadas).

É como colocar coisas dentro de caixas. Você tem uma grande caixa, `<html></html>`; dentro dela há `<body></body>`, sendo que esta ainda contém caixas menores: `<p></p>`.

Você precisa seguir essas regras de *fechamento* de tags, e de *aninhamento* de elementos - se você não fizer isso, o navegador poderá não estar apto para interpretar seu código de maneira correta e sua página será exibida de maneira incorreta.

Customize seu template

Agora você pode se divertir um pouco tentando customizar o seu template! Aqui estão algumas tags úteis para isso:

- `<h1>Um título</h1>` - para o título mais importante
- `<h2>Um sub-título</h2>` para um título um nível abaixo
- `<h3>Um sub-sub-título</h3>` ... e por aí vai, até `<h6>`
- `texto` enfatiza seu texto
- `text` enfatiza fortemente seu texto
- `
` pula para a próxima linha (você não pode colocar nada dentro de `br`)
- `link` cria um link
- `primeiro itemsegundo item` cria uma lista, exatamente como essa!
- `<div></div>` define uma seção da página

Aqui está um exemplo de um template completo:

```

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

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

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

```

Nós criamos três seções `div` aqui.

- O primeiro elemento `div` possui o título do nosso blog - é um título e um link
- Os outros dois elementos `div` possuem nossas postagens com a data de publicação, `h2` com o título da postagem que é clicável e dois `p`s (parágrafos) de texto, um para a data e outro para o texto da postagem.

Isso nos dá o seguinte efeito:



Yaaay! Mas, até agora, nosso template mostra exatamente **sempre a mesma informação** - sendo que, anteriormente, nós falávamos sobre templates como uma maneira para exibir informações **diferentes** em um **mesmo formato**.

O que nós realmente queremos fazer é exibir postagens reais que foram adicionadas no Django admin - e isso é o que faremos em seguida.

Mais uma coisa: deploy!

Seria bom ver tudo isto na Internet, certo? Vamos fazer outro deploy PythonAnywhere:

Commit, e ponha seu código no GitHub

Primeiro de tudo, vejamos quais arquivos foram alterados desde a última implantação:

```
$ git status
```

Verifique se você está no diretório `djangogirls` e vamos dizer ao `git` para incluir todas as mudanças dentro deste diretório:

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

Nota `-A` (abreviação de "all", tudo em inglês) significa que o `git` também reconhecerá se você deletou algum arquivo (por padrão, o `git` apenas reconhece arquivos criados/modificados). Lembre-se também (do capítulo 3) que `.` significa o diretório atual.

Antes de nós fazermos o upload de todos os arquivos, chequemos o que o `git` enviará (todos os arquivos que o `git` for enviar deverá aparecer em verde):

```
$ git status
```

Estamos quase lá! Agora é hora de dizer a ele para salvar essa modificação em seu histórico. Nós daremos a ele uma "mensagem de commit" onde nós descrevemos as modificações que fizemos. Você pode escrever o que você quiser agora, mas será mais útil se você escrever alguma coisa mais descritiva, algo para você poder se lembrar das coisas que você fez futuramente.

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

Certifique-se que você usou aspas duplas para delimitar a mensagem do commit.

Quando fizermos isso, nós fazemos upload (envio) das nossas mudanças para o PythonAnywhere:

```
git push
```

Bote seu novo código no PythonAnywhere e recarregue o seu aplicativo da web

- Abra a [página de consoles de PythonAnywhere](#) e vá para o seu **console Bash** (ou começar um novo). Em seguida, execute:

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

Eveja seu código sendo baixado. Se você deseja verificar se já chegou, pode ir para a **Files tab** e ver seu código no PythonAnywhere.

- Finalmente, pule para a [Web tab](#) e aperte **Reload** em seu aplicativo web.

Sua atualização deve estar live! Vá em frente e atualize seu site no navegador. As alterações devem ser visíveis :)

QuerySets e ORM do Django

Neste capítulo você vai aprender como Django se conecta ao banco de dados e como ele armazena dados. Vamos nessa!

O que é um QuerySet?

Um QuerySet (conjunto de pesquisa), no fundo, é uma lista de objetos de um dado modelo. Um QuerySet permite que você leia os dados do banco, filtre e ordene o mesmo.

É mais fácil aprender por exemplos. Vamos tentar?

O Shell do Django

Abra o terminal e digite:

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

O resultado deve ser:

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

Agora você está no console interativo do Django. Ele é como o prompt do Python só que com umas mágicas a mais :). Você pode usar todos os comandos do Python aqui também, é claro.

Todos os objetos

Antes, vamos tentar mostrar todas as nossas postagens. Podemos fazer isso com o seguinte comando:

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

Oops! Um erro apareceu. Ele nos diz que não existe algo chamado Post. É verdade -- nós esquecemos de importá-lo primeiro!

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

Isso é simples: importamos o modelo `Post` de dentro do `blog.models`. Vamos tentar mostrar todas as postagens novamente:

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

É uma lista dos posts que criamos anteriormente! Criamos esses posts usando a interface de administração do Django. No entanto, agora queremos criar novas mensagens utilizando o python, então como é que fazemos isso?

Criando um objeto

É assim que você cria um objeto Post no banco de dados:

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

Mas aqui temos um ingrediente que faltava: `me`. Precisamos passar uma instância de `User` modelo como autor. Como fazer isso?

Primeiro vamos importar o modelo `User`:

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

Quais usuários temos no nosso banco de dados? Experimente isso:

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

É o superusuário que criamos anteriormente! Vamos obter uma instância de usuário agora:

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

Como você pode ver, nós agora usamos um `get` a `User` with a `username` igual a 'ola'. Claro, você tem que adaptar a seu nome de usuário.

Agora finalmente podemos criar nossa primeira postagem:

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

Viva! Quer ver se funcionou?

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

Adicione mais postagens

Agora, você pode se divertir um pouco e adicionar mais postagens para ver como funciona. Adicione mais 2-3 e siga para a próxima parte.

Filtrar objetos

Uma grande parte de QuerySets é a habilidade de filtrá-los. Digamos que queremos encontrar todos as postagens escritas pelo usuário `ola`. Nós usaremos o `filter` em vez de `all` em `Post.objects.all()`. Entre parênteses indicamos que as condições precisam ser atendidas por um postagem de blog para acabar em nosso queryset. Em nosso caso é `author` que é igual a `me`. A maneira de escrever isso no Django é: `author=me`. Agora o nosso trecho de código parece com este:

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

Ou talvez nós queremos ver todos os posts que contenham a palavra 'title' no campo de `title` ?

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

Nota Existem dois caracteres de sublinhado (`_`) entre o `title` e `contains`. Django ORM usa esta sintaxe para separar nomes de campo ("title") e operações ou filtros ("contains"). Se você usar apenas um sublinhado, você obterá um erro como "FieldError: Cannot resolve keyword title_contains".

Você também pode obter uma lista de todos os posts publicados. Fazemos isso filtrando todos os posts com

`published_date` definido no passado:

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

Infelizmente, nenhum dos nossos posts estão publicados ainda. Nós podemos mudar isso! Primeiro obtenha uma instância de um post que queremos publicar:

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

E então publicá-lo com o nosso método de `publish` !

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

Agora tente obter a lista de posts publicados novamente (pressione a seta para cima botão 3 vezes e tecla Enter):

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

Ordenando objetos

Um QuerySet também nos permite ordenar a lista de objetos. Vamos tentar ordenar as postagens pelo campo

`created_date` :

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

Você também pode inverter a ordem adicionando `-` no início:

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

Legal! Você já está pronto para a próxima parte! Para fechar o terminal digite:

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

Django Querysets

Nós temos diferentes peças aqui: o model `Post` está definido em `models.py`, nós temos `post_list` no `views.py` e o template adicionado. Mas como nós faremos de fato para fazer com que as nossas postagens apareçam no nosso template em HTML? Porque é isso que nós queremos: pegar algum conteúdo (models salvos no banco de dados) e exibi-lo de uma maneira bacana no nosso template, certo?

E isso é exatamente o que as `views` devem fazer: conectar models e templates. Na nossa view `post_list view` nós vamos precisar pegar os models que queremos exibir e passá-los para o template. Então, basicamente, em uma `view` nós decidimos o que (um model) será exibido no template.

Certo, e como nós faremos isso?

Precisamos abrir o nosso `blog/views.py`. Até agora a `view post_list` se parece com isso:

```
from django.shortcuts import render

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

Lembra quando falamos sobre a inclusão de código escrito em arquivos diferentes? Agora é o momento em que temos de incluir o model que temos escrito em `models.py`. Vamos adicionar esta linha `from .models import Post` como este:

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

O ponto depois de `from` significa o *diretório atual* ou o *aplicativo atual*. Como `views.py` e `models.py` estão no mesmo diretório podemos simplesmente usar `.` e o nome do arquivo (sem `.py`). Então nós importamos o nome do modelo (`Post`).

E o que vem agora? Para pegar os posts reais do model `Post` nós precisamos de uma coisa chamada `QuerySet`.

QuerySet

Você já deve estar familiarizado com o modo que os QuerySets funcionam. Nós conversamos sobre isso no [capítulo ORM do Django \(QuerySets\)](#). Agora nós estamos interessados em uma lista de posts que são publicados e classificados por `published_date`, certo? Nós já fizemos isso no capítulo QuerySets!

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

Agora nós colocamos este pedaço de código dentro do arquivo `blog/views.py` adicionando-o à função `def post_list(request):`:

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

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

Note que criamos uma *variável* para nosso o QuerySet: `posts`. Trate isto como o nome do nosso QuerySet. De agora em diante nós podemos nos referir a ele por este nome.

A última parte que falta é passar o QuerySet `posts` para o template (veremos como exibi-lo em um próximo capítulo).

Na função `render` já temos o parâmetro `request` (tudo o que recebemos do usuário através da Internet) e um arquivo de template `'blog/post_list.html'`. O último parâmetro, que se parece com isso: `{}` é um lugar em que podemos acrescentar algumas coisas para que o template use. Precisamos nomeá-los (ficaremos com `'posts'` por enquanto :)). Deve ficar assim: `{'posts': posts}`. Observe que a parte antes de `:` está entre aspas `' '`.

Então finalmente nosso arquivo `blog/views.py` deve se parecer com isto:

```
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})
```

Feito! Hora de voltar para o nosso template e exibir essa QuerySet!

Se quiser ler mais sobre QuerySets no Django você deve dar uma olhada aqui:

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

Templates

Hora de exibir algum dado! Django nos dá **tags de templates** embutidas bastante úteis para isso.

O que são tags de template?

Como pode ver, você não pode colocar código Python no HTML, porque os navegadores não irão entender. Eles apenas conhecem HTML. Nós sabemos que HTML é bastante estático, enquanto Python é muito mais dinâmico.

Tags de template Django nos permite transformar objetos Python em código HTML, para que você possa construir sites dinâmicos mais rápido e mais fácil. Uhuu!

Modelo de lista de post de exibição

No capítulo anterior, nós fornecemos ao nosso template uma lista de postagens e a variável `posts`. Agora vamos exibir em nosso HTML.

Para exibir uma variável no Django template, nós usamos colchetes duplos com o nome da variável dentro, exemplo:

```
{{ posts }}
```

Tentar fazer isso no seu template `blog/templates/blog/post_list.html` (substitua o segundo e o terceiro par de tags `< div >>` / `div >` pela linha `{{ posts }}`), salve o arquivo e atualize a página para ver os resultados:



Você pode ver, tudo que temos é:

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

Isto significa que o Django a entende como uma lista de objetos. Lembre-se de **introdução ao Python** como podemos exibir listas? Sim, com os loops! Em um template Django, fazemos isso da seguinte maneira:

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

Tente fazer isso no seu template.

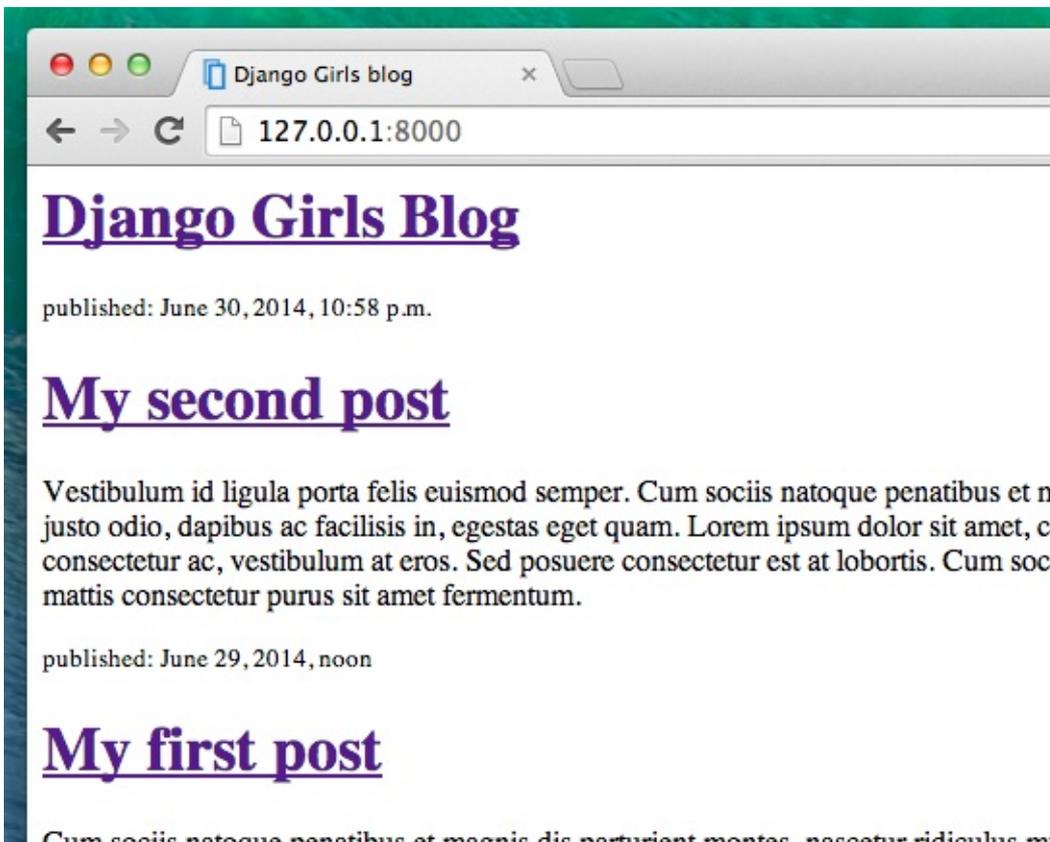


Funciona! Mas nós queremos que eles sejam exibidos como os posts estáticos, como os que criamos anteriormente no capítulo de **Introdução a HTML**. Nós podemos misturar HTML com tags de template. O conteúdo da tag `body` ficará assim:

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

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

Tudo que você põe entre `{% for %}` e `{% endfor %}` será repetido para cada objeto na lista. Atualize sua página:



Você notou que dessa vez nós usamos uma notação um pouco diferente `{{ post.title }}` ou `{{ post.text }}`? Nós estamos acessando os dados em cada um dos campos que definimos no modelo do `Post`. Além disso, `|linebreaksbr` está passando o texto do post por um filtro, convertendo quebras de linha em parágrafos.

Mais uma coisa

Seria bom ver se seu site ainda estará funcionando na internet, certo? Vamos tentar implantar a PythonAnywhere novamente. Aqui está um resumo dos passos...

- Primeiro, envie seu código para o Github

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

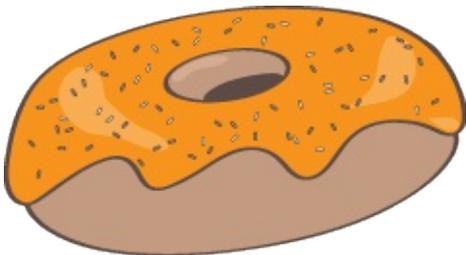
- Em seguida, faça login em [PythonAnywhere](#) e vá para seu **Bash console** (ou comece um novo) e execute:

```
` $ cd my-first-blog $ git pull
```

- Finalmente, pule para a [Web tab](#) e aperte **Reload** em seu aplicativo web. Sua atualização deve estar live!

Parabéns! Agora vá em frente e tente adicionar um novo post em seu Django admin (Lembre-se de adicionar `published_date!`), em seguida, atualize a página para ver se o post aparece por lá.

Funciona como mágica? Estamos orgulhosos! Afaste-se do seu computador um pouco, você ganhou uma pausa. :)



CSS - Deixe mais bonito!

Nosso blog ainda parece feio, certo? Está na hora de deixar ele melhor! Para isso nós usaremos o CSS.

O que é CSS?

Do inglês "Cascading Style Sheets", CSS é uma linguagem usada para descrever o aspecto e a formatação de um website escrito numa linguagem de marcação (como HTML). Imagine ele como sendo um tipo de "maquiagem" para nosso site ;).

Mas nós não queremos iniciar do zero de novo, certo? Nós tentaremos, mais uma vez, usar algo que foi feito e disponibilizado de graça por programadores na internet. Você sabe, reinventar a roda não é nada divertido.

Vamos usar o Bootstrap!

Bootstrap é um dos mais famosos e populares frameworks de HTML e CSS para desenvolver sites bonitos:

<https://getbootstrap.com/>

Foi escrito por programadores que trabalharam no Twitter e agora é desenvolvido por voluntários de todo o mundo.

Instalar Bootstrap

Para instalar o Bootstrap, você precisa adicionar ao seu cabeçalho (na tag `<head>` dentro do seu arquivo `.html`) (`blog/templates/blog/post_list.html`):

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

Isso não adiciona nenhum arquivo ao seu projeto. O código apenas aponta para arquivos que existem na internet. Apenas siga em frente, abra seu site e atualize a página. Aqui ele está!



Já parecendo melhor!

Arquivos estáticos no Django

Finalmente nós teremos um olhar mais atento nessas coisas que chamamos **arquivos estáticos**. Arquivos estáticos são todas as suas imagens e arquivos CSS -- arquivos que não são dinâmicos, então seu conteúdo não depende do contexto da requisição e será o mesmo para todos os usuários.

Onde colocar os arquivos estáticos para Django

Como você viu quando rodamos `collectstatic` no servidor, Django já sabe onde encontrar os arquivos estáticos para o built-in "admin" app. Agora só precisamos adicionar alguns arquivos estáticos para nosso próprio app, `blog`.

Fazemos isso através da criação de uma pasta chamada `static` dentro do aplicativo do blog:

```
djangoirls
├── blog
│   ├── migrations
│   └── static
└── mysite
```

Django encontrará automaticamente todas as pastas chamadas "static" dentro de qualquer uma das pastas dos seus apps, e será capaz de usar seu conteúdo como arquivos estáticos.

Seu primeiro arquivo CSS!

Vamos criar um arquivo CSS agora, para adicionar seu próprio estilo para sua página da web. Crie um novo diretório chamado `css` dentro de seu diretório `static`. Em seguida, crie um novo arquivo chamado `blog.css` dentro do diretório `css`. Pronto?

```
djangoirls
├── blog
│   └── static
│       ├── css
│       └── blog.css
```

Hora de escrever CSS! Abra o arquivo `static/css/blog.css` no seu editor de código.

Não vamos nos aprofundar muito em customizar e aprender sobre CSS aqui, porque é bem fácil e você pode aprender no seu próprio após este workshop. Recomendamos fortemente fazer este [Codecademy HTML & CSS course](#) para aprender tudo o que você precisa saber sobre como tornar seus sites mais bonitos com CSS.

Mas vamos fazer pelo menos um pouco. Talvez possamos mudar a cor do nosso cabeçalho? Para entender as cores, computadores usam códigos especiais. Eles começam com `#` e são seguidos por 6 letras (A-F) e números (0-9). Você pode encontrar exemplos de códigos de cores aqui: <http://www.colorpicker.com/>. Você pode também usar [cores predefinidas](#), como `red` e `green`.

Em seu arquivo `static/css/blog.css` você deve adicionar o seguinte código:

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

`h1 a` é um seletor de CSS. Isso significa que nós estamos aplicando nossos estilos para qualquer elemento `a` dentro de um elemento `h1` (i.e. quando tivermos no código algo como: `<h1>link</h1>`). Neste caso nós estamos dizendo para mudar a cor para `#FCA205`, que é laranja. Claro, você pode colocar a cor que você quiser aqui!

Em um arquivo CSS podemos determinar estilos para elementos no arquivo HTML. Os elementos são identificados pelo nome do elemento (ou seja, `a`, `h1`, `body`), o atributo de `class` ou o atributo `id`. Classe e id são nomes que você mesmo dá ao elemento. Classes definem grupos de elementos, e ids apontam para elementos específicos. Por exemplo, a

seguinte tag pode ser identificada por CSS usando a tag de nome `a`, a classe `link_externo` ou a identificação de `link_para_a_pagina_wiki`:

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

Leia sobre [Seletores CSS em w3schools](#).

Então, precisamos também contar o nosso template HTML que nós adicionamos CSS. Abra o arquivo

`blog/templates/blog/post_list.html` e adicione essa linha no início do mesmo:

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

Estamos apenas carregando arquivos estáticos aqui :). Depois, entre o `<head>` e `</head>`, depois dos links para os arquivos de CSS do Bootstrap (o navegador lê os arquivos na ordem que eles são dados, então o código em nosso arquivo pode substituir o código em arquivos de inicialização), adicione esta linha:

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

Só dissemos que nosso modelo onde se encontra nosso arquivo CSS.

Agora, seu arquivo deve ficar assim:

```
{% load staticfiles %}
<html>
  <head>
    <title>Django Girls blog</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
    <link rel="stylesheet" href="{% static 'css/blog.css' %}">
  </head>
  <body>
    <div>
      <h1><a href="/">Django Girls Blog</a></h1>
    </div>

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

OK, salve o arquivo e atualize o site!



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

My second post

Vestibulum id ligula porta felis euismod semper. Cum sociis natoque

Bom trabalho! Talvez a gente também queira dar um pouco de ar ao nosso site e aumentar a margem do lado esquerdo? Vamos tentar!

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

Adicione isto ao seu arquivo CSS, salve e veja como ele funciona!



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

My second post

Vestibulum id ligula porta felis euismod semper. Cum sociis natoque

Talvez a gente possa customizar a fonte no nosso cabeçalho? Cole na seção `<head>` do arquivo `blog/templates/blog/post_list.html` o seguinte:

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

Essa linha irá importar uma fonte chamada *Lobster* do Google Fonts (<https://www.google.com/fonts>).

Agora adicione a linha `font-family: 'Lobster';` no CSS do arquivo `static/css/blog.css` dentro do bloco de declaração `h1` a (o código entre as chaves `{ e }`) e atualize a página:

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



Incrível!

Como mencionado acima, CSS usa o conceito de classes, que basicamente permite que você nomeie parte do código HTML e aplique estilos apenas à esta parte, sem afetar as outras. É super útil se você tiver duas divs, mas eles estão fazendo algo muito diferente (como o seu cabeçalho e seu post), então você não quer que eles fiquem parecidos.

Vá em frente e o nomeie algumas partes do código HTML. Adicione uma classe chamada de `page-header` para o `div` que contém o cabeçalho, assim:

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

E agora, adicione uma classe `post` em sua `div` que contém um post de blog.

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

Agora adicionaremos blocos de declaração de seletores diferentes. Seletores começando com `.` se referem às classes. Existem muitos tutoriais e explicações sobre CSS na Web para ajudar você a entender o código a seguir. Por enquanto, basta copiar e colá-lo em seu arquivo `mysite/static/css/blog.css` :

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

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

.content {
  margin-left: 40px;
}

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

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

.save {
  float: right;
}

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

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

.post {
  margin-bottom: 70px;
}

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

Então envolva o código HTML que exibe as mensagens com declarações de classes. Substitua isto:

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

no arquivo `blog/templates/blog/post_list.html` por isto:

```

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

```

Salve esses arquivos e atualize seu site.



Uhuu! Ficou incrível, né? O código que nós acabamos de colar não é tão difícil de entender e você deve ser capaz de entender a maior parte apenas lendo.

Não tenha medo de mexer um pouco com esse CSS e tentar mudar algumas coisas. Se você quebrar alguma coisa, não se preocupe, você sempre pode desfazê-lo!

De qualquer forma, recomendamos que faça esse curso on-line [Codecademy HTML & CSS Course](#) como dever de casa pós-workshop para aprender tudo o que você precisa saber sobre como tornar seus sites mais bonitos com CSS.

Pronto para o próximo capítulo?! :)

Estendendo os templates

Outra coisa boa que o Django tem pra você é o **template extending**. O que isso significa? Isso significa que você pode usar as mesmas partes do seu HTML em diferentes páginas do seu site.

Dessa forma você não precisa ficar se repetindo em cada arquivo quando quiser usar a mesma informação/layout. E se você quiser mudar alguma coisa não precisa fazer isso em todo template, só uma vez!

Criar template base

Um template base é o template mais básico que você estenderá em cada página do seu site.

Vamos criar um arquivo `base.html` na pasta `blog/templates/blog/`:

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

Abra-o e copie tudo que está no arquivo `post_list.html` para `base.html`, desse jeito:

```
{% load staticfiles %}
<html>
  <head>
    <title>Django Girls blog</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
    <link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext" rel="stylesheet" type="text/css" >
    <link rel="stylesheet" href="{% static 'css/blog.css' %}">
  </head>
  <body>
    <div class="page-header">
      <h1><a href="/">Django Girls Blog</a></h1>
    </div>

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

Então em `base.html`, substitua todo seu `<body>` (tudo entre `<body>` e `</body>`) com isso:

```
<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>
```

Basicamente nós substituímos tudo entre `{% for post in posts %}{% endfor %}` por:

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

O que isso significa? Você acabou de criar um `block` (bloco), que é uma tag de template que te permite inserir HTML neste bloco em outros templates que estendem `base.html`. Nós vamos te mostrar como fazer isso já.

Salve e abra o arquivo `blog/templates/blog/post_list.html` novamente. Apague exatamente tudo que não estiver dentro da tag `body` e apague também `<div class="page-header"></div>`, de forma que o arquivo fique da seguinte maneira:

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

Agora adicione esta linha ao início do arquivo:

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

Isso significa que, agora, nós estamos estendendo o template `base.html` em `post_list.html`. Uma última coisa: colocar tudo (exceto pela linha que acabamos de adicionar) entre `{% block content %}` e `{% endblock content %}`. Como a seguir:

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

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

É isso! Veja se o seu site ainda está funcionando direito :)

Se ocorrer um erro de `TemplateDoesNotExist`, que diz que não existe nenhum arquivo chamado `blog/base.html` e se você tiver o `runserver` executando no terminal, tenta interrompê-lo (pressionando `Ctrl+C` - o botão Control mais o botão C juntos) e reinicie ele rodando o comando `python manage.py runserver`.

Amplie sua aplicação

Já concluímos todos os passos necessários para a criação do nosso site: sabemos como criar um modelo, uma url, uma view e um template. Também sabemos como melhorar a aparência do nosso website.

Hora de praticar!

A primeira coisa que precisamos no nosso blog é, obviamente, uma página para mostrar uma postagem, certo?

Já temos um modelo de `Post`, então não precisamos adicionar nada ao `models.py`.

Criar um link no template

Vamos começar com a adição de um link dentro do arquivo `blog/templates/blog/post_list.html`. Neste momento ele deve se parecer com:

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

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

Queremos ter um link para uma página de detalhe no título do post. Vamos transformar `< h1 >< href = "" >{{ post.title }} < /a >< / h1 >` em um link:

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

Tempo para explicar o misterioso `{% url 'blog.views.post_detail' pk=post.pk %}`. Como você pode suspeitar, a notação de `{% %}` significa que estamos usando as tags de template do Django. Desta vez vamos usar uma que vai criar uma URL para nós!

`blog.views.post_detail` é um caminho para um `post_detail` *Vista* que queremos criar. Preste atenção: `blog` é o nome da sua aplicação (o diretório `blog`), `views` vem do nome do arquivo `views.py` e, a última parte - `post_detail` - é o nome da *view*.

Agora quando formos para: <http://127.0.0.1:8000/> teremos um erro (como esperado, já que não temos uma URL ou uma *view* para `post_detail`). Vai se parecer com isso:



Vamos criar a URL em `urls.py` para a nossa `post_detail` *view*!

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

Queremos criar uma URL para guiar o Django para a *view* chamada `post_detail`, que irá mostrar um post completo do blog. Adicione a linha `url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail)`, ao arquivo `blog/urls.py`. Deve ficar assim:

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

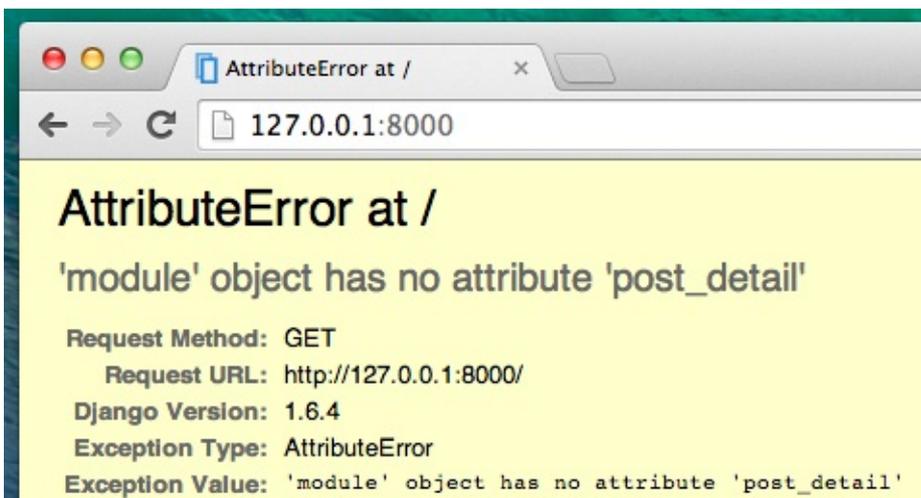
urlpatterns = [
    url(r'^$', views.post_list),
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail),
]
```

Parece assustador, mas não se preocupe - vamos explicar ele para você: - começa com `^` de novo... "o início" - `post /` significa apenas que após o começo, da URL deve ter a palavra **post** e `/`. Até aqui, tudo bem. - `(?P<pk>[0-9]+)` - essa parte é mais complicada. Isso significa que o Django vai levar tudo que você colocar aqui e transferir para uma *view* como uma variável chamada `pk`. `[0-9]` também nos diz que só pode ser um número, não uma letra (tudo entre 0 e 9). `+` significa que precisa existir um ou mais dígitos. Então algo como `http://127.0.0.1:8000/post//` não é válido, mas `http://127.0.0.1:8000/post/1234567890/` é perfeitamente ok! - `/` - então precisamos de `/` outra vez - `$` - "o fim"!

Isso significa que se você digitar `http://127.0.0.1:8000/post/5/` em seu navegador, Django vai entender que você está procurando uma *view* chamada `post_detail` e transferir a informação de que `pk` é igual a `5` para aquela *view*.

`pk` é uma abreviação para `primary key` (chave primária). Esse nome geralmente é usado nos projetos feitos em Django. Mas você pode dar o nome que quiser às variáveis (lembre-se: minúsculo e `_` ao invés de espaços em branco!). Por exemplo em vez de `(?P<pk>[0-9]+)` podemos ter uma variável `post_id`, então esta parte ficaria como: `(?P<post_id>[0-9]+)`.

Razoável! Vamos atualizar a página: <http://127.0.0.1:8000/> / Boom! Ainda outro erro! Como esperado!



Você se lembra qual é o próximo passo? Claro: adicionando uma view!

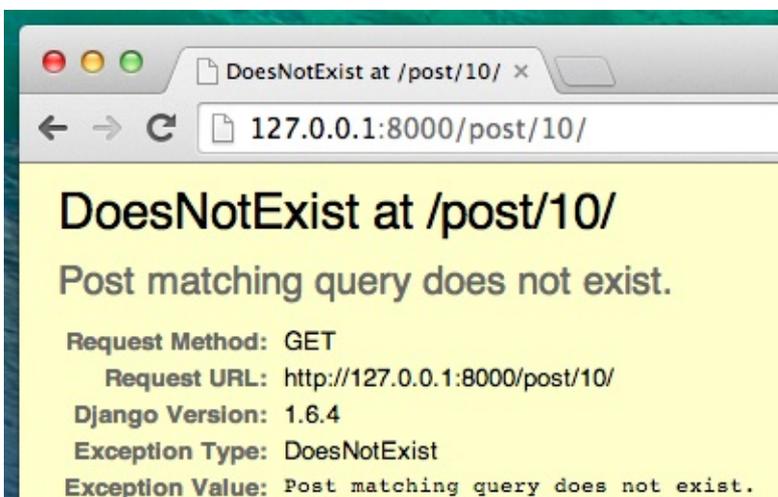
post_detail view

Desta vez a nossa *view* recebe um parâmetro extra `pk`. Nossa *view* precisa pegá-la, certo? Então vamos definir nossa função como `def post_detail (request, pk):`. Observe que precisamos usar exatamente o mesmo nome que especificamos em urls (`pk`). Omitir essa variável é errado e resultará em um erro!

Agora queremos receber apenas um post do blog. Para isso podemos usar *querysets* como este:

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

Mas este código tem um problema. Se não houver nenhum `Post` com a *chave primária* (`pk`) fornecida teremos um erro horróroso!



Não queremos isso! Mas, claro, o Django vem com algo que vai lidar com isso para nós: `get_object_or_404`. Caso não haja nenhum `Post` com o dado `pk` exibirá uma página muito mais agradável (chamada `Page Not Found 404` - página não encontrada).



A boa notícia é que você realmente pode criar sua própria página de `Page not found` e torná-lo tão bonita quanto você quiser. Mas isso não é super importante agora, então nós vamos ignorá-la.

Ok, hora de adicionar uma `view` ao nosso arquivo `views.py` !

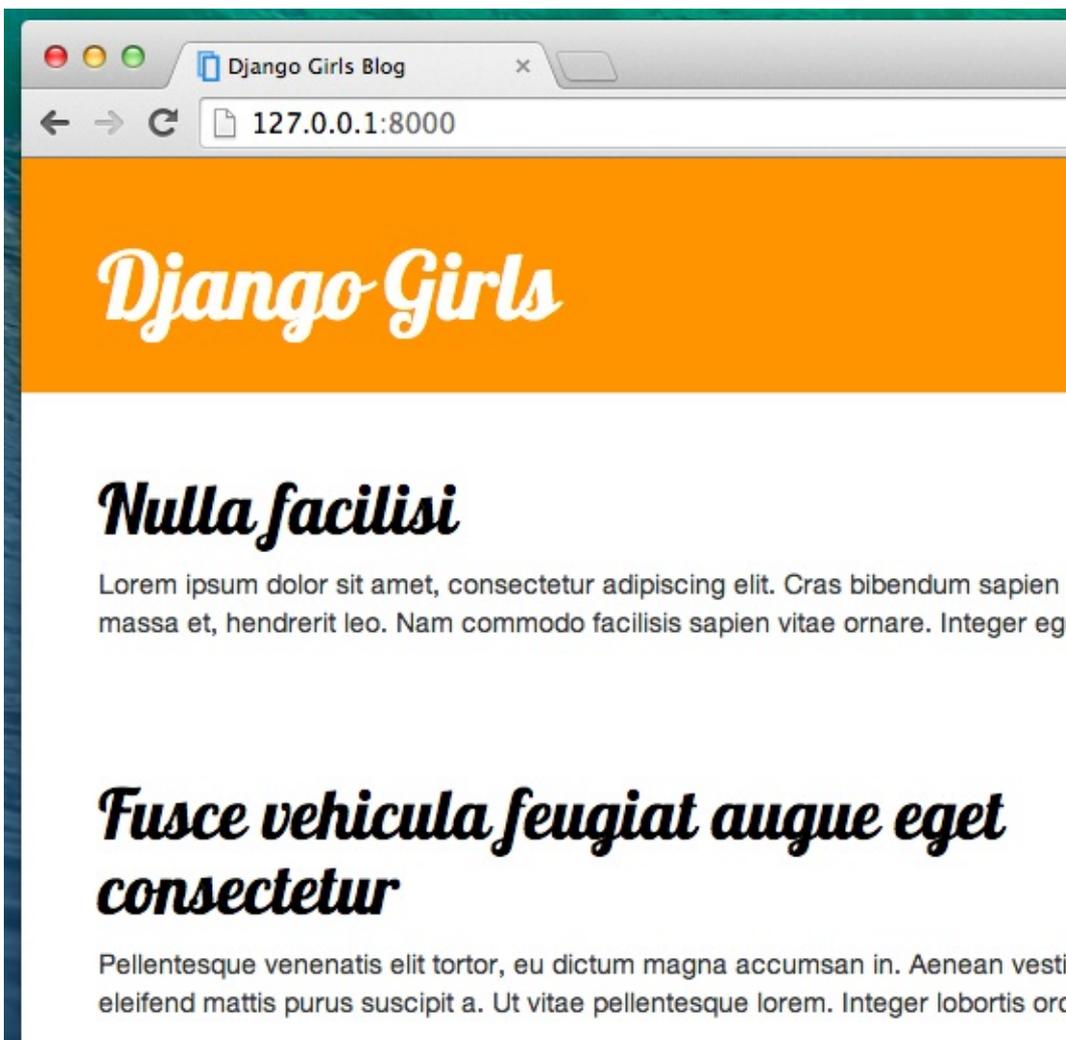
Devemos abrir `blog/views.py` e adicionar o seguinte código:

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

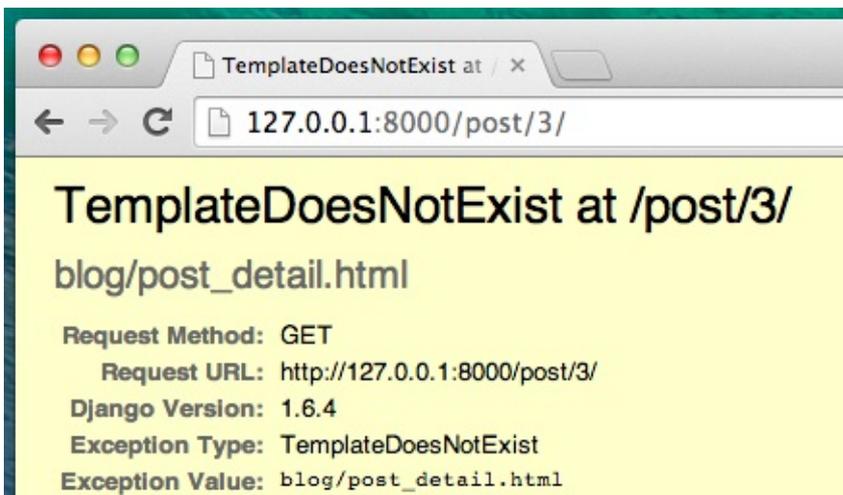
Perto de outras linhas `from` . E no final do arquivo, adicionaremos a nossa `view`:

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

Sim. Está na hora de atualizar a página: <http://127.0.0.1:8000/>



Funcionou! Mas o que acontece quando você clica em um link no título do post do blog?



Ah não! Outro erro! Mas nós já sabemos como lidar com isso, né? Precisamos adicionar um template!

Vamos criar um arquivo em `blog/templates/blog` chamado `post_detail.html`.

Será algo parecido com isto:

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

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

Mais uma vez estamos estendendo `base.html`. No bloco de `content` queremos exibir o `published_date` (data de publicação) do post (se houver), título e texto. Mas devemos discutir algumas coisas importantes, certo?

`{% if ... %} ... {% endif %}` é uma tag de template que podemos usar quando queremos verificar algo (Lembre-se `if ... else...` do **capítulo introdução ao Python?**). Neste cenário, queremos verificar se `published_date` de um post não está vazia.

Ok, podemos atualizar nossa página e ver se `Page not found` já se foi.



Yay! Funcional!

Mais uma coisa: hora de implantar!

Seria bom ver se seu site ainda estará trabalhando em PythonAnywhere, certo? Vamos tentar fazer deploy novamente.

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

- Então, em um [console PythonAnywhere Bash](#):

```
$ cd my-first-blog  
$ git pull
```

- Finalmente, pule para a [Web tab](#) e aperte **Reload**.

E deve ser isso! Parabéns :)

Formulários

Por último queremos uma forma legal de adicionar e editar as postagens do nosso blog. A ferramenta de administração do Django é legal, mas ela é um pouco difícil de customizar e de deixar mais bonita. Se usarmos formulários teremos controle absoluto sobre nossa interface - podemos fazer qualquer coisa que imaginarmos!

Uma coisa legal do Django é que nós podemos tanto criar um formulário do zero como podemos criar um `ModelForm` que salva o resultado do formulário para um determinado modelo.

Isso é exatamente o que nós queremos fazer: criaremos um formulário para o nosso modelo `Post`.

Assim como toda parte importante do Django, forms tem seu próprio arquivo: `forms.py`.

Precisamos criar um arquivo com este nome dentro da pasta `blog`.

```
blog
├── forms.py
```

Ok, vamos abri-lo e escrever nele o seguinte:

```
from django import forms

from .models import Post

class PostForm(forms.ModelForm):

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

Primeiro precisamos importar o módulo de formulários do Django (`from django import forms`) e, obviamente, nosso modelo `Post` (`from .models import Post`).

`PostForm`, como você já deve suspeitar, é o nome do nosso formulário. Precisamos dizer ao Django que este formulário é um `ModelForm` (assim o Django pode fazer a mágica pra gente) - o `forms.ModelForm` é o responsável por isso.

Segundo, nós temos a classe `Meta` onde dizemos ao Django qual modelo deveria ser usado para criar este formulário (`model = Post`).

Finalmente, nós podemos dizer qual(is) campo(s) deveriam entrar em nosso formulário. Nesse cenário nós queremos apenas o `title` e `text` para ser exposto - `author` deveria ser a pessoa que está logada no sistema (nesse caso, você!) e `created_date` deveria ser setado automaticamente quando nós criamos um post (no código), correto?

E é isso aí! Tudo o que precisamos fazer agora é usar o formulário em uma *view* e mostrá-lo em um *template*.

Então, mais uma vez, nós iremos criar: um link para a página, uma URL, uma *view* e um *template*.

Link para a página com o formulário

É hora de abrir `blog/templates/blog/base.html`. Nós iremos adicionar um link em `div` nomeado `page-header`:

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

Note que nós queremos chamar nossa nova *visão* `post_new`.

Depois de adicionar a linha, seu html deve se parecer com isso:

```
{% load staticfiles %}
<html>
  <head>
    <title>Django Girls blog</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
    <link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext" rel="stylesheet" type="text/css"
  >
    <link rel="stylesheet" href="{% static 'css/blog.css' %}">
  </head>
  <body>
    <div class="page-header">
      <a href="{% url 'blog.views.post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span>
    </a>
    <h1><a href="/">Django Girls Blog</a></h1>
  </div>
  <div class="content container">
    <div class="row">
      <div class="col-md-8">
        {% block content %}
        {% endblock %}
      </div>
    </div>
  </div>
</body>
</html>
```

Depois de salvar e recarregar a página `http://127.0.0.1:8000` você verá, obviamente, um erro familiar `NoReverseMatch` certo?

URL

Vamos abrir o arquivo `blog/urls.py` e escrever:

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

O código final deve se parecer com isso:

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

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

Após recarregar o site, nós veremos um `AttributeError`, desde que nós não temos a visão `post_new` implementada. Vamos adicioná-la agora.

post_new view

Hora de abrir o arquivo `blog/views.py` e adicionar as linhas seguintes com o resto das linhas `from`:

```
from .forms import PostForm
```

e nossa `view`:

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

Para criar um novo formulário `Post`, nós devemos chamar `PostForm()` e passá-lo para o template. Nós iremos voltar para esta *view*, mas por agora vamos criar rapidamente um template para o formulário.

Template(modelos)

Precisamos criar um arquivo `post_edit.html` na pasta `blog/templates/blog`. Pra fazer o formulário funcionar precisamos de muitas coisas:

- Temos que exibir o formulário. Podemos fazer isso simplesmente com um ``.
- A linha acima precisa estar dentro de uma tag HTML form: `<form method="POST">...</form>`
- Precisamos de um botão `salvar`. Fazemos isso com um botão HTML: `<button type="submit">Save</button>`
- E finalmente, depois de abrir a tag `<form ...>` precisamos adicionar um `{% csrf_token %}`. Isso é muito importante, pois é isso que faz o nosso formulário ficar seguro! O Django vai reclamar se você esquecer de adicionar isso e simplesmente salvar o formulário:

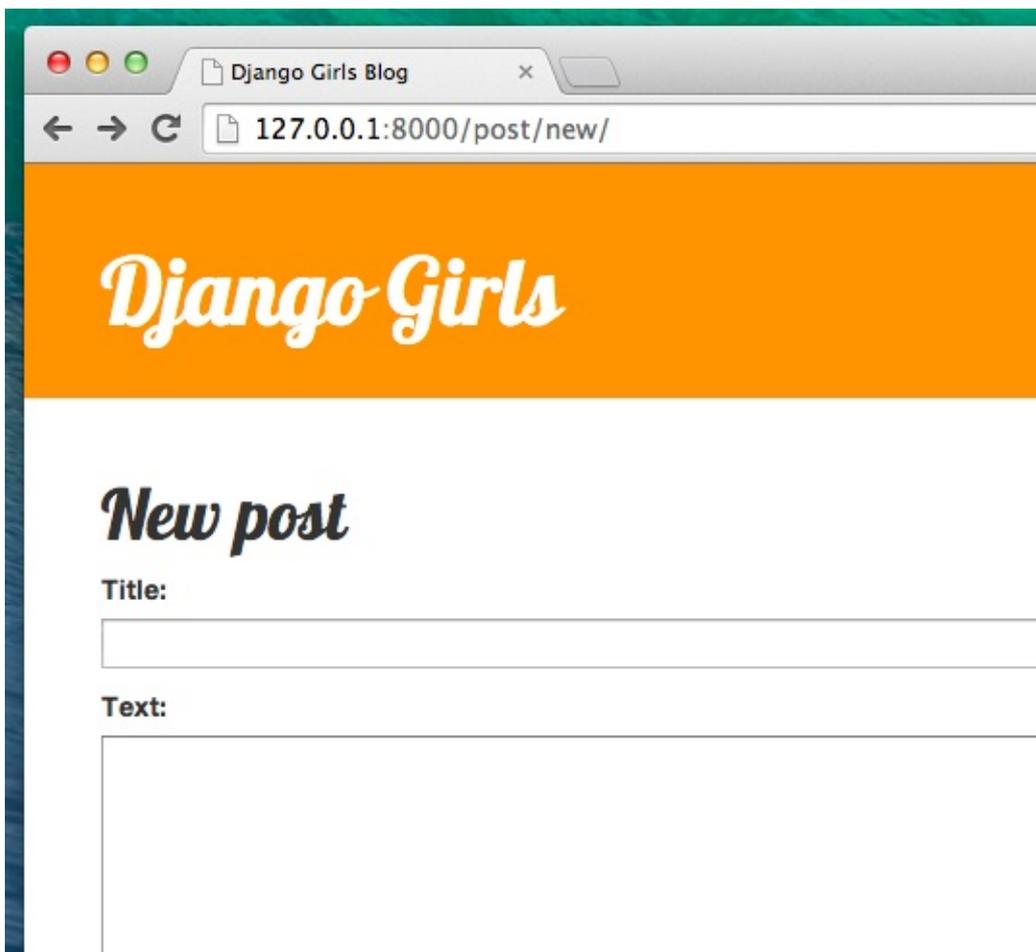


Beleza, então vamos ver como ficou o HTML `post_edit.html`:

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

{% block content %}
    <h1>New post</h1>
    <form method="POST" class="post-form">{% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Guardar</button>
    </form>
{% endblock %}
```

Hora de atualizar! Há! Seu formulário apareceu!



Mas, espere um minuto! Quando você digita alguma coisa nos campos `title` e `text` e tenta salvar o que acontece?

Nada! Estamos novamente na mesma página e nosso texto sumiu... E nenhum post foi adicionado. Então o que deu errado?

A resposta é: nada. Precisamos trabalhar um pouco mais na nossa `view`.

Salvando o formulário

Abra `blog/views.py` mais uma vez. Atualmente tudo que temos na visão `post_new` é:

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

Quando nós enviamos o formulário, somos trazidos de volta para a mesma visão, mas desta vez temos mais alguns dados no `request`, mais especificamente em `request.POST` (o nome não tem nada com "post" de blog, tem a ver com o fato de que estamos "postando" dados). Você se lembra que no arquivo HTML nossa definição de `<form>` tem a variável `method="POST"`? Todos os campos vindos do "form" estarão disponíveis agora em `request.POST`. Você não deveria renomear `POST` para nada diferente disso (o único outro valor válido para `method` é `GET`, mas nós não temos tempo para explicar qual é a diferença).

Então na nossa `view` nós temos duas situações separadas para lidar. A primeira é quanto acessamos a página pela primeira vez e queremos um formulário em branco. E a segunda, é quando nós temos que voltar para a `view` com todos os dados do formulário que nós digitamos. Desse modo, precisamos adicionar uma condição (usaremos `if` para isso).

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

Está na hora de preencher os pontos `[...]`. Se `method` é `POST` então nós queremos construir o `PostForm` com os dados que veem do formulário, certo? Nós iremos fazer assim:

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

Fácil! Próxima coisa é verificar se o formulário está correto(todos os campos requeridos são definidos e valores incorretos não serão salvos). Fazemos isso com `form.is_valid()`.

Verificamos se o formulário é válido e se estiver tudo certo, podemos salvá-lo!

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

Basicamente, temos duas coisas aqui: Salvamos o formulário com `form.save` e adicionamos um autor(desde que não haja o campo `author` em `PostForm`, e este campo é obrigatório!). `commit=False` significa que não queremos salvar o modelo `Post` ainda - queremos adicionar autor primeiro. Na maioria das vezes você irá usar `form.save()`, sem `commit=False`, mas neste caso, precisamos fazer isso. `post.save()` irá preservar as alterações(adicionando autor) e é criado um novo `post` no blog!

Finalmente, não seria fantástico se nós pudéssemos imediatamente ir à página de `post_detail` para o recém-criado blog post, certo? Para fazer isso nós precisaremos de mais uma importação:

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

Adicione-o logo no início do seu arquivo. E agora podemos dizer: vá para a página `post_detail` para um recém-criado post.

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

`blog.views.post_detail` é o nome da view que queremos ir. Lembre-se que essa view exige uma variável `pk`? Para passar isso nas views usamos `pk=post.pk`, onde `post` é o recém-criado blog post.

Ok, nós falamos muito, mas provavelmente queremos ver o que toda a view irá parecer agora, certo?

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

Vamos ver se funciona. Vá para o página <http://127.0.0.1:8000/post/new/>, adicione um `title` e o `text`, salve... e voilà! O novo blog post é adicionado e nós somos redirecionados para a página de `post_detail` !

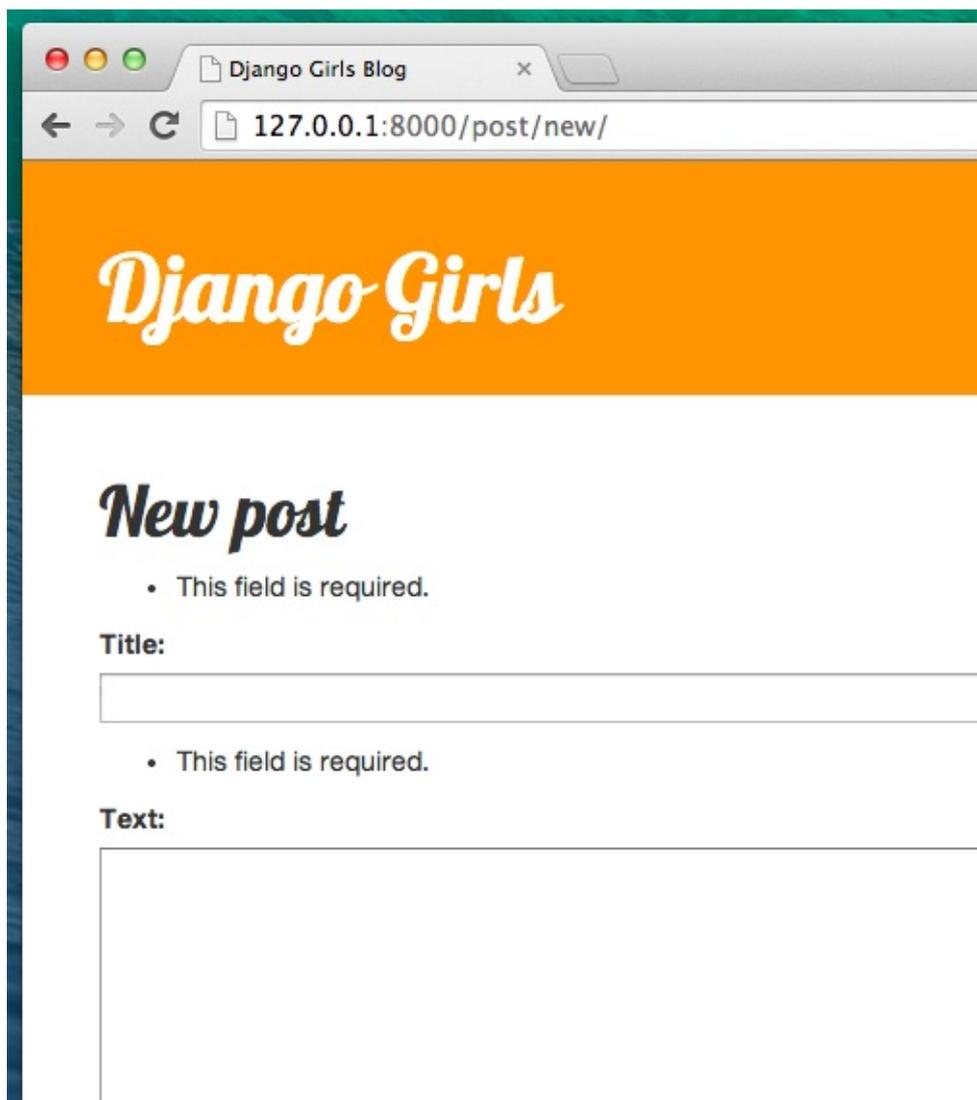
Você provavelmente notou que nós não estamos definindo a data de publicação em tudo. Vamos introduzir um *botão de publicação* em **Django Girls Tutorial: Extensões**.

Isso é incrível!

Validação de formulários

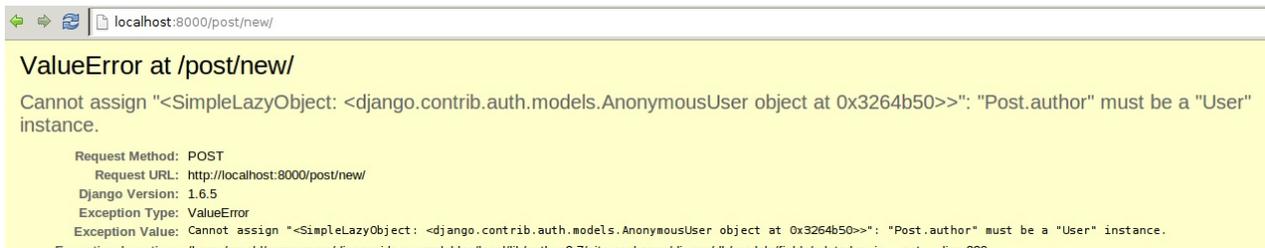
Agora, nós lhe mostraremos como os fórmularios são legais. O post do blog precisa ter os campos `title` e `text`. Em nosso modelo `Post` não dissemos (em oposição a `published_date`) que esses campos não são necessários, então Django, por padrão, espera-os a ser definido.

Tente salvar o formulário sem `title` e `text`. Adivinhe o que vai acontecer!



Django está tomando conta de validar se todos os campos de nosso formulário estão corretos. Não é incrível?

Como recentemente usamos a interface de administração do Django o sistema entende que estamos logados. Existem algumas situações que poderiam levar a sermos deslogados do sistema (fechar o navegador, reiniciar banco de dados etc.). Se você perceber que erros estão aparecendo ao criar um post que referencia um usuário que não está logado, vá para a página admin <http://127.0.0.1:8000> e logue novamente. Isso vai resolver o problema temporariamente. Há um ajuste permanente esperando por você em **lição de casa: adicionar segurança no seu site!**, capítulo após o tutorial principal.



Editando o formulário

Agora sabemos como adicionar um novo formulário. Mas e se quisermos editar um já existente? É muito semelhante ao que fizemos. Vamos criar algumas coisas importantes rapidamente (se você não entender alguma coisa - você deve perguntar a seu professor ou veja os capítulos anteriores, já cobrimos todas essas etapas anteriormente).

Abra `blog/templates/blog/post_detail.html` e adicione a linha:

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

Agora o modelo estará parecido com:

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

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

Em `blog/urls.py` adicionamos esta linha:

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

Nós reutilizaremos o modelo `blog/templates/blog/post_edit.html`, então a última coisa que falta é uma *view*.

Vamos abrir `blog/views.py` e adicionar no final do arquivo:

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

Isso é quase exatamente igual a nossa view de `post_new`, certo? Mas não totalmente. Primeira coisa: passamos um parâmetro extra da url `pk`. Em seguida: pegamos o modelo `Post` que queremos editar com `get_object_or_404` (`Post, pk=pk`) e então, quando criamos um formulário passamos este post como uma `instância`, tanto quando salvamos o formulário:

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

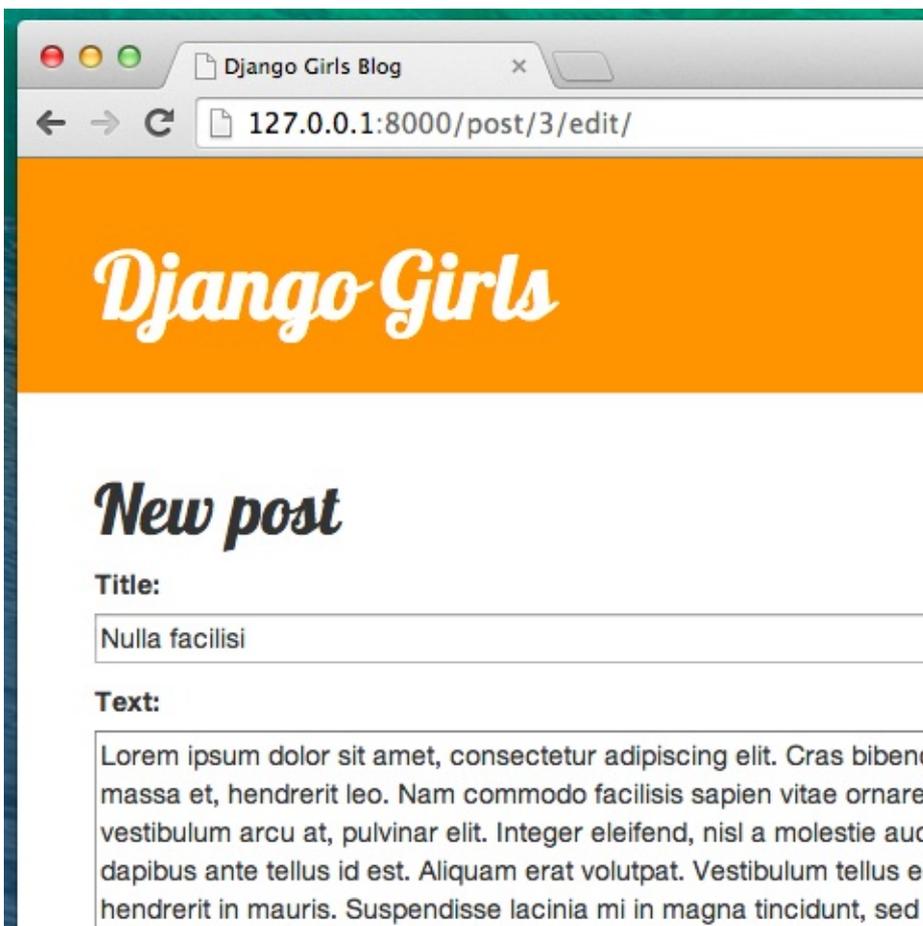
como quando nós apenas abrimos um formulário com este post para editar:

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

Ok, vamos testar se funciona! Vamos para a página `post_detail`. Deve haver um botão editar no canto superior direito:



Quando você clicar nele você verá o formulário com a nossa postagem:



Sinta-se livre para mudar o título ou o texto e salvar as mudanças!

Parabéns! Sua aplicação está ficando cada vez mais completa!

Se você precisar de mais informações sobre formulários do Django você deve ler a documentação:

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

Mais uma coisa: hora de implantar!

Vamos ver se tudo isso funciona na PythonAnywhere. Tempo para outro deploy!

- Primeiro, commit o seu novo código e coloque no Github

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

- Então, em um [console PythonAnywhere Bash](#):

```
$ cd my-first-blog
$ git pull
```

- Finalmente, pule para a [Web tab](#) e aperte **Reload**.

E deve ser isso! Parabéns :)

Domínio

PythonAnywhere te deu um domínio gratuito, mas talvez você não queira ter ".pythonanywhere.com" no final da URL do seu blog. Talvez você queira seu blog apenas "www.infinite-kitten-pictures.org" ou "www.3d-printed-steam-engine-parts.com" ou "www.antique-buttons.com" ou "www.mutant-unicornz.net", ou seja o que vai ser.

Aqui vamos falar um pouco sobre onde obter um domínio e como ligá-lo a seu aplicativo da web em PythonAnywhere. No entanto, você deve saber que a maioria dos domínios custam dinheiro e PythonAnywhere também cobra uma taxa mensal para usar seu próprio nome de domínio -- não é muito dinheiro, no total, mas isso provavelmente é algo que você só quer fazer se você está realmente comprometido!

Onde registrar um domínio?

Um domínio normal custa mais ou menos 15 dólares por ano. Existem domínios mais caros e mais baratos dependendo do provedor. Existem uma série de empresas das quais você pode comprar um domínio: uma simples [pesquisa no google](#) pode listar uma série delas.

O nosso favorito é o [I want my name](#) (eu quero meu nome). Eles anunciam seu serviço como "gestão de domínio indolor", e ele, realmente, é indolor.

Você também pode obter domínios gratuitamente. [dot.tk](#) é um lugar para pegar um, mas você deve estar ciente de que domínios grátis às vezes parecem muito baratos -- se seu site vai ser para um profissional de negócios, você deve pensar em pagar por um domínio "correto" que termina em `.com`.

Como apontar seu domínio no PythonAnywhere

Se você passou por [iwantmyname.com](#), clique em `domínios` no menu e escolha seu domínio recém-adquirido. Em seguida, localize e clique no link `manage DNS records`:

Nameservers	ns1.iwantmyname.net ns2.iwantmyname.net ns3.iwantmyname.net ns4.iwantmyname.net	update nameservers manage DNS records
--------------------	--	--

Agora você precisa localizar este formulário:

Hostname ?	Type ?	Value ?	TTL ?
<input type="text" value="e.g. www, blog or leave empty"/>	<input type="text" value="A"/>	<input type="text" value="e.g. 72.32.231.8, web.me.com"/>	<input type="text" value="3600"/> <input type="button" value="add"/>

E preencher com os seguintes detalhes: -Hostname: `www` - tipo: `CNAME` - valor: seu domínio de PythonAnywhere (por exemplo `djangoirls.pythonanywhere.com`) - TTL: `60`

Hostname ?	Type ?	Value ?	TTL ?
<input type="text" value="www"/> ✓	<input type="text" value="CNAME"/>	<input type="text" value="djangoirls.herokuapp.com"/> ✓	<input type="text" value="3600"/> ✓ <input type="button" value="add"/>

Clique no botão Adicionar e salve as mudanças na parte de baixo.

Nota Se você usou um provedor de domínio diferente, o UI exata para encontrar o seu DNS / configurações de CNAME será diferente, mas seu objetivo é o mesmo: para configurar um CNAME que aponta seu novo domínio no `yourusername.pythonanywhere.com`.

Pode levar alguns minutos para o seu domínio começar a trabalhar, então seja paciente!

Configure o domínio através de um web app na PythonAnywhere.

Você também precisa dizer PythonAnywhere que você deseja usar o seu domínio personalizado.

Vá para a [página PythonAnywhere contas](#) e upgrade sua conta. A opção mais barata (um plano de "Hacker") é bom para começar, você pode sempre atualizá-lo mais tarde quando você ficar super famoso e tiver milhões de acessos.

Em seguida, vá na [Web tab](#) e anote algumas coisas:

- Copie o **caminho para seu virtualenv** e coloque em um lugar seguro
- Clique para seu **arquivo de configuração do wsgi**, copie o conteúdo e cole em um lugar seguro.

Em seguida, **exclua** seu antigo web app. Não se preocupe, isso não vai excluir nada do seu código, ele apenas irá se desligar do domínio *yourusername.pythonanywhere.com*. Em seguida, crie um novo aplicativo web e siga estes passos:

- Digite seu nome de domínio novo
- Escolha "manual configuration"
- Escolha Python 3.4
- E é isso!

Quando você tiver voltado para a web tab.

- Colar o caminho virtualenv que você salvou antes
- Clicar no arquivo de configuração wsgi e colar o conteúdo do seu arquivo de configuração antigo

Clique em reload web app e você deve encontrar seu site live no novo domínio!

Se você tiver qualquer problema, clique no link "Enviar feedback" no site PythonAnywhere, e um dos seus administradores amigáveis vai estar lá para ajudá-lo.

O que vem depois?

Parabéns! Você é **demais**. Estamos orgulhosos! <3

O que fazer agora?

Faça uma pausa e relaxe. Você acabou de fazer algo realmente grande.

Depois disso:

- Siga Django girls no [Facebook](#) ou [Twitter](#) para ficar atualizada

Você pode recomendar outras fontes?

Sim! Primeiro, vá em frente e tente nosso outro livro, chamado [Django Girls Tutorial: Extensions](#).

Depois você pode tentar as fontes listadas abaixo. Todas elas são recomendadas!

- [Django's official tutorial](#)
- [New Coder tutorials](#)
- [Code Academy Python course](#)
- [Code Academy HTML & CSS course](#)
- [Django Carrots tutorial](#)
- [Learn Python The Hard Way book](#)
- [Getting Started With Django video lessons](#)
- [Two Scoops of Django: Best Practices for Django book](#)