

Tutorial

djangogirls

Tabela de conteúdos

Introdução	1.1
Instalação	1.2
Configuração do Chromebook	1.3
Como a Internet funciona	1.4
Introdução à linha de comando	1.5
Instalação do Python	1.6
Editor de Código	1.7
Introdução ao Python	1.8
O que é Django?	1.9
Instalação do Django	1.10
Seu primeiro projeto Django!	1.11
Modelos do Django	1.12
Django Admin	1.13
Deploy!	1.14
URLs	1.15
Django views - hora de criar!	1.16
Introdução ao HTML	1.17
QuerySets e ORM do Django	1.18
Dados dinâmicos em templates	1.19
Templates do Django	1.20
CSS - Deixe tudo mais bonito!	1.21
Estendendo os templates	1.22
Amplie sua aplicação	1.23
Formulários do Django	1.24
O que vem agora?	1.25

Tutorial Django Girls

[chat on gitter](#)

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/>

Bem-vinda

Bem-vinda ao Tutorial Django Girls! Estamos felizes em vê-la aqui :) Neste tutorial, vamos levá-la em uma viagem por baixo do capô das tecnologias web, oferecendo-lhe um vislumbre de todos os bits e peças que precisam se unir para fazer a web funcionar tal como a conhecemos.

Como todas as coisas desconhecidas, isto vai ser uma aventura - mas não se preocupe! Já que você teve coragem para estar aqui, você vai ficar bem :)

Introdução

Você já sentiu como se o mundo fosse cada vez mais tecnológico e que de alguma forma você ficou 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 mesmo para você tentar fazer alguma coisa sozinha?

Bem, nós temos boas notícias! 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 uma programadora. Se você quer ser boa nisso, precisa de meses ou até mesmo anos de treino e prática. Mas nós queremos te mostrar que a programação e a criação de websites não são tão complicadas quanto parecem. Nós tentaremos explicar as diferentes etapas tão bem quanto pudermos, de forma que você não se sinta intimidada pela tecnologia.

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

O que você vai aprender durante o tutorial?

Quando terminar o tutorial, você terá uma pequena aplicação web funcional: seu próprio blog. Nós vamos mostrar como colocá-lo online para que outras pessoas vejam seu trabalho!

Ele se parecerá (mais ou menos) com isso:

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

- Nulla facilisi** (published: 28-06-2014)
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras bibendum sapien interdum, posuere massa et, hendrerit leo. Nam commodo facilisis sapien vitae ornare. Integer eget purus posuere, vesti...
- Fusce vehicula feugiat augue eget consectetur** (published: 28-06-2014)
Pellentesque venenatis elit tortor, eu dictum magna accumsan in. Aenean vestibulum velit arcu, eleifend mattis purus suscipit a. Ut vitae pellentesque lorem. Integer lobortis orci in est molestie t...
- Duis quis imperdiet justo** (published: 28-06-2014)
Nulla ut metus luctus, tristique massa sit amet, venenatis eros. Aliquam hendrerit ligula nec viverra euismod. Vivamus eu sagittis diam, eget pharetra libero. Vestibulum ante ipsum primis in faucib...

Se você seguir o tutorial por conta própria e não tiver uma monitora para ajudar, podemos de ajudar a resolver qualquer problema por esse chat: [chat on gitter](#). Nós pedimos às nossas monitoras e participantes anteriores para acessarem o chat de tempos em tempos para ajudar outras pessoas com o tutorial! Não tenha medo de fazer sua pergunta!

OK, [vamos começar pelo começo...](#)

Seguindo o tutorial em casa

É incrível participar de um workshop de Django Girls, mas estamos conscientes de que isso nem sempre é possível. É por isso que encorajamos você a tentar seguir este tutorial em casa. Para leitores em casa, estamos atualmente preparando vídeos que facilitarão seguir o tutorial por conta própria. Ainda é um trabalho em andamento, mas mais e mais coisas serão abordadas em breve no canal do YouTube [Coding is for girls](#).

Em cada capítulo que já abordamos, há um link que aponta para o vídeo correspondente.

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](#).

Você gostaria de nos ajudar a traduzir este tutorial para outros idiomas?

No momento, 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 [abra uma requisição](#) informando-nos do idioma para que possamos adicioná-lo.

Se você estiver fazendo o tutorial em casa

Se você estiver fazendo o tutorial em casa e não em um dos [eventos Django Girls](#), você pode pular este capítulo e ir direto para o capítulo [Como a internet funciona](#).

Isto porque cobrimos estas coisas no tutorial de qualquer maneira, e esta é apenas uma página adicional que reúne todas as instruções de instalação em um só lugar. O evento Django Girls inclui uma "noite de instalação" quando instalamos tudo que vamos precisar durante o workshop, então essa seção é útil para nós.

Se você achar útil, pode continuar lendo este capítulo também. Mas se quiser começar a aprender antes de instalar um monte de coisas no seu computador, pule este capítulo e explicaremos a parte de instalação para você mais tarde.

Boa sorte!

Instalação

No workshop, você construirá um blog. No tutorial, aparecem certas tarefas de configuração que você deve fazer antes para estar pronta para escrever códigos no dia do evento.

Chromebook setup (if you're using one) Você pode [pular este capítulo]

(<http://tutorial.djangogirls.org/en/installation/#install-python>) se não estiver usando um Chromebook. Caso esteja, sua experiência de instalação será um pouco diferente. Você pode ignorar o restante das instruções de instalação. **Cloud 9** Cloud 9 é uma ferramenta que te dá um editor de código e acesso a um computador conectado à Internet onde você pode instalar, escrever e executar software. Durante o tutorial, o Cloud 9 atuará como sua *máquina local*. Você ainda estará executando comandos em uma interface de terminal, como seus colegas de classe no OS X, Ubuntu ou Windows, mas seu terminal estará conectado a um computador que está em algum outro lugar que o Cloud 9 configurou para você. 1.

Instale o Cloud 9 através da [Chrome Web Store]

(<https://chrome.google.com/webstore/detail/cloud9/nbdmccoknlfggadpfkmcpcnamfnbkmkcp>) 2. Acesse o site [c9.io]

(<https://c9.io>) 3. Cadastre-se para criar uma conta 4. Clique em *Create a New Workspace* 5. Dê o nome de *django-girls* 6. Selecione *Blank* (segundo a partir da direita na linha inferior com logotipo laranja) Agora a sua tela deve exibir uma interface com uma barra lateral, uma grande janela principal com algum texto e uma pequena janela na parte inferior, semelhante a isto:

Cloud 9

seunomededeusuário:~/workspace \$ Esta área inferior é seu *terminal*, onde você vai dar as instruções para o computador que o Cloud 9 preparou para você. Você pode redimensionar a janela para torná-la um pouco maior. **Ambiente Virtual** Um ambiente virtual (também chamado de virtualenv) é como se fosse uma caixa privada onde nós podemos colocar código de computador útil para um projeto que estejamos trabalhando. Nós os utilizamos para manter os vários pedaços de código que queremos para nossos vários projetos separados, para que as coisas não se misturem entre os projetos. No seu terminal, na parte de baixo da interface do Cloud 9, execute o seguinte:

Cloud 9

sudo apt update sudo apt install python3.6-venv Se isso ainda não funcionar, peça ajuda ao seu treinador. Em seguida, execute:

Cloud 9

mkdir djangogirls cd djangogirls python3.6 -mvenv myvenv source myvenv/bin/activate pip install django~=2.0.6 (note que na última linha nós utilizamos um til seguido de um sinal de igual: ~=). **GitHub** Crie uma conta no [GitHub] (<https://github.com>). **PythonAnywhere** O tutorial do Django Girls inclui uma seção que chamamos de Deployment (algo como "implantação", em Português), que é o processo de pegar o código que alimenta a sua nova aplicação web e movê-lo para um computador de acesso público (chamado de servidor), então outras pessoas serão capazes de ver seu trabalho. Esta parte é um pouco esquisita quando o tutorial é feito num Chromebook, dado que já estamos usando um computador que está na Internet (ao contrário de, digamos, um laptop). Porém, ainda útil, já que podemos pensar no

ambiente do Cloud 9 como um lugar pra nosso trabalho "em andamento" e Python Anywhere como um lugar para mostrar nosso trabalho conforme ele vai ficando mais completo. Assim, crie uma nova conta Python Anywhere em [\[www.pythonanywhere.com\]\(https://www.pythonanywhere.com\)](https://www.pythonanywhere.com).

Instale o Python

Para leitoras em casa: esse capítulo é abordado no vídeo [Instalando Python & Editor de Código](#).

Esta seção baseia-se em tutoriais da Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>)

Django é escrito em Python. Precisamos dele para fazer qualquer coisa no Django. Por isso, vamos começar instalando o Python! Nós precisamos que você instale o Python 3.6. Se você tiver alguma versão mais antiga, é preciso atualizá-la.

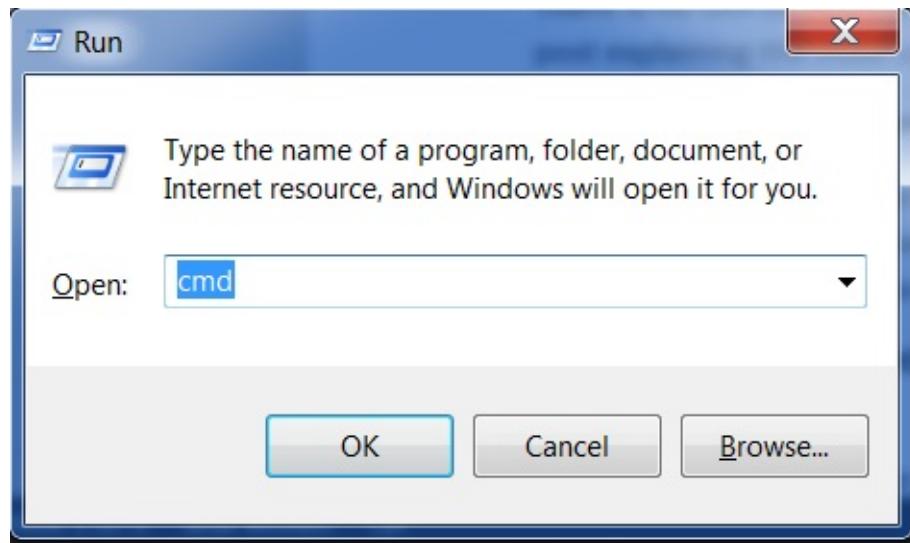
Install Python: Windows

Primeiro, verifique se o computador está executando a versão 32-bit ou a versão 64-bit do Windows. Faça isso pressionando a tecla windows no seu teclado (aquela com a janela do windows) + a tecla Pause/Break. Feito isso, uma tela abrirá com as informações do seu windows. Nessa tela, verifique a seção "Tipo de sistema" e confira que versão está sendo executada. Você pode baixar o Python para Windows no website <https://www.python.org/downloads/windows/>. Clique no link: "Versão Mais Recente Python 3 - Python x.x.x". Se seu computador está executando a versão **64-bit** do windows, baixe o **instalador executável do Windows x86-64**. Caso contrário, baixe o **instalador executável x86 do Windows**. Depois de baixar o instalador, você precisa executá-lo (dando um duplo-clique nele) e seguir as instruções.

Existe uma coisa com a qual você precisa ter cuidado: durante a instalação, você verá uma janela marcada como "Setup". Certifique-se de selecionar a caixa "Adicionar Python 3.6 ao CAMINHO" e clique em "Instalar agora", conforme mostrado aqui:



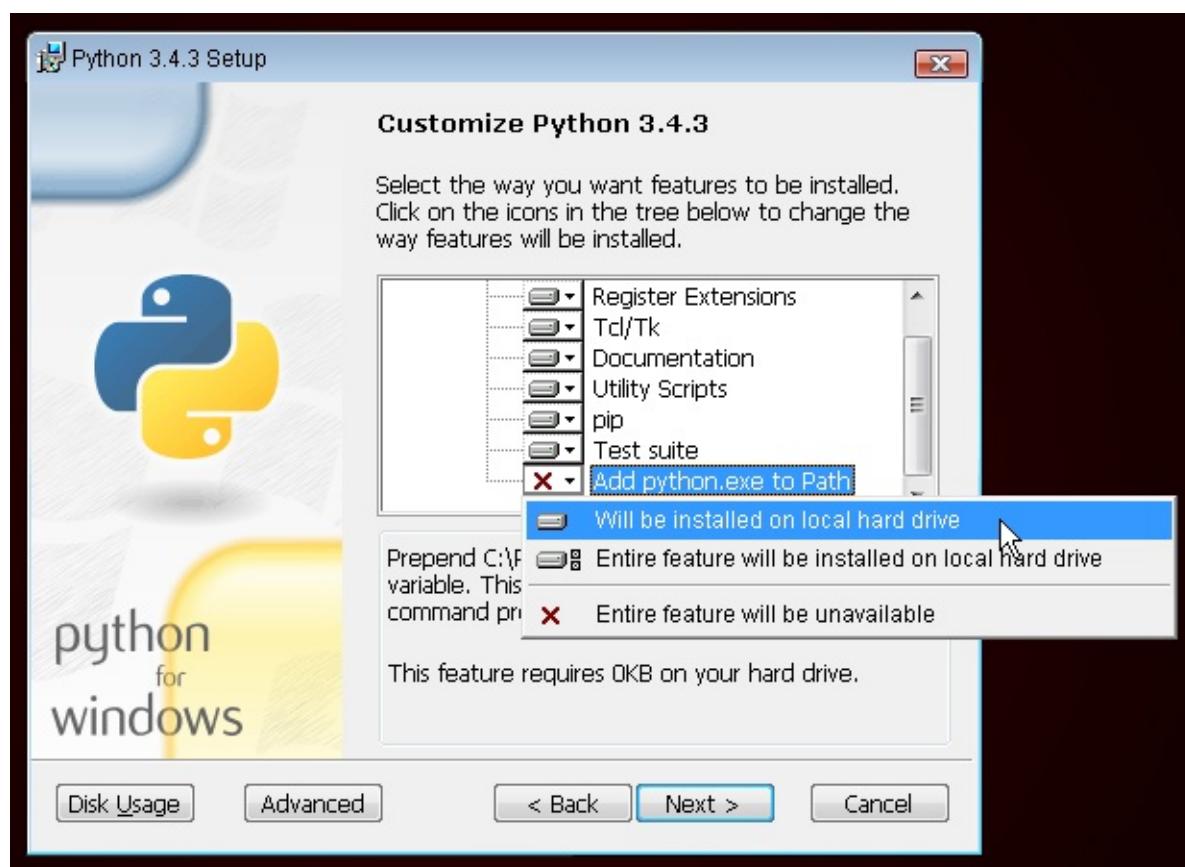
Nas próximas etapas, você usará a linha de comando do Windows (vamos te explicar tudo sobre isso também). Por enquanto, se você precisa digitar alguns comandos, vá ao menu iniciar e digite "Command Prompt" no campo de busca. (Em versões mais antigas do Windows, é possível iniciar a linha de comando com Start menu → Sistema do Windows → Prompt de comando.) Você também pode segurar a tecla windows + "R" até aparecer a janela "Executar". Para abrir a Linha de Comando, digite "cmd" e pressione enter na janela "Executar".



<0>Observação: se você estiver usando uma versão antiga do Windows (7, Vista, ou qualquer outra mais antiga) e o instalador do 3.6. x Python falhar com um erro, você pode tentar:

1. instalar todas as atualizações do Windows e tentar instalar o Python 3.6 novamente; ou
2. instalar uma [versão mais antiga do Python](#), por exemplo, [3.4.6](#).

Se você instalar uma versão mais antiga do Python, a tela de instalação pode parecer um pouco diferente da mostrada acima. Certifique-se de rolar até ver a opção "Add python.exe to Path", então clique no botão à esquerda e escolha "Will be installed on local hard drive":



Install Python: OS X

Observação: Antes de instalar o Python no Mac OS X, você deve garantir que suas configurações permitam a instalação de pacotes que não estejam na App Store. Vá para preferências do sistema (dentro da pasta Aplicativos), clique em "Segurança & Privacidade" e depois na guia "Geral". Se a configuração "Permitir que apps baixados:" estiver definida como "Mac App Store," mude para "Mac App Store e desenvolvedores identificados."

Você precisa visitar <https://www.python.org/downloads/release/python-361/> e baixar o instalador do Python:

- Faça o download do arquivo *Mac OS X 64-bit/32-bit installer*,
- Dê um duplo clique no arquivo *python-3.6.1-macosx10.6.pkg* para executar o instalador.

Install Python: 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 o terminal e digite o seguinte comando:

command-line

```
$ python3 --version  
Python 3.6.1
```

Se você tem instalada uma outra "versão micro" do Python, por exemplo, 3.6.0, você não precisa atualizá-la. Se você não tiver o Python instalado ou quiser uma versão diferente, faça assim:

Install Python: Debian or Ubuntu

Digite o seguinte comando no terminal:

command-line

```
$ sudo apt install python3.6
```

Install Python: Fedora

Use o seguinte comando no terminal:

command-line

```
$ sudo dnf install python3
```

Se você estiver em versões mais antigas do Fedora, pode receber um erro dizendo que o comando `dnf` não foi encontrado. Nesse caso, você precisa usar o `yum` em vez disso.

Install Python: openSUSE

Use o seguinte comando no terminal:

command-line

```
$ sudo zypper install python3
```

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

command-line

```
$ python3 --version  
Python 3.6.1
```

Observação: Se você estiver no Windows e receber uma mensagem de erro dizendo que o `python3` não foi encontrado, tente utilizar `python` (sem o `3`) e verifique se ela corresponde à versão Python 3.6.

Se você tem alguma dúvida ou se alguma coisa deu errado e você não tem a menor ideia do que fazer, pergunte à sua monitora! Nem sempre tudo sai conforme o esperado e é melhor pedir ajuda a alguém mais experiente.

Configure o virtualenv e instale o Django

Esta seção baseia-se em tutoriais da Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Parte deste capítulo é baseada em [tutorial django-marcador](#) licenciado sob 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 instalar o Django, vamos instalar uma ferramenta muito útil para ajudar a manter o ambiente de trabalho no nosso computador organizado. Você pode pular esse passo, mas ele é altamente recomendado. Começar com a melhor instalação possível poupará você de muito trabalho no futuro!

Vamos criar um **ambiente virtual** (também chamado um *virtualenv*). O *virtualenv* isolará seu código Python/Django em um ambiente organizado por projetos. Isso significa que as alterações que você fizer em um website não afetarão os outros projetos que você estiver desenvolvendo ao mesmo tempo. Legal, né?

Tudo o que você precisa fazer é encontrar o diretório em que você quer criar o `virtualenv`; seu diretório Home, por exemplo. No Windows, pode aparecer como `C:\Users\Name` (onde `Name` é seu usuário de login).

Observação: No Windows, certifique-se de que esse diretório não contém palavras acentuadas ou caracteres especiais; se o seu usuário contém caracteres acentuados, use um diretório diferente, por exemplo: `C:\djangogirls`.

Para este tutorial, usaremos um novo diretório `djangogirls` no seu diretório home:

command-line

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

Vamos fazer um *virtualenv* chamado `meuenv`. O formato geral desse comando é:

command-line

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

Virtual environment: Windows

Para criar um novo `virtualenv`, você deve abrir o terminal e executar `python -m venv myvenv`. Deve ficar assim:

command-line

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

Onde `myvenv` é o nome do seu `virtualenv`. Você pode usar qualquer outro nome, mas sempre use minúsculas e não use espaços, acentos ou caracteres especiais. Também é uma boa ideia manter o nome curto - você irá referenciá-lo muitas vezes!

Virtual environment: Linux and OS X

Podemos criar um `virtualenv` no Linux ou no OS X executando `python3 -m venv myvenv`. Deve ficar assim:

command-line

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

`myvenv` é o nome do seu `virtualenv`. Você pode usar qualquer outro nome, mas use sempre letras minúsculas e não use espaços entre as palavras. Também é uma boa ideia manter o nome curto pois você vai escrevê-lo muitas vezes!

Observação: Em algumas versões do Debian/Ubuntu, você pode receber o seguinte erro:

command-line

```
The virtual environment was not created successfully because ensurepip is not available. On Debian/Ubuntu systems, you need to install the python3-venv package using the following command.  
    apt install python3-venv  
You may need to use sudo with that command. After installing the python3-venv package, recreate your virtual environment.
```

Caso você receba esse erro, siga as instruções acima e instale o pacote `python3-venv`:

command-line

```
$ sudo apt install python3-venv
```

Observação: Em algumas versões do Debian/Ubuntu, iniciar o ambiente virtual com este comando gera o seguinte erro:

command-line

```
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`.

command-line

```
$ sudo apt install python-virtualenv  
$ virtualenv --python=python3.6 myvenv
```

Observação: Se você obtiver um erro como

command-line

```
E: Unable to locate package python3-venv
```

no lugar do comando mostrado acima, execute esse:

command-line

```
sudo apt install python3.6-venv
```

Trabalhando com o `virtualenv`

O comando acima criará um diretório chamado `myvenv` (ou qualquer que seja o nome que você escolheu) que contém o nosso ambiente virtual (basicamente um conjunto de diretórios e arquivos).

Working with `virtualenv`: Windows

Inicie o seu ambiente virtual executando:

command-line

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

Observação: no Windows 10, você pode obter um erro no Windows PowerShell que diz `execution of scripts is disabled on this system`. Neste caso, abra uma outra janela do Windows PowerShell com a opção de "Executar como Administrador". Assim, execute o comando abaixo antes de iniciar o seu ambiente virtual:

command-line

```
C:\WINDOWS\system32> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the about_Execution_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy? [Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): A
```

Working with virtualenv: Linux and OS X

Inicie o seu ambiente virtual executando:

command-line

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

Lembre-se de substituir `myvenv` pelo nome que você escolheu para o `virtualenv`!

Observação: às vezes `source` pode não estar disponível. Nesses casos, tente fazer isso:

command-line

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

Você vai saber que tem um `virtualenv` funcionando quando vir que a linha de comando no seu console tem o prefixo `(myvenv)`.

Ao trabalhar em de um ambiente virtual, o comando `python` irá automaticamente se referir à versão correta para que você possa digitar `python` em vez de `python3`.

Pronto, já temos todas as dependências importantes no lugar. Finalmente podemos instalar o Django!

Instalando o Django

Agora que você tem seu `virtualenv` ativo, pode instalar o Django.

Antes de fazer isto, devemos garantir que temos instalada a última versão do `pip`, que é o software que usamos para instalar o Django:

command-line

```
(myvenv) ~$ python3 -m pip install --upgrade pip
```

Instalando pacotes com requisitos

O arquivo "requirements.txt" guarda as dependências que serão instaladas utilizando o `pip install`:

Primeiramente, crie um arquivo `requirements.txt` dentro da pasta `djangogirls/`:

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

E adicione o seguinte texto ao arquivo `djangogirls/requirements.txt`:

djangogirls/requirements.txt

```
Django~=2.0.6
```

Agora, execute `pip install -r requirements.txt` para instalar o Django.

command-line

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

Installing Django: Windows

Se você receber um erro ao chamar o pip na plataforma Windows, verifique se o caminho do projeto contém espaços, acentos ou caracteres especiais (exemplo, `c:\Users\User Name\django`). Se sim, considere movê-lo para outro lugar sem espaços, acentos ou caracteres especiais (sugestão: `c:\django`). Crie um novo virtualenv no diretório recém-criado, exclua o mais velho e tente novamente executar o comando acima. (Mover o diretório de virtualenv não vai funcionar pois o virtualenv usa caminhos absolutos.)

Installing Django: Windows 8 and Windows 10

Sua linha de comando pode congelar depois de você tentar instalar o Django. Neste caso, ao invés do comando acima, use:

command-line

```
C:\Users\Name\django> python -m pip install -r requirements.txt
```

Installing Django: Linux

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

É isto! Você agora (finalmente) está pronta para criar uma aplicação Django!

Instale um editor de código

Existem muitos editores de código diferentes e escolher um trata-se essencialmente de preferência pessoal. A maioria dos programadores de Python usa os complexos, mas extremamente poderosos IDEs (Integrated Development Environments, ou, em português, Ambientes de Desenvolvimento Integrado), tais como o PyCharm. Para um iniciante, entretanto, estas IDEs não são muito adequadas; nossas recomendações são igualmente poderosas, mas bem mais simples.

Nossas sugestões estão logo abaixo, mas sinta-se livre para perguntar à sua monitora quais são suas preferências - será mais fácil escolher com a ajuda dela.

Gedit

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

[Baixe-o aqui](#)

Sublime Text 3

O Sublime Text é um editor muito popular. Ele tem um período de avaliação gratuito e está disponível para vários sistemas operacionais.

[Baixe-o aqui](#)

Atom

O Atom é um editor de código recente criado pelo [GitHub](#). Ele é gratuito, tem código aberto e está disponível para Windows, OS X e Linux.

[Baixe-o aqui](#)

Por que estamos instalando um editor de código?

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

A primeira razão é que o código precisa estar em **texto puro, sem formatação**, e o problema com programas como o Word e o Textedit é que eles não produzem texto puro de fato. Eles produzem texto rico (com fontes e formatação), usando formatos personalizados, como [RTF](#) (Rich Text Format, ou Formato de Texto Rico, em português).

A segunda razão é que editores de código são especializados em edição de código, então eles podem fornecer funcionalidades úteis, como destacar o código com cores de acordo com seu significado ou fechar aspas automaticamente para você.

Veremos tudo isso em ação mais pra frente. Logo, logo, seu bom e velho editor de código será uma de suas ferramentas preferidas. :)

Instale o Git

O Git é um "sistema de controle de versão" usado por muitos programadores. Este software pode acompanhar mudanças em arquivos ao longo do tempo para que você possa recuperar versões específicas mais tarde. Um pouco parecido com o recurso "controlar alterações" do Microsoft Word, mas muito mais poderoso.

Instalando o Git

Installing Git: Windows

Você pode baixar o Git em [git-scm.com](#). Clique em "next" em todos os passos, exceto em um: no passo intitulado "Ajustando o PATH do seu ambiente" (em inglês "Adjusting your PATH environment"), escolha "Use Git e ferramentas Unix opcionais do Prompt de Comando do Windows" (em inglês "Use Git and optional Unix tools from the Windows Command Prompt"), que é a opção mais abaixo. Fora isso, as configurações padrão estão ótimas. "Checkout Windows-style, commit Unix-style line endings" está bom.

Não se esqueça de reiniciar o prompt de comando ou o powershell depois que a instalação terminar com sucesso.

Installing Git: OS X

Baixe o Git em [git-scm.com](#) e siga as instruções.

Observação: Se estiver rodando o OS X 10.6, 10.7, ou 10.8, você precisará instalar essa versão do git: [Instalado Git para o OS X Snow Leopard](#)

Installing Git: Debian or Ubuntu

command-line

```
$ sudo apt install git
```

Installing Git: Fedora

command-line

```
$ sudo dnf install git
```

Installing Git: openSUSE

command-line

```
$ sudo zypper install git
```

Crie uma conta no GitHub

Vá para [GitHub.com](#) e cadastre uma conta de usuário.

Crie uma conta no PythonAnywhere

Cadastre uma conta gratuita de "Beginner" (Iniciante) no PythonAnywhere:

- [www.pythonanywhere.com](#)

Observação: Quando escolher seu nome de usuário, lembre-se de que a URL do seu blog vai ter o formato `nomedeusuário.pythonanywhere.com`, então escolha seu próprio apelido ou um nome que represente o tema do seu blog.

Criando um token de API do PythonAnywhere

Você só precisa fazer isso uma vez. Quando estiver inscrita no PythonAnywhere, você será levada ao seu dashboard. Encontre o link no lado direito superior da página "Accounts", e então selecione a aba "API token" e clique no botão que diz "Create new API token".

Comece a ler

Parabéns, você tem tudo configurado e pronto para começar! Se você ainda tem tempo antes do workshop, seria útil começar a ler alguns dos capítulos iniciais:

- [Como a internet funciona](#)
- [Introdução à linha de comando](#)
- [Introdução ao Python](#)
- [O que é Django?](#)

Aproveite o workshop!

Quando você começar a oficina, já será capaz de ir direto para o [seu primeiro projeto Django!](#) porque já viu todo o material dos capítulos anteriores.

Configuração do Chromebook

Nota Se você já realizou os passos da Instalação, não é preciso fazer isso de novo - você pode pular direto para a [Introdução ao Python](#).

Você pode [pular este capítulo](#) se não estiver usando um Chromebook. Caso esteja, sua experiência de instalação será um pouco diferente. Você pode ignorar o restante das instruções de instalação.

Cloud 9

Cloud 9 é uma ferramenta que te dá um editor de código e acesso a um computador conectado à Internet onde você pode instalar, escrever e executar software. Durante o tutorial, o Cloud 9 atuará como sua *máquina local*. Você ainda estará executando comandos em uma interface de terminal, como seus colegas de classe no OS X, Ubuntu ou Windows, mas seu terminal estará conectado a um computador que está em algum outro lugar que o Cloud 9 configurou para você.

1. Instale o Cloud 9 através da [Chrome Web Store](#)
2. Acesse o site [c9.io](#)
3. Cadastre-se para criar uma conta
4. Clique em *Create a New Workspace*
5. Dê o nome de *djangogirls*
6. Selecione *Blank* (segundo a partir da direita na linha inferior com logotipo laranja)

Agora a sua tela deve exibir uma interface com uma barra lateral, uma grande janela principal com algum texto e uma pequena janela na parte inferior, semelhante a isto:

Cloud 9

```
seunomedusuário:~/workspace $
```

Esta área inferior é seu *terminal*, onde você vai dar as instruções para o computador que o Cloud 9 preparou para você. Você pode redimensionar a janela para torná-la um pouco maior.

Ambiente Virtual

Um ambiente virtual (também chamado de *virtualenv*) é como se fosse uma caixa privada onde nós podemos colocar código de computador útil para um projeto que estejamos trabalhando. Nós os utilizamos para manter os vários pedaços de código que queremos para nossos vários projetos separados, para que as coisas não se misturem entre os projetos.

No seu terminal, na parte de baixo da interface do Cloud 9, execute o seguinte:

Cloud 9

```
sudo apt update  
sudo apt install python3.6-venv
```

Se isso ainda não funcionar, peça ajuda ao seu treinador.

Em seguida, execute:

Cloud 9

```
mkdir djangogirls  
cd djangogirls  
python3.6 -m venv myvenv  
source myvenv/bin/activate  
pip install django~=2.0.6
```

(note que na última linha nós utilizamos um til seguido de um sinal de igual: ~=).

GitHub

Crie uma conta no [GitHub](#).

PythonAnywhere

O tutorial do Django Girls inclui uma seção que chamamos de Deployment (algo como "implantação", em Português), que é o processo de pegar o código que alimenta a sua nova aplicação web e movê-lo para um computador de acesso público (chamado de servidor), então outras pessoas serão capazes de ver seu trabalho.

Esta parte é um pouco esquisita quando o tutorial é feito num Chromebook, dado que já estamos usando um computador que está na Internet (ao contrário de, digamos, um laptop). Porém, ainda útil, já que podemos pensar no ambiente do Cloud 9 como um lugar pra nosso trabalho "em andamento" e Python Anywhere como um lugar para mostrar nosso trabalho conforme ele vai ficando mais completo.

Assim, crie uma nova conta Python Anywhere em www.pythonanywhere.com.

Como a Internet funciona

Para leitoras em casa: este capítulo é abordado no vídeo [How the Internet Works](#).

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

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

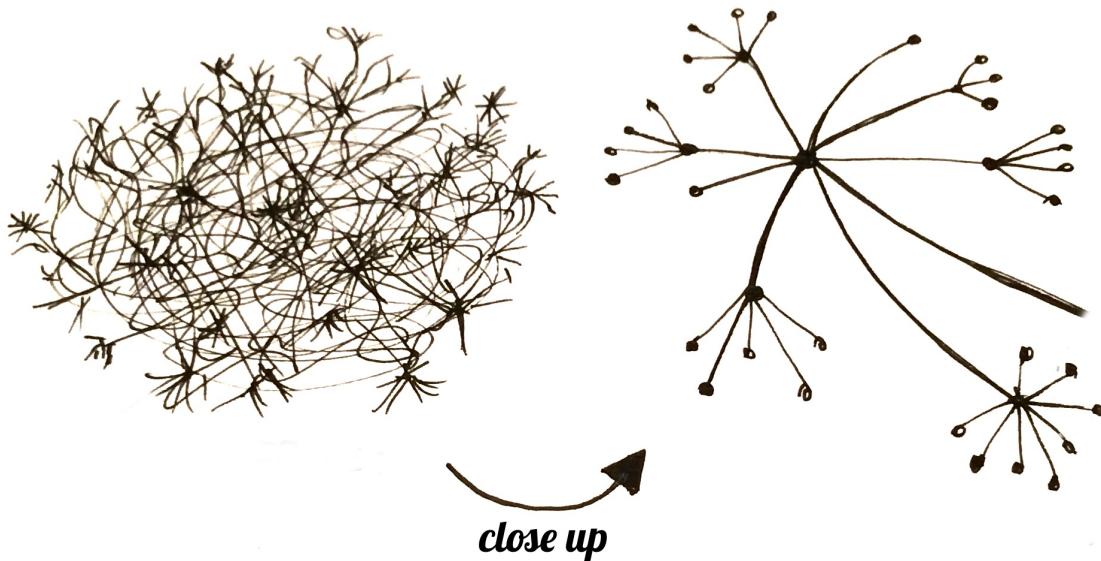
A primeira coisa que você precisa entender é que um site é só um monte de arquivos salvos em um disco rígido. Igual a seus filmes, músicas, ou imagens. No entanto, há uma parte que é exclusiva para sites: ela inclui códigos de computador chamados HTML.

Se você não estiver familiarizada com programação, pode ser difícil compreender o HTML no começo, mas seus navegadores da web (como o Chrome, Safari, Firefox, etc) amam ele. Os navegadores da Web são projetados para entender esse código, seguir suas instruções e apresentar esses arquivos de que seu site é feito, exatamente como você quer.

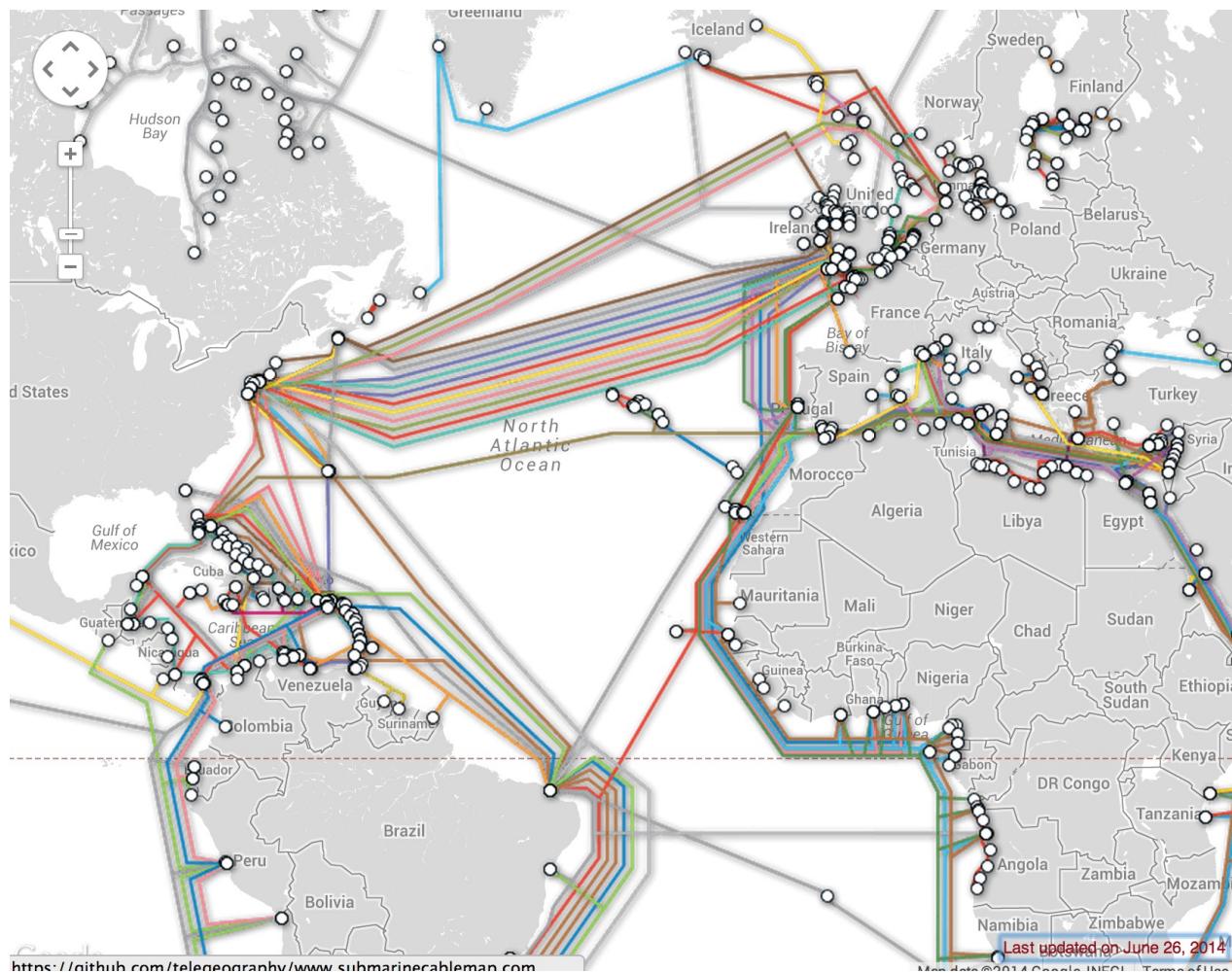
Como qualquer arquivo, os arquivos HTML precisam ser armazenados num disco rígido. Para a internet, usamos poderosos computadores especiais chamados *servidores*. Eles não têm uma tela, um mouse ou um teclado, porque sua finalidade principal é armazenar dados e servi-los. É por isso que eles são chamados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

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

Fizemos um desenho para ajudar! Veja:

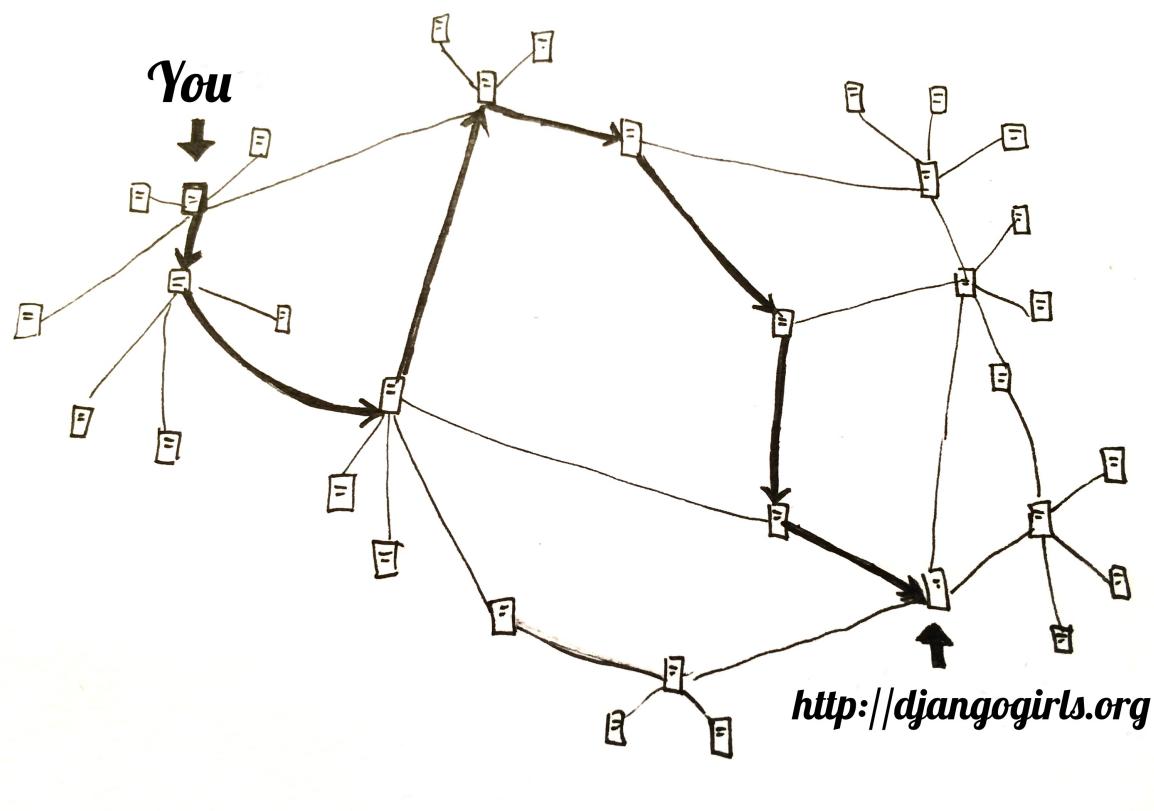


Que bagunça, né? Na verdade, a internet é uma rede de máquinas conectadas (os *servidores* mencionados acima). São centenas de milhares de máquinas! Muitos, muitos quilômetros de cabos em todo o mundo! Para ver o quanto complicada a internet é, você pode visitar um site (<http://submarinecablemap.com/>)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



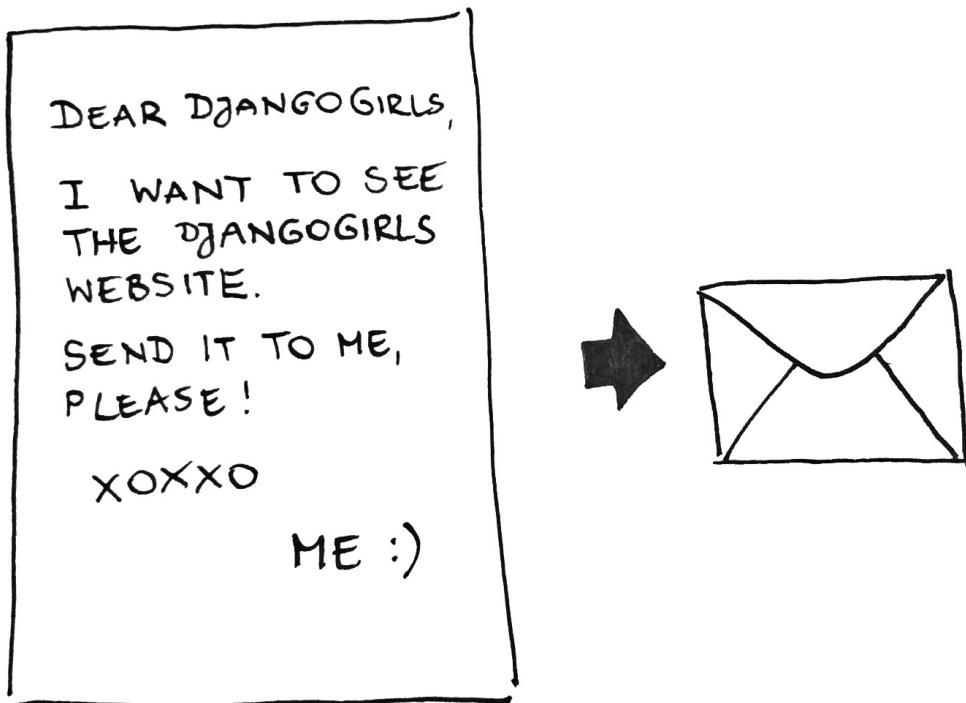
Fascinante, né? Mas, obviamente, não é possível ter um fio ligando todas as máquinas conectadas à internet. Logo, para alcançar uma máquina (por exemplo aquela onde [https://django.org](https://.djangoproject.org) está salva), precisamos passar uma requisição por muitas máquinas diferentes.

É algo assim:



Imagine que quando digita <http://djangogirls.org>, você envia uma carta que diz: "Queridas Django Girls, eu desejo ver o site djangogirls.org. Enviem-no para mim, por favor!"

Sua carta vai para a agência dos correios mais próxima de você. Então, ela vai para outra agência um pouco mais perto do destinatário e, em seguida, para outra e outra até ser entregue. A única coisa diferente é que se você enviar muitas cartas (*pacotes de dados*) para o mesmo lugar, elas podem passar por agências totalmente diferentes (*roteadores*). Isso depende 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, na internet usamos endereços de IP. Primeiro seu computador pergunta pelo DNS (Domain Name System - Sistema de Nome de Domínio) para traduzir djangogirls.org para um endereço de IP. Isso funciona mais ou menos como as antigas listas telefônicas em que você podia procurar o número e endereço da pessoa que queria contactar.

Quando você envia uma carta, ela precisa ter certas características para ser entregue corretamente: um endereço, um selo, etc. E você usa uma linguagem que o destinatário comprehende, certo? O mesmo se aplica aos *pacotes de dados* que você envia para acessar um site. Nós usamos um protocolo chamado HTTP (Hypertext Transfer Protocol).

Então, de forma simplificada, um site precisa ter um *servidor* (máquina) onde ele vive. Quando o *servidor* recebe uma *solicitação* de entrada (numa carta), ele envia em resposta seu website (em outra carta).

Este é um tutorial de Django, então você deve estar imaginando o que o Django faz. Quando envia uma resposta, nem sempre você quer mandar a mesma coisa para todo mundo. É muito melhor que as cartas sejam personalizadas, especialmente para a pessoa que acabou de nos escrever, né? O Django nos ajuda a criar essas cartas personalizadas. :)

Chega de falar, é hora de criar!

Introdução à linha de comando

Para as leitoras em casa: este capítulo é coberto no vídeo [Sua nova amiga: a linha de comando](#).

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

Vamos apresentá-la à sua primeira nova amiga: a linha de comando!

As etapas a seguir mostraram 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.

Observação: Note que ao longo deste tutorial, intercalamos o uso dos termos 'diretório' e 'pasta', mas eles significam a mesma coisa.

O que é a linha de comando?

A janela, que geralmente é chamada de **linha de comando** ou **interface de linha de comando**, é uma aplicação de texto para ver e manipular arquivos em seu computador. É como Windows Explorer ou o Finder no Mac, mas sem a 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.

Opening: Windows

Vá para o Menu iniciar → Sistema do Windows → Prompt de comando.

Em versões antigas do Windows, procure em Menu iniciar → Todos programas → Acessórios → Prompt de comando.

Opening: OS X

Vá para Aplicações → Utilidades → Terminal.

Opening: Linux

Provavelmente você vai achar em Aplicativos → Acessórios → 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.

Prompt: OS X and Linux

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

command-line

```
$
```

Prompt: Windows

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

command-line

```
>
```

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

Uma pequena observação: pode ser que apareça algo como `C:\Users\ola>` ou `Olas-MacBook-Air:~ ola$` antes do cursor e isso está 100% correto.

A parte que vai até e inclui o `$` ou `>` é chamada de *prompt de linha de comando*, ou *prompt*, de forma breve. Ele está pedindo que você digite algo.

No tutorial, quando queremos que você digite um comando, nós incluiremos o `$` ou `>` e algumas vezes algum texto adicional à esquerda. Você pode ignorar o que está à esquerda e apenas digitar o comando que inicia após o prompt.

Seu primeiro comando (Uhuu!)

Vamos começar digitando este comando:

Your first command: OS X and Linux

command-line

```
$ whoami
```

Your first command: Windows

command-line

```
> whoami
```

E então pressione a tecla `enter`. Este é o nosso resultado:

command-line

```
$ whoami  
olasitarska
```

Como você pode ver, o computador acabou de mostrar seu nome de usuário na tela. Legal, né? :)

Tente escrever cada comando, não copie e cole. Assim você vai se lembrar melhor deles!

O Básico

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

Pasta atual

Seria legal saber em que diretório estamos, né? Para isso, digite o seguinte comando e clique `enter`:

Current directory: OS X and Linux

command-line

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

Observação: 'pwd' significa 'print working directory' (imprima/mostre o diretório de trabalho).

Current directory: Windows

command-line

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

Observação: 'cd' significa 'change directory' em inglês, o que se traduz para 'mudar de diretório'. Com o powershell, você pode utilizar pwd da mesma forma como no Linux ou Mac OS X.

Você provavelmente vai ver algo parecido em seu computador. A linha de comando geralmente inicia no diretório principal do usuário, também chamado de diretório "home", em Inglês.

Listando arquivos e pastas

Então, o que tem no seu computador? Seria legal descobrir. Vamos ver:

List files and directories: OS X and Linux

command-line

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

List files and directories: Windows

command-line

```
> 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  
...
```

Observação: No powershell, você também pode usar 'ls' como no Linux e Mac OS X.

Entrando em outra pasta

Agora vamos para a pasta Desktop:

Change current directory: OS X and Linux

command-line

```
$ cd Desktop
```

Change current directory: Windows

command-line

```
> cd Desktop
```

Veja se realmente entramos na pasta:

Create if changed: OS X and Linux

command-line

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

Create if changed: Windows

command-line

```
> 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 em sua área de trabalho para praticar? Use o seguinte comando:

Create directory: OS X and Linux

command-line

```
$ mkdir practice
```

Create directory: Windows

command-line

```
> mkdir practice
```

Esse pequeno comando criará um diretório chamado `practice` em sua área de trabalho. Você pode verificar se o diretório realmente está lá olhando sua área de trabalho ou executando o comando `ls` ou `dir`! 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: crie um diretório chamado `test` dentro do diretório `practice`. (Use os comandos `cd` e `mkdir`.)

Solução:

Exercise solution: OS X and Linux

command-line

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

Exercise solution: Windows

command-line

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

Parabéns! :)

Limpando

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

Primeiro, precisamos voltar para a pasta Desktop:

Clean up: OS X and Linux

command-line

```
$ cd ..
```

Clean up: Windows

command-line

```
> cd ..
```

Ao utilizar o `..` junto com o comando `cd`, você muda do diretório atual para o diretório pai (o diretório que contém o seu diretório atual).

Veja onde você está:

Check location: OS X and Linux

command-line

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

Check location: Windows

command-line

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

Agora é hora de deletar o diretório `practice`:

Atenção: A exclusão de arquivos usando `del`, `rmdir` ou `rm` é irreversível; ou seja, os arquivos excluídos são perdidos para sempre! Então, tenha cuidado com este comando.

Delete directory: Windows Powershell, OS X and Linux

command-line

```
$ rm -r practice
```

Delete directory: Windows Command Prompt

command-line

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

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

Check deletion: OS X and Linux

command-line

```
$ ls
```

Check deletion: Windows

command-line

```
> dir
```

Saindo

Isso é tudo, por enquanto! Agora você pode fechar a janela do terminal, mas vamos fazer do jeito hacker, né? :)

Exit: OS X and Linux

command-line

```
$ exit
```

Exit: Windows

command-line

```
> 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
cd	pwd	Mostra o diretório atual	cd (Windows) ou pwd (Mac OS / Linux)
dir	ls	Lista as pastas e/ou 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
rmdir (ou del)	rm	Exclui arquivo	del c:\test\test.txt
rmdir /S	rm -r	Exclui diretório	rm -r testdirectory

Esses são apenas alguns dos comandos que você pode rodar na sua linha de comando, mas não vamos usar mais do que isso hoje.

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

Pronta?

Vamos mergulhar no Python!

Vamos começar com o Python

Finalmente chegamos aqui!

Mas primeiro, vamos contar o que é Python. Python é uma linguagem de programação muito popular que pode ser usada para criar sites, jogos, software científicos, gráficos e muito, muito mais.

O Python foi criado na década de 1980 e seu principal objetivo é ser legível por seres humanos (e não apenas pelas máquinas!). Por isso ele parece mais simples que outras linguagens de programação, mas não se preocupe - o Python também é muito poderoso!

Instalação do Python

Observação: Se você está usando um Chromebook, pule este capítulo e certifique-se de seguir as instruções para [Configuração do Chromebook](#).

Observação: Se você já passou pelas etapas de Instalação, não precisa fazer isso novamente -- pode seguir em frente e ir para o próximo capítulo!

Para leitoras em casa: esse capítulo é abordado no vídeo [Instalando Python & Editor de Código](#).

Esta seção baseia-se em tutoriais da Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>)

Django é escrito em Python. Precisamos dele para fazer qualquer coisa no Django. Por isso, vamos começar instalando o Python! Nós precisamos que você instale o Python 3.6. Se você tiver alguma versão mais antiga, é preciso atualizá-la.

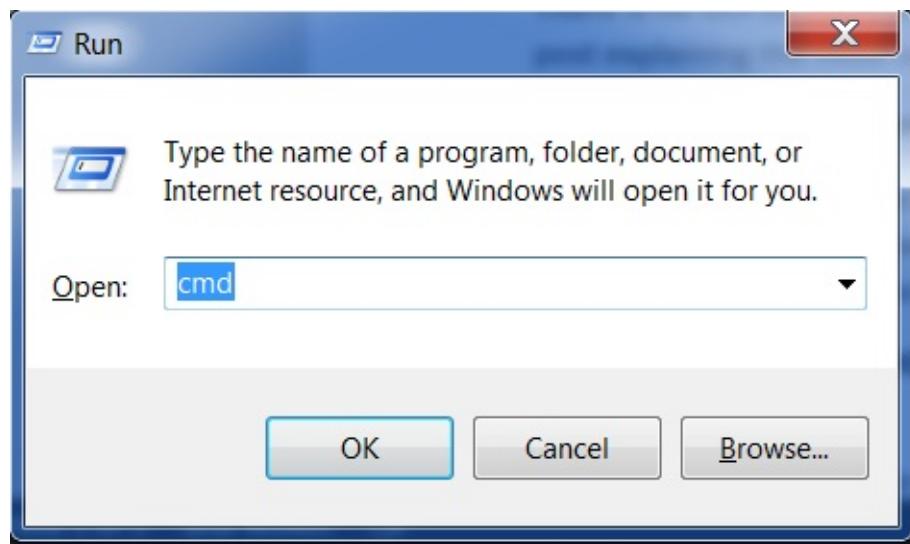
Install Python: Windows

Primeiro, verifique se o computador está executando a versão 32-bit ou a versão 64-bit do Windows. Faça isso pressionando a tecla do windows no seu teclado (aquela com a janela do windows) + a tecla Pause/Break. Feito isso, uma tela abrirá com as informações do seu windows. Nessa tela, verifique a seção "Tipo de sistema" e confira que versão está sendo executada. Você pode baixar o Python para Windows no website <https://www.python.org/downloads/windows/>. Clique no link: "Versão Mais Recente Python 3 - Python x.x.x". Se seu computador está executando a versão **64-bit** do windows, baixe o **instalador executável do Windows x86-64**. Caso contrário, baixe o **instalador executável x86 do Windows**. Depois de baixar o instalador, você precisa executá-lo (dando um duplo-clique nele) e seguir as instruções.

Existe uma coisa com a qual você precisa ter cuidado: durante a instalação, você verá uma janela marcada como "Setup". Certifique-se de selecionar a caixa "Adicionar Python 3.6 ao CAMINHO" e clique em "Instalar agora", conforme mostrado aqui:



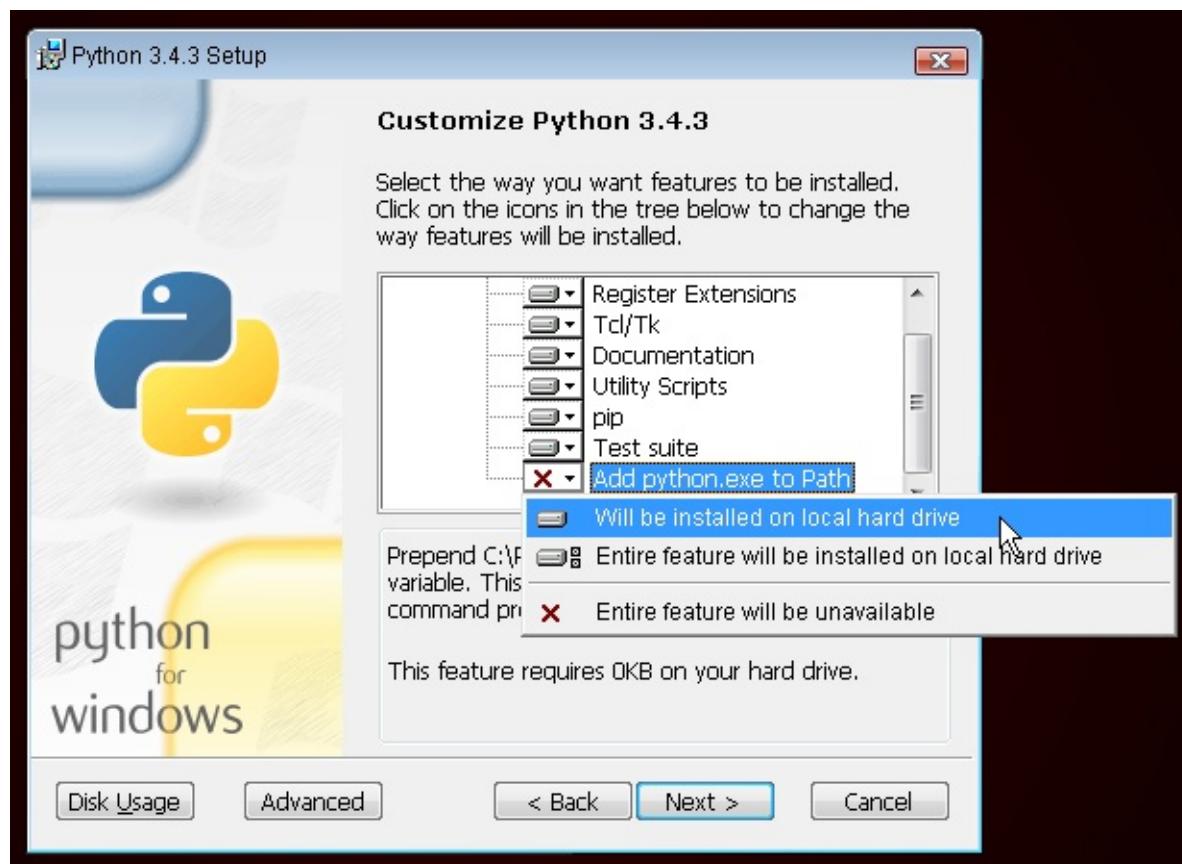
Nas próximas etapas, você usará a linha de comando do Windows (vamos te explicar tudo sobre isso também). Por enquanto, se você precisa digitar alguns comandos, vá ao menu iniciar e digite "Command Prompt" no campo de busca. (Em versões mais antigas do Windows, é possível iniciar a linha de comando com Start menu → Sistema do Windows → Prompt de comando.) Você também pode segurar a tecla windows + "R" até aparecer a janela "Executar". Para abrir a Linha de Comando, digite "cmd" e pressione enter na janela "Executar".



<0>Observação: se você estiver usando uma versão antiga do Windows (7, Vista, ou qualquer outra mais antiga) e o instalador do 3.6. x Python falhar com um erro, você pode tentar:

1. instalar todas as atualizações do Windows e tentar instalar o Python 3.6 novamente; ou
2. instalar uma [versão mais antiga do Python](#), por exemplo, [3.4.6](#).

Se você instalar uma versão mais antiga do Python, a tela de instalação pode parecer um pouco diferente da mostrada acima. Certifique-se de rolar até ver a opção "Add python.exe to Path", então clique no botão à esquerda e escolha "Will be installed on local hard drive":



Install Python: OS X

Observação: Antes de instalar o Python no Mac OS X, você deve garantir que suas configurações permitam a instalação de pacotes que não estejam na App Store. Vá para preferências do sistema (dentro da pasta Aplicativos), clique em "Segurança & Privacidade" e depois na guia "Geral". Se a configuração "Permitir que apps baixados:" estiver definida como "Mac App Store," mude para "Mac App Store e desenvolvedores identificados."

Você precisa visitar <https://www.python.org/downloads/release/python-361/> e baixar o instalador do Python:

- Faça o download do arquivo *Mac OS X 64-bit/32-bit installer*,
- Dê um duplo clique no arquivo *python-3.6.1-macosx10.6.pkg* para executar o instalador.

Install Python: 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 o terminal e digite o seguinte comando:

command-line

```
$ python3 --version
Python 3.6.1
```

Se você tem instalada uma outra "versão micro" do Python, por exemplo, 3.6.0, você não precisa atualizá-la. Se você não tiver o Python instalado ou quiser uma versão diferente, faça assim:

Install Python: Debian or Ubuntu

Digite o seguinte comando no terminal:

command-line

```
$ sudo apt install python3.6
```

Install Python: Fedora

Use o seguinte comando no terminal:

command-line

```
$ sudo dnf install python3
```

Se você estiver em versões mais antigas do Fedora, pode receber um erro dizendo que o comando `dnf` não foi encontrado. Nesse caso, você precisa usar o `yum` em vez disso.

Install Python: openSUSE

Use o seguinte comando no terminal:

command-line

```
$ sudo zypper install python3
```

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

command-line

```
$ python3 --version  
Python 3.6.1
```

Observação: Se você estiver no Windows e receber uma mensagem de erro dizendo que o `python3` não foi encontrado, tente utilizar `python` (sem o `3`) e verifique se ela corresponde à versão Python 3.6.

Se você tem alguma dúvida ou se alguma coisa deu errado e você não tem a menor ideia do que fazer, pergunte à sua monitora! Nem sempre tudo sai conforme o esperado e é melhor pedir ajuda a alguém mais experiente.

Editor de Código

Para leitoras em casa: esse capítulo é abordado no vídeo [Instalando Python & Editor de Código](#).

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

Observação: Se você estiver usando um Chromebook, pule este capítulo e certifique-se de seguir as instruções para a [Configuração do Chromebook](#).

Observação: Você talvez já tenha feito isso anteriormente no capítulo de Instalação - se for o caso, pode pular direto para o próximo capítulo!

Existem muitos editores de código diferentes e escolher um trata-se essencialmente de preferência pessoal. A maioria dos programadores de Python usa os complexos, mas extremamente poderosos IDEs (Integrated Development Environments, ou, em português, Ambientes de Desenvolvimento Integrado), tais como o PyCharm. Para um iniciante, entretanto, estas IDEs não são muito adequadas; nossas recomendações são igualmente poderosas, mas bem mais simples.

Nossas sugestões estão logo abaixo, mas sinta-se livre para perguntar à sua monitora quais são suas preferências - será mais fácil escolher com a ajuda dela.

Gedit

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

[Baixe-o aqui](#)

Sublime Text 3

O Sublime Text é um editor muito popular. Ele tem um período de avaliação gratuito e está disponível para vários sistemas operacionais.

[Baixe-o aqui](#)

Atom

O Atom é um editor de código recente criado pelo [GitHub](#). Ele é gratuito, tem código aberto e está disponível para Windows, OS X e Linux.

[Baixe-o aqui](#)

Por que estamos instalando um editor de código?

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

A primeira razão é que o código precisa estar em **texto puro, sem formatação**, e o problema com programas como o Word e o Textedit é que eles não produzem texto puro de fato. Eles produzem texto rico (com fontes e formatação), usando formatos personalizados, como [RTF](#) (Rich Text Format, ou Formato de Texto Rico, em português).

A segunda razão é que editores de código são especializados em edição de código, então eles podem fornecer funcionalidades úteis, como destacar o código com cores de acordo com seu significado ou fechar aspas automaticamente para você.

Veremos tudo isso em ação mais pra frente. Logo, logo, seu bom e velho editor de código será uma de suas ferramentas preferidas. :)

Introdução ao Python

Parte deste capítulo é baseada no tutorial da Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Vamos escrever um pouco de código!

Interpretador de Python

Para as leitoras em casa: esta parte é abordada no vídeo [Python Basics: Integers, Strings, Lists, Variables and Errors](#).

Para começar a trabalhar com Python, precisamos abrir uma *linha de comando* no computador. Você provavelmente já sabe como fazer isso - aprendeu no capítulo [Introdução à Linha de Comando](#).

Assim que estiver pronta, siga as instruções abaixo.

Queremos abrir um console do Python, então digite `python` no Windows ou `python3` no Mac OS/Linux e pressione `enter`.

command-line

```
$ python3
Python 3.6.1 (...)
Digite "ajuda", "direitos autorais", ou "licença" para mais informações.
>>>
```

Seu primeiro comando em Python!

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

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

Por enquanto, não queremos sair do console do Python. Queremos saber mais sobre ele. Vamos começar digitando um pouco de matemática, como `2 + 3`, e clicar `entrar`.

command-line

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

Legal! Viu como a resposta apareceu? O Python sabe matemática! Você pode tentar outros comandos como:

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

Para executar cálculo exponencial, como 2 elevado a 3, digitamos:

command-line

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

Brinque um pouco com isso 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:

command-line

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

Você acabou de criar sua primeira string! Uma string é uma sequência de caracteres que pode ser processada pelo computador. Ela deve sempre começar e terminar com o mesmo caractere. Este caractere pode ser aspas duplas (") ou simples (') (não há nenhuma diferença!). Elas dizem ao Python que o que está entre elas é uma string.

Strings podem ser juntadas. Tente isto:

command-line

```
>>> "Olá " + "Ola"  
'Olá Ola'
```

(Ola é o nome das duas criadoras do Django Girls!). Você também pode multiplicar strings por um número:

command-line

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

Se você precisa colocar uma apóstrofe dentro de sua string, pode fazê-lo de duas maneiras.

Usando aspas duplas:

command-line

```
>>> "Roda d'água"  
"Roda d'água"
```

Ou escapando a aspa simples (o que sinaliza para o Python que aquele sinal é uma apóstrofe, e não uma aspa marcando o final da string) com uma contra-barra (\`):

command-line

```
>>> "Roda d\'água"  
"Roda d'água"
```

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

command-line

```
>>> "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 tem que realizar em um determinado objeto (`"Ola"`) sempre que você o chamar.

Se você quer saber o número de letras contidas em seu nome, há uma função para isso também!

command-line

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

Talvez você esteja se perguntando porque algumas vezes chamamos funções com um `.` depois de uma string (como `"ola".upper()`) e outras vezes primeiro chamamos a função e colocamos a string entre 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**. Em 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 de 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.

Esse é o básico de todas as linguagens de programação que você aprender. Pronta para algo mais difícil? Apostamos que sim!

Erros

Vamos tentar algo novo. Será que conseguimos saber a extensão de um número da mesma forma que descobrimos a dos nossos nomes? Digite `len(304023)` e clique `enter`:

command-line

```
>>> 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! O ícone é a forma que o Python tem para avisar que o código que você está prestes a executar não vai funcionar conforme o esperado. Cometer erros (inclusive intencionalmente) é uma parte importante da aprendizagem!

Nossa primeira mensagem de erro nos diz que objetos do tipo "int" (inteiros, naturais) não têm comprimento algum. Então o que podemos fazer agora? Podemos escrever nosso número como string? Strings têm comprimento, certo?

command-line

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

Funcionou! Usamos a função `str` dentro da função `len`. A função `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 nem sempre é possível converter texto em números -- o que `int('hello')` quer dizer?

Variáveis

Variáveis são um conceito importante em programação. Uma variável é apenas um nome que você atribui a alguma coisa de tal forma que possa usá-lo mais tarde. Os programadores usam essas variáveis para armazenar dados, para tornar seus códigos mais legíveis e para não ter que se lembrar a todo momento o que são certas coisas.

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

command-line

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

Digitamos: nome igual Ola.

Como você deve ter percebido, a última linha de código não retornou nada como nos exemplos anteriores. Então como vamos saber se a variável realmente existe? Basta digitar `name` e clicar `enter` :

command-line

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

Uhuuu! Sua primeira variável! :) Você sempre pode mudar o valor dela:

command-line

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

Você pode usá-la também em funções:

command-line

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

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

command-line

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

E se digitarmos errado o nome da variável? Você consegue imaginar o que aconteceria? Vamos tentar!

command-line

```
>>> 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, o Python tem diferentes tipos de erros e este é chamado **NameError**. O Python mostrará esta mensagem de erro se você tentar usar uma variável que ainda não foi definida. Se você encontrar esse erro mais tarde, confira no seu código se você não digitou errado o nome de uma variável.

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

A função print

Tente o seguinte:

command-line

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

Quando você digita `name`, o interpretador de Python responde com a *representação* da variável 'name' na forma de string, que é sequência de letras M-a-r-i-a, entre aspas simples. Quando você disser para o Python `print(name)`, ele vai "imprimir" o conteúdo da variável na tela sem as aspas, que é mais bonitinho. :)

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 números inteiros, o Python tem muitos tipos diferentes de objetos. Agora vamos apresentar um chamado **lista**. Listas são exatamente o que você imagina: objetos que são listas de outros objetos. :)

Vá em frente e crie uma lista:

command-line

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

Sim, esta é uma lista vazia. Não é muito útil, né? Vamos criar uma lista de números de loteria. Para não precisar repetir o código o tempo todo, vamos atribuí-la a uma variável:

command-line

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

Legal, criamos uma lista! O que podemos fazer com ela? Vamos ver quantos números de loteria ela tem. Você tem ideia de qual é a função que deve usar para isso? Você já aprendeu ;)

command-line

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

Sim! `len()` pode te dizer o número de objetos que fazem parte de uma lista. Uma mão na roda, né? Agora vamos organizá-los:

command-line

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

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

command-line

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

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

E se quisermos inverter a ordem? Vamos fazer isso!

command-line

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

Moleza, né? Se você quiser adicionar alguma coisa à sua lista, digite o seguinte comando:

command-line

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

Se você quiser ver apenas o primeiro número da lista, pode usar **índices**. Um índice é o número que diz onde na lista um item está. Programadores preferem começar a contar a partir do zero, então o primeiro objeto em sua lista está no índice 0, o segundo no 1 e assim por diante. Tente isso:

command-line

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

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

Para apagar algum objeto da sua lista, você precisa usar **índices**, como aprendemos acima, e o método `pop()`. Vamos usar um exemplo para reforçar o que já aprendemos: vamos deletar o primeiro número de nossa lista.

command-line

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

Funcionou perfeitamente!

Agora tente alguns outros índices, como: 6, 7, 1000, -1, -6 ou -1000. Veja se você consegue prever o resultado antes de executar o comando. Os resultados fazem sentido para você?

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

Para leitoras em casa: este capítulo é abordado no vídeo [Python Basics: Dictionaries](#).

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

command-line

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

Você acabou de criar um dicionário vazio. Uhuuu!

Agora escreva o seguinte comando (e tente colocar suas próprias informações):

command-line

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

Com esse comando, você criou uma variável chamada `participant` com três pares de chave-valor:

- A chave `name` aponta para o valor `'Ola'` (um objeto `string`),
- a chave `country` aponta para `'Poland'` (outra `string`),
- e a chave `favorite_numbers` aponta para `[7, 42, 92]` (uma `list` de três números).

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

command-line

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

É similar a uma lista, mas você não precisa lembrar o índice - apenas o nome.

O que acontece se perguntarmos ao Python qual é o valor de uma chave que não existe? Você consegue adivinhar? Vamos testar e descobrir!

command-line

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

Olha, outro erro! Esse é um `KeyError`. O Python é bastante prestativo e te diz que a chave `'age'` não existe no nesse dicionário.

Você deve estar se perguntando quando deve usar um dicionário ou uma lista, certo? Boa pergunta! A resposta rápida é:

- Você precisa de uma sequência ordenada de itens? Use uma lista.
- Você precisa associar valores a chaves para poder procurá-los eficientemente (pela chave) mais tarde? Use um dicionário.

Dicionários, assim como listas, são *mutáveis*. Isso significa que eles podem ser alterados depois de criados. Você pode adicionar um novo par chave-valor a um dicionário depois de ele ser criado, por exemplo:

command-line

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

Como nas listas, usar a função `len()` em dicionários retorna o número de pares chave-valor contidos nele. Vá em frente e digite o comando:

command-line

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

Espero que esteja fazendo sentido até agora. :) Pronta para mais diversão com dicionários? Pule para a próxima linha para aprender mais coisas incríveis.

Você pode usar o método `pop()` para deletar um item do dicionário. Digamos que você queira excluir a entrada correspondente à chave `'favorite_numbers'`. Basta digitar o seguinte comando:

command-line

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

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

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

command-line

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

Agora, o valor da chave `'country'` foi alterado de `'Poland'` para `'Germany'`. :) Emocionante? Uhu! 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 mensagens de erro que aparecem quando o Python não entende um comando que você deu;
- **variáveis** -- nomes para objetos que permitem que você programe facilmente e deixam seu código mais legível;
- **listas** -- listas de objetos armazenados em uma ordem específica;
- **dicionários** - objetos armazenados como pares chave-valor.

Animada para a próxima parte? :)

Compare coisas

Para leituras em casa: esta seção é abordada no vídeo [Python Basics: Comparisons](#).

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

command-line

```
>>> 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, ele pode comparar não apenas números, mas também resultados de métodos. Legal, né?

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

Dê ao Python mais duas tarefas:

command-line

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

Já vimos `>` e `<`, mas o que significam `>=` e `<=`? Leia 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 fazer mais? Tente isto:

command-line

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

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

- `and` -- se você usar o operador `and`, as duas comparações terão que ser verdadeiras para que a expressão seja verdadeira (`True`)
- `or` - se você usar o operador `or`, apenas uma das comparações precisa ser verdadeira para que a expressão seja verdadeira (`True`)

Já ouviu a expressão "comparar alhos com bugalhos"? Vamos tentar o equivalente em Python:

command-line

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

Aqui vemos que assim como não podemos comparar alhos com bugalhos, o Python não é capaz de comparar um número (`int`) e uma string (`str`). Em vez de retornar um resultado, ele mostrou um **TypeError** e nos disse que os dois tipos não podem ser comparados um ao outro.

Booleanos

Aliás, você acabou de aprender sobre um novo tipo de objeto em Python. Ele se chama **booleano**.

Existem apenas dois objetos booleanos:

- `True` (verdadeiro)
- `False` (falso)

Para que o Python entenda, você precisa escrever exatamente 'True' (primeira letra maiúscula e as outras minúsculas -- mas sem as aspas). **true, TRUE ou tRUE não vão funcionar -- só True está correto.** (A mesma coisa vale para 'False', obviamente.)

Booleanos também podem ser variáveis! Veja:

command-line

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

Você também pode fazer desse jeito:

command-line

```
>>> 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!

Salve!

Para leitoras em casa: esta seção é abordada no vídeo [Python Basics: Saving files and "If" statement](#).

Até agora, escrevemos todos os códigos no interpretador de Python, que nos limita a digitar uma linha por vez. Programas normais são salvos em arquivos e executados pelo nosso **interpretador** de linguagem de programação ou **compilador**.

Até aqui, executamos nossos programas uma linha de cada vez no **interpretador** de Python. Vamos precisar de mais de uma linha de código para as próximas tarefas, então precisaremos rapidamente:

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

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

command-line

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

Isso vai levá-la de volta ao prompt de comando.

Mais cedo, nós escolhemos um editor de código da seção [editor de código](#). Agora, precisamos abrir o editor e escrever algum código em um novo arquivo:

editor

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

Agora você é uma desenvolvedora Python bastante experiente, então sinta-se livre para escrever códigos com o que aprendeu hoje.

Agora precisamos salvar o arquivo e dar a ele um nome descritivo. Vamos nomear o arquivo **python_intro.py** e salvá-lo na sua área de trabalho. Podemos chamá-lo como quisermos, mas é importante que o nome termine com **.py**. A extensão **.py** diz ao sistema operacional que esse é um **arquivo Python executável** e o interpretador de Python pode rodá-lo.

Observação: Você deve reparar numa das coisas mais legais nos editores de código: cores! No interpretador de Python tudo é da mesma cor, mas agora você deve estar vendo que a função `print` tem uma cor diferente da string que ela recebe como argumento. Isso se chama destaque de sintaxe ("syntax highlighting", do Inglês) e é uma funcionalidade muito útil quando escrevemos código. As cores de cada elemento nos dão dicas sobre o código. Elas avisam, por exemplo, sobre strings que esquecemos de fechar ou palavras reservadas que digitamos errado (como a palavra `def` na definição de uma função, que veremos adiante). Esta é uma das razões pelas quais usamos um editor de código. :)

O arquivo está salvo, então é hora de executá-lo! Com as habilidades que você aprendeu na seção sobre linhas de comando, use o terminal para **ir para os diretórios no desktop**.

Change directory: OS X

Em um Mac, o comando é esse:

command-line

```
$ cd ~/Desktop
```

Change directory: Linux

No Linux, o comando é assim (a palavra "Desktop" pode estar traduzida para o português "Área de Trabalho"):

command-line

```
$ cd ~/Desktop
```

Change directory: Windows Command Prompt

No prompt de comando Windows, é assim:

command-line

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

Change directory: Windows Powershell

E no Windows Powershell, é assim:

command-line

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

Se você tiver alguma dificuldade, é só pedir ajuda.

Agora use o interpretador de Python para executar o código que está no arquivo, assim:

command-line

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

Observação: no Windows, 'python3' não é reconhecido como um comando. Em vez disso, use 'python' para executar o arquivo:

command-line

```
> python python_intro.py
```

Muito bem! Você acabou de rodar seu primeiro programa em Python que foi salvo em um arquivo. Arrasou!

Agora, você pode começar a aprender uma ferramenta essencial na programação:

If ... elif ... else

Várias coisas em um código só podem ser executadas se determinadas condições forem atendidas. É por isso que o Python tem um comando chamado **if**.

Substitua o código no arquivo **python_intro.py** pelo seguinte:

python_intro.py

```
if 3 > 2:
```

Se você salvar e executar esse código, verá um erro como este:

command-line

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

O Python espera receber de nós instruções que devem ser executadas caso a condição `3 > 2` seja verdadeira (ou `True`). Vamos tentar fazer o Python mostrar na tela "Funciona!". Altere o código no seu arquivo **python_intro.py** para o seguinte:

python_intro.py

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

Notou que o texto na linha seguinte ao "if" começa depois de quatro espaços? Chamamos esse tipo de formatação de endentação. Precisamos dessa endentação para que o Python saiba quais linhas executar se a condição dentro do if for verdadeira. Você pode usar quantos espaços quiser, mas para que os códigos tenham uma aparência mais limpa e organizada, os programadores de Python costumam usar quatro. Uma tabulação conta como quatro espaços se você configurar seu editor de texto assim. Quando escolher quantos espaços usar, não mude! Se você já começou a endentar com quatro espaços, siga esse padrão em todo o código -- ou você poderá encontrar problemas.

Salve o código e execute outra vez:

command-line

```
$ python3 python_intro.py
Funciona!
```

Observação: Lembre-se que no Windows 'python3' não é reconhecido como um comando. Se você usa esse sistema operacional, de agora em diante, substitua 'python3' 'python' para executar o arquivo.

E se uma condição não for verdadeira?

Nos exemplos anteriores, o código foi executado somente quando as condições eram verdadeiras. No entanto, o Python também tem as instruções `elif` e `mais`:

`python_intro.py`

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

Quando esse código for executado, o Python mostrará:

command-line

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

Se 2 fosse um número maior que 5, o segundo comando seria executado. Fácil, né? Vamos ver como funciona o `elif`:

`python_intro.py`

```
name = 'Sonja'
if name == 'Ola':
    print('Olá Ola!')
elif name == 'Sonja':
    print('Olá Sonja!')
else:
    print('Olá estranho!')
```

e executando:

command-line

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

Viu o que aconteceu? O `elif` possibilita adicionar uma condição que só vai ser executada se a primeira condição for falsa.

Você pode adicionar quantos `elif` quiser depois do `if`. Por exemplo:

`python_intro.py`

```

volume = 57
if volume < 20:
    print("Está um pouco baixo")
elif 20 <= volume < 40:
    print("Está bom para música ambiente")
elif 40 <= volume < 60:
    print("Perfeito, posso ouvir todos os detalhes")
elif 60 <= volume < 80:
    print("Ótimo para festas!")
elif 80 <= volume < 100:
    print("Está muito alto!")
else:
    print("Meus ouvidos doem! :(")

```

O Python testa cada condição na sequência em que aparece no código e então mostra:

command-line

```
$ python3 python_intro.py
Perfeito, posso ouvir todos os detalhes
```

Comentários

Comentários são linhas que começam com `#`. Você pode escrever o que quiser após o `#` e o Python vai ignorar. Comentários podem tornar seu código mais fácil para outras pessoas entenderem.

Vamos ver como isso funciona:

`python_intro.py`

```

# Mudar o volume se estiver muito alto ou muito baixo
if volume < 20 or volume > 80
    volume = 50
    print("Bem melhor!")

```

Você não precisa escrever um comentário para cada linha de código, mas eles são úteis para explicar porque o seu código faz alguma coisa ou para fornecer um breve resumo de algo mais complexo.

Sumário

Nos últimos exercícios você aprendeu:

- **a comparar coisas** -- em Python, você pode comparar objetos usando os operadores `>`, `>=`, `==`, `<=`, `<` e `and`, `or`.
- **Booleano** -- um tipo de objeto que só tem dois valores possíveis: `True` ou `False`.
- **a salvar arquivos** -- armazenar código em arquivos para que você possa executar programas maiores.
- **if... elif... else** -- instruções que permitem que você execute o código somente se determinadas condições forem atendidas.
- **comentários** -- linhas que o Python não executa e que permitem que você documente seu código.

Chegamos à última parte do capítulo!

Suas próprias funções!

Para leitoras em casa: esta parte do capítulo é abordada no vídeo [Python Basics: Functions](#).

Lembra de funções como `len()`? Boas notícias: agora você vai aprender como escrever suas próprias funções!

Uma função é uma sequência de instruções que o Python deve executar. Cada função em Python começa com a palavra reservada `def` seguida de um nome e, opcionalmente, de uma lista de parâmetros. Vamos fazer uma tentativa. Substitua o código salvo no arquivo `python_intro.py` pelo seguinte:

`python_intro.py`

```
def oi():
    print('Olá!')
    print('Tudo bem?')

oi()
```

Ok, nossa primeira função está pronta!

Reparou que escrevemos o nome da função no começo e no final do código? O Python lê e executa o código de cima para baixo, então para usar a função que definimos, devemos chamá-la no final. Esclarecendo: no bloco de texto formado pela linha que começa com "def" e as linhas endentadas seguintes, definimos a função, mas não executamos nenhum comando. Precisamos dizer explicitamente ao Python que a execute (escrevendo "oi()").

Vamos executá-la agora e ver o que acontece:

command-line

```
$ python3 python_intro.py
Olá!
Tudo bem?
```

Observação: se não funcionou, não entre em pânico! A saída vai te ajudar a entender o que aconteceu:

- Se você recebeu uma mensagem `NameError`, provavelmente foi um erro de digitação, então confira se você usou o mesmo nome ao criar a função com `def hi()` e quando a chamou no final com `hi()`.
- Se recebeu uma mensagem `IndentationError`, confira se as duas linhas de `print` têm o mesmo recuo no começo: o Python precisa que o código dentro da função esteja bem alinhado.
- Se a função não retornou nenhum resultado, certifique-se de que o último `oi()` não esteja endentado - se ele estiver, essa linha vai se tornar parte da função e o Python não vai receber nenhum comando para executar.

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

`python_intro.py`

```
def oi(nome):
```

Como você pode ver, agora nossa função tem um parâmetro chamado `nome`:

`python_intro.py`

```
def oi(nome):
    if nome == 'Ola':
        print('Olá Ola!')
    elif nome == 'Sonja':
        print('Olá Sonja!')
    else:
        print('Olá estranho!')

hi()
```

Não esqueça: a função `print` está endentada com 4 espaços depois do `if`. Isso é necessário porque a função só rodará se a condição for verdadeira. Vamos ver como isso funciona:

command-line

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

Oops, um erro. Felizmente, o Python nos fornece uma mensagem de erro bastante útil. Ela diz que a função `oi()` (aquele que definimos) tem um argumento obrigatório (chamado `nome`) e que nós esquecemos de passá-lo ao chamar a função. Vamos resolver isso no final da função:

python_intro.py

```
oi("Ola")
```

E rode novamente:

command-line

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

E se mudarmos o nome?

python_intro.py

```
oi("Sonja")
```

E rode novamente:

command-line

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

Agora, o que você acha que aconteceria se escrevesse um outro nome lá (diferente de "Ola" ou "Sonja")? Faça um teste e verifique se você estava certa. Deve aparecer o seguinte:

command-line

```
Olá estranho!
```

Legal, né? Dessa maneira você não precisa se repetir cada vez que for mudar o nome da pessoa que a função pretende cumprimentar. E é exatamente por isso que precisamos de funções - você não quer precisar repetir seu código!

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

python_intro.py

```
def oi(name):
    print('Olá ' + name + '!')

oi("Rachel")
```

Vamos chamar o código agora:

command-line

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

Parabéns! Você acabou de aprender como criar funções. :)

Laços

Para leitoras em casa: este capítulo é abordado no vídeo [Python Basics: For Loop](#).

Essa já é a última parte! Rápido, né? :)

Programadores não gostam de repetir código. Programar é automatizar coisas, então não queremos cumprimentar cada pessoa manualmente, certo? É aí que entram os laços (ou "loops", em Inglês).

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

`python_intro.py`

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

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

`python_intro.py`

```
for name in girls:
```

A instrução `for` se comporta de maneira similar ao `if`; o código abaixo de qualquer uma destas instruções deve ser indentado com quatro espaços.

Aqui está o código completo que será salvo no arquivo:

`python_intro.py`

```
def oi(nome):
    print('Olá ' + nome + '!')

girls = ['Rachel', 'Monica', 'Phoebe', 'Ola', 'você']
for name in girls:
    oi(name)
    print('Próxima')
```

E quando rodamos:

command-line

```
$ python3 python_intro.py
Olá Rachel!
Próxima
Olá Monica!
Próxima
Olá Phoebe!
Próxima
Olá Ola!
Próxima
Olá você!
Next girl
```

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

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

`python_intro.py`

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

Que deve imprimir:

command-line

```
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)` ; ele conta de 1 a 5, mas não inclui o 6). Isso acontece porque o intervalo é semi-aberto, o que significa que ele inclui o primeiro valor, mas não o último.

Sumário

É isso. **Arrasou!** Esse foi um capítulo difícil, então você deve estar orgulhosa. Nós estamos orgulhosas de você por ter conseguido ir tão longe!

Para um tutorial de Python oficial e completo, visite <https://docs.python.org/3/tutorial/>. Lá você encontrará um guia da linguagem mais exaustivo e completo. Até lá :)

Talvez você queira fazer uma breve pausa -- se espreguiçar, andar um pouco, descansar os olhos -- antes de ir para o próximo capítulo. :)



O que é Django?

Django (/dʒæŋgəʊ/ *jang-goh*) é um framework para aplicações web gratuito e de código aberto, escrito em Python. Um web framework é um conjunto de componentes que ajuda você a desenvolver sites de forma mais rápida e fácil.

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

Felizmente, outras pessoas perceberam muito tempo atrás que desenvolvedores web enfrentam problemas similares quando estão construindo um novo site, então elas se juntaram e criaram frameworks (Django é um deles) que te dão componentes já prontos para usar.

Frameworks existem para evitar que você tenha que reinventar a roda e para ajudar a aliviar parte do trabalho extra quando você está construindo um novo site.

Por que você precisa de um framework?

Para entender para quê o Django de fato serve, precisamos olhar os servidores mais de perto. Primeiramente, o servidor precisa saber que você quer que ele te sirva uma página web.

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

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

Quando uma requisição chega a um servidor web, ela é passada para o Django, que tenta entender o que está sendo requisitado de fato. Primeiro, ele pega um endereço de página web e tenta descobrir o que fazer. Isso parte é feito pelo **urlresolver** do Django (note que um endereço de website é chamado de URL - Uniform Resource Locator -- Localizador de Recursos Uniforme, em tradução livre --, então o nome *urlresolver* faz sentido). Ele não é muito inteligente - ele pega uma lista de padrões e tenta achar em qual deles a URL se encaixa. O Django checa os padrões de cima pra baixo, e se algum se encaixa, ele passa a requisição para a função associada (que é chamada de *view*).

Imagine uma funcionária dos Correios com uma carta. Ela está andando pela rua e compara cada número de casa com o que está na carta. Se os números forem correspondentes, ela entrega a carta lá. É assim que funciona o urlresolver!

Todas as coisas interessantes são feitas dentro da função *view*: podemos dar uma olhada no banco de dados para procurar algumas informações. O usuário solicitou alguma mudança nos dados? Como uma carta dizendo "Por favor mude a descrição do meu emprego." A *view* pode checar se você tem permissão para fazer isso, e então atualizar a descrição do emprego e enviar de volta a mensagem: "Pronto!" Então a *view* gera uma resposta e o Django pode enviá-la para o navegador web do usuário.

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, vamos simplesmente começar criando algo com o Django e assim aprender toda as partes importantes ao longo do caminho!

Instalação do Django

Observação: Se você está usando um Chromebook, pule este capítulo e certifique-se de seguir as instruções para [Configuração do Chromebook](#).

Observação: Se você já seguiu o passo a passo da instalação, pode ignorar essa seção - vá direto para o próximo capítulo!

Esta seção baseia-se em tutoriais da Geek Girls Carrots (<https://github.com/ggcarrots/django-carrots>).

Parte deste capítulo é baseada em [tutorial django-marcador](#) licenciado sob 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 instalar o Django, vamos instalar uma ferramenta muito útil para ajudar a manter o ambiente de trabalho no nosso computador organizado. Você pode pular esse passo, mas ele é altamente recomendado. Começar com a melhor instalação possível poupará você de muito trabalho no futuro!

Vamos criar um **ambiente virtual** (também chamado um *virtualenv*). O *virtualenv* isolará seu código Python/Django em um ambiente organizado por projetos. Isso significa que as alterações que você fizer em um website não afetarão os outros projetos que você estiver desenvolvendo ao mesmo tempo. Legal, né?

Tudo o que você precisa fazer é encontrar o diretório em que você quer criar o `virtualenv`; seu diretório Home, por exemplo. No Windows, pode aparecer como `c:\users\Name` (onde `Nome` é seu usuário de login).

Observação: No Windows, certifique-se de que esse diretório não contém palavras acentuadas ou caracteres especiais; se o seu usuário contém caracteres acentuados, use um diretório diferente, por exemplo: `c:\djangogirls`.

Para este tutorial, usaremos um novo diretório `djangogirls` no seu diretório home:

command-line

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

Vamos fazer um *virtualenv* chamado `meuenv`. O formato geral desse comando é:

command-line

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

Virtual environment: Windows

Para criar um novo `virtualenv`, você deve abrir o terminal e executar `python -m venv myvenv`. Deve ficar assim:

command-line

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

Onde `myvenv` é o nome do seu *virtualenv*. Você pode usar qualquer outro nome, mas sempre use minúsculas e não use espaços, acentos ou caracteres especiais. Também é uma boa ideia manter o nome curto - você irá referenciá-lo muitas vezes!

Virtual environment: Linux and OS X

Podemos criar um `virtualenv` no Linux ou no OS X executando `python3 -m venv myvenv`. Deve ficar assim:

command-line

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

`myvenv` é o nome do seu `virtualenv`. Você pode usar qualquer outro nome, mas use sempre letras minúsculas e não use espaços entre as palavras. Também é uma boa ideia manter o nome curto pois você vai escrevê-lo muitas vezes!

Observação: Em algumas versões do Debian/Ubuntu, você pode receber o seguinte erro:

command-line

```
The virtual environment was not created successfully because ensurepip is not available. On Debian/Ubuntu systems, you need to install the python3-venv package using the following command.  
apt install python3-venv  
You may need to use sudo with that command. After installing the python3-venv package, recreate your virtual environment.
```

Caso você receba esse erro, siga as instruções acima e instale o pacote `python3-venv`:

command-line

```
$ sudo apt install python3-venv
```

Observação: Em algumas versões do Debian/Ubuntu, iniciar o ambiente virtual com este comando gera o seguinte erro:

command-line

```
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`.

command-line

```
$ sudo apt install python-virtualenv  
$ virtualenv --python=python3.6 myvenv
```

Osservação: Se você obtiver um erro como

command-line

```
E: Unable to locate package python3-venv
```

no lugar do comando mostrado acima, execute esse:

command-line

```
sudo apt install python3.6-venv
```

Trabalhando com o `virtualenv`

O comando acima criará um diretório chamado `myvenv` (ou qualquer que seja o nome que você escolheu) que contém o nosso ambiente virtual (basicamente um conjunto de diretórios e arquivos).

Working with `virtualenv`: Windows

Inicie o seu ambiente virtual executando:

command-line

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

Observação: no Windows 10, você pode obter um erro no Windows PowerShell que diz `execution of scripts is disabled on this system`. Neste caso, abra uma outra janela do Windows PowerShell com a opção de "Executar como Administrador". Assim, execute o comando abaixo antes de iniciar o seu ambiente virtual:

command-line

```
C:\WINDOWS\system32> Set-ExecutionPolicy -ExecutionPolicy RemoteSigned
Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might expose you to the security risks described in the about_Execution_Policies help topic at http://go.microsoft.com/fwlink/?LinkID=135170. Do you want to change the execution policy? [Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): A
```

Working with virtualenv: Linux and OS X

Inicie o seu ambiente virtual executando:

command-line

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

Lembre-se de substituir `myvenv` pelo nome que você escolheu para o `virtualenv`!

Observação: às vezes `source` pode não estar disponível. Nesses casos, tente fazer isso:

command-line

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

Você vai saber que tem um `virtualenv` funcionando quando vir que a linha de comando no seu console tem o prefixo (`myvenv`).

Ao trabalhar em um ambiente virtual, o comando `python` irá automaticamente se referir à versão correta para que você possa digitar `python` em vez de `python3`.

Pronto, já temos todas as dependências importantes no lugar. Finalmente podemos instalar o Django!

Instalando o Django

Agora que você tem seu `virtualenv` ativo, pode instalar o Django.

Antes de fazer isto, devemos garantir que temos instalada a última versão do `pip`, que é o software que usamos para instalar o Django:

command-line

```
(myvenv) ~$ python3 -m pip install --upgrade pip
```

Instalando pacotes com requisitos

O arquivo "requirements.txt" guarda as dependências que serão instaladas utilizando o `pip install`:

Primeiramente, crie um arquivo `requirements.txt` dentro da pasta `djangogirls/`:

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

E adicione o seguinte texto ao arquivo `djangogirls/requirements.txt` :

command-line

```
Django~=2.0.6
```

Agora, execute `pip install -r requirements.txt` para instalar o Django.

command-line

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

Installing Django: Windows

Se você receber um erro ao chamar o pip na plataforma Windows, verifique se o caminho do projeto contém espaços, acentos ou caracteres especiais (exemplo, `c:\Users\UserName\djangogirls`). Se sim, considere movê-lo para outro lugar sem espaços, acentos ou caracteres especiais (sugestão: `c:\djangogirls`). Crie um novo virtualenv no diretório recém-criado, exclua o mais velho e tente novamente executar o comando acima. (Mover o diretório de virtualenv não vai funcionar pois o virtualenv usa caminhos absolutos.)

Installing Django: Windows 8 and Windows 10

Sua linha de comando pode congelar depois de você tentar instalar o Django. Neste caso, ao invés do comando acima, use:

command-line

```
C:\Users\Name\djangogirls> python -m pip install -r requirements.txt
```

Installing Django: Linux

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

É isto! Você agora (finalmente) está pronta para criar uma aplicação Django!

Seu primeiro projeto Django!

Parte deste capítulo é baseada nos tutoriais da 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 é iniciar um novo projeto Django. Basicamente, isso significa que devemos rodar alguns scripts providos pelo Django que vão criar um esqueleto de projeto Django para nós. O resultado é um conjunto de diretórios e arquivos que nós iremos utilizar e modificar mais tarde.

Os nomes de alguns arquivos e diretórios são muito importantes para o Django. Você não deve renomear os arquivos que estamos prestes a criar. Mover para um lugar diferente também não é uma boa idéia. O Django precisa manter uma certa estrutura para conseguir encontrar algumas coisas importantes.

Lembre-se de rodar tudo no virtualenv. Se você não vê um prefixo `(myvenv)` em seu console, é necessário ativar o virtualenv. Nós explicamos como fazer isso no capítulo [Instalação do Django](#) na parte [Ambiente Virtual](#). Digitar `myvenv\Scripts\activate` no Windows ou `source myvenv/bin/activate` no Mac OS / Linux fará isso para você.

Create project: OS X or Linux

No MacOS ou no console do Linux, rode o comando abaixo (**não esqueça de adicionar o ponto `.` no final!):

command-line

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

O ponto `.` é crucial por que ele diz para o script instalar o Django no diretório atual (o ponto `.` é um atalho para referenciar este diretório).

Observação: Quando digitar o comando acima, lembre-se de digitar apenas a parte que começa em `django-admin`. A parte `(myvenv) ~/djangogirls$` apresentada aqui é apenas um exemplo do que pode aparecer no seu terminal quando você digitar seus comandos.

Create project: Windows

No Windows, rode o seguinte comando (**não esqueça de adicionar o ponto `.` no final!**):

command-line

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

O ponto `.` é crucial por que ele diz para o script instalar o Django no diretório atual (o ponto `.` é um atalho para referenciar este diretório).

Observação: Quando digitar o comando acima, lembre-se de digitar apenas a parte que começa em `django-admin.exe`. A parte `(myvenv) C:\Users\Name\djangogirls>` apresentada aqui é apenas um exemplo do que pode aparecer no seu terminal quando você digitar seus comandos.

`django-admin` é um script que criará os diretórios e arquivos para você. Agora, você deve ter uma estrutura de diretório parecida com isso:

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

Observação: em sua estrutura de diretórios, você também verá o o diretório do virtualenv, `venv`, que criamos antes.

`manage.py` é um script que ajuda com a gestão do site. Com ele, podemos 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 de quando falamos sobre um carteiro verificando onde entregar uma carta? O arquivo `urls.py` contém uma lista dos padrões usados por `urlresolver`.

Vamos ignorar os outros arquivos por enquanto pois não vamos modificá-los. Só precisamos lembrar de não excluí-los acidentalmente!

Mudando as configurações

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

Observação: Lembre-se de que o `settings.py` é um arquivo comum, como qualquer outro. Você pode abri-lo de dentro do editor de código usando as ações de menu "Arquivo-> Abrir". Assim, você deve encontrá-lo na janela usual para selecionar arquivos e abri-lo. Ou então, é possível abrir o arquivo navegando até o diretório do djangogirls e abrindo o arquivo com o botão direito. Uma vez clicado, selecione o seu editor de código preferido da lista. Selecionar o editor apropriado é importante uma vez que você pode ter outros programas instalados que podem abrir o arquivo, mas não editá-lo.

Para começar, seria bom ter a hora correta no nosso site. Para isto, você configurar o fuso horário correto de onde está. Se você estiver no Brasil, é bem provável que o fuso horário seja `America/Sao_Paulo` (aqui conhecido como horário de Brasília). Caso queira saber mais, vá para [Wikipedia's list of time zones](#) e copie e cole o fuso horário correspondente à sua localização.

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

`mysite/settings.py`

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

Um código de idioma se refere à língua, por exemplo, `en` para inglês ou `pt` para português e o código do país, por exemplo, `br` para Brasil ou `pt` para a Portugal. Já que o inglês provavelmente não é sua língua nativa, você pode adicionar um novo código de país para deixar os botões padrão e notificações de Django em seu idioma. Assim, você teria por exemplo um botão "Cancel" traduzido para a língua da sua escolha (ex: "Cancelar" em português). O [Django vem com um monte de traduções já preparadas](#).

Se você quiser um idioma diferente do inglês, especifique o código de idioma alterando a seguinte linha:

`mysite/settings.py`

```
LANGUAGE_CODE = 'pt-BR'
```

Também precisamos adicionar o caminho para os arquivos estáticos. (Discutiremos tudo sobre arquivos estáticos e CSS mais adiante no tutorial.) Vá até o *final* do arquivo e, logo abaixo da linha com `STATIC_URL`, adicione uma nova variável chamada `STATIC_ROOT`:

mysite/settings.py

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

Quando `DEBUG` for `True` e `ALLOWED_HOSTS` estiver vazia, o domínio do site será validado como `['localhost', '127.0.0.1', '[::1]']`. Isso não corresponderá ao nosso domínio no PythonAnywhere quando implantarmos a nossa aplicação, então vamos mudarmos a seguinte configuração:

mysite/settings.py

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

Observação: Se você estiver utilizando um Chromebook, adicione esta linha ao final do arquivo `settings.py`:

```
MESSAGE_STORAGE = 'django.contrib.messages.storage.SessionStorage'
```

Também inclua `.c9users.io` à lista de `ALLOWED_HOSTS` se você estiver utilizando o cloud9.

Configurando um banco de dados

Existem vários software de banco de dados diferentes que podem armazenar dados para o seu site. Nós vamos usar o padrão do Django, o `sqlite3`.

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

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 executar o seguinte comando 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 assim:

command-line

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

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

Iniciando o servidor web

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`:

command-line

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

Se você usa um Chromebook, execute este comando:

Cloud 9

```
(myvenv) ~/djangogirls$ python manage.py runserver 0.0.0.0:8080
```

Se você estiver no Windows e o comando falhar com `UnicodeDecodeError`, use o comando alternativo:

command-line

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

Agora, precisamos verificar se o nosso site está rodando. Abra seu browser (Firefox, Chrome, Safari, Internet Explorer ou qualquer outro que você utilizar) e digite o endereço:

browser

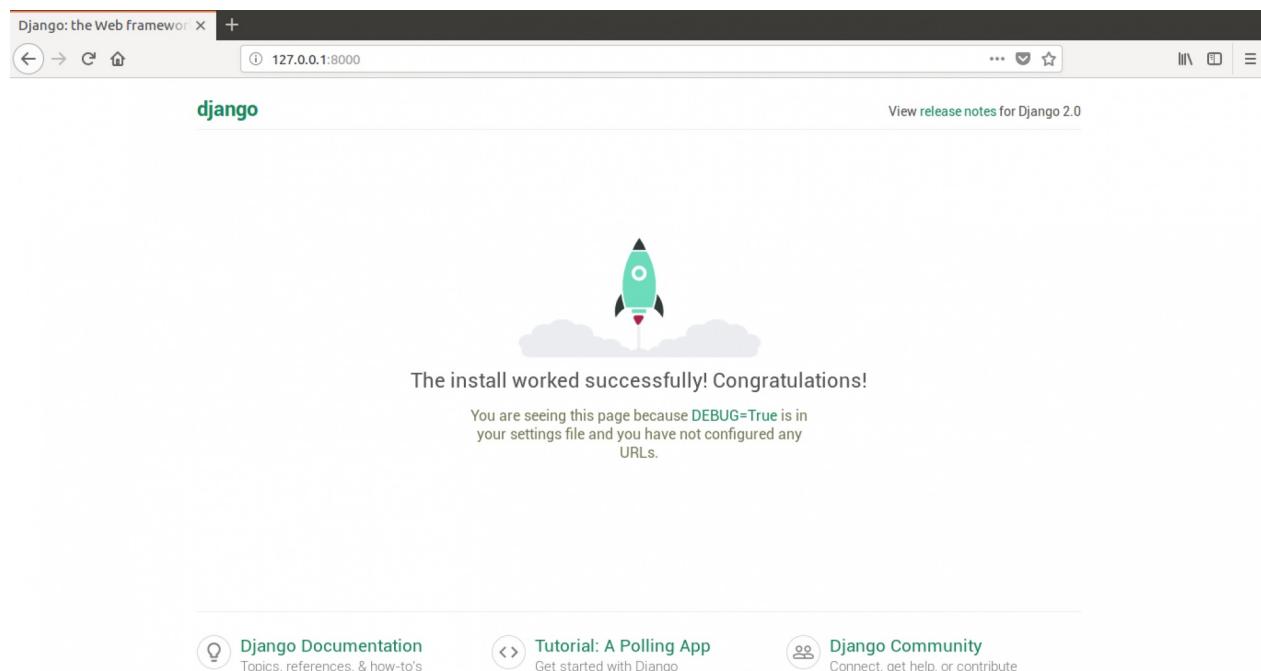
```
http://127.0.0.1:8000/
```

Se você estiver com um Chromebook, sempre visite o servidor de teste acessando:

browser

```
https://django-girls-<your cloud9 username>.c9users.io
```

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



Enquanto o servidor estiver rodando, o prompt do terminal não estará disponível para receber novos comandos. Na verdade, o terminal permite que você escreva texto, mas não irá executar nenhum comando. Isto acontece porque o servidor bloqueia o terminal enquanto ele mesmo recebe novos comandos.

Nós revisamos como servidores web funcionam no capítulo [Como a Internet funciona](#).

Para escrever novos comandos enquanto o servidor estiver rodando, abra uma nova janela do terminal e execute o virtualenv. Para interromper o seu servidor, volte para a janela onde ele está rodando e pressione CTRL+C -- botões Control e C juntos (no Windows; tente Ctrl+Break se o primeiro não funcionar).

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

Modelos do Django

Queremos criar algo que armazene todos os posts em nosso blog. Mas para fazer isto, temos que falar um pouco sobre `objects`.

Objetos

Existe um conceito em programação chamado `programação orientada a objetos`. A ideia é que ao invés de escrever tudo como uma sequência entediante de instruções de programação, possamos 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 mostrar um exemplo.

Se quisermos modelar um gato, podemos criar um objeto `Gato` que possui propriedades como `cor`, `idade`, `humor` (como bom, mal ou sonolento ;)), e `dono` (que seria atribuído a um objeto de `Pessoa` – ou talvez, no caso de um gato de rua, essa propriedade fosse vazia).

`Gato` também realiza algumas ações: `ronronar`, `arranhar` ou `alimentar -se` (no qual podemos dar ao gato alguma `ComidaDeGato`, que poderia ser um objeto separado com propriedades como `sabor`).

```
Gato
-----
cor
idade
temperamento
dono
ronronar()
arranhar()
alimentar(comida_de_gato)

ComidaDeGato
-----
sabor
```

A ideia básica é descrever coisas reais em código a partir de suas propriedades (chamadas de `atributos`) e ações (chamadas de `métodos`).

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

Para isto, precisamos responder as questões: O que é um post de blog? Que propriedades (atributos) ele deve ter?

Bem, com certeza uma postagem precisa de um texto com seu conteúdo e um título, certo? Também seria legal saber quem escreveu – então precisamos de um autor. Finalmente, queremos saber quando a postagem foi criada e publicada. Para ficar chique, vamos fazer em inglês.

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

Que tipo de ações podem ser feitas com uma postagem? Seria legal ter algum `método` que publique a postagem, não é mesmo?

Então, nós precisaremos de um método para publicar (`publish`).

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

Modelos do Django

Sabendo o que um objeto é, criamos 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. Ele 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 banco de dados padrão do Django -- e ele será o suficiente neste primeiro 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 uma aplicação separada dentro do nosso projeto. É muito bom ter tudo organizado desde o início. Para criar uma aplicação, precisamos executar o seguinte comando no console (a partir do diretório `djangogirls` onde está o arquivo `manage.py`):

Mac OS X and Linux:

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

Windows:

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

Você vai notar que um novo diretório `blog` foi criado e que ele contém vários arquivos. Algo como a lista abaixo:

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

Depois de criar uma aplicação, também precisamos dizer ao Django que ele deve usá-la. Fazemos isso no arquivo `mysite/settings.py`. Precisamos encontrar o `INSTALLED_APPS` e adicionar uma linha com `'blog'`, logo acima do `]`. O resultado final ficará assim:

```
mysite/settings.py
```

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

Criando um modelo de postagem para o 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`, remover tudo dele e escrever o código assim:

`blog/models.py`

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

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

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

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

Certifique-se de que usou dois caracteres de sublinhado (`_`) em cada lado de `str`. Esta convenção é utilizada frequentemente em Python e, muitas vezes, a chamamos de "dunder" (vem da expressão em inglês "double-underscore" - sublinhado duplo em português).

Parece assustador, né? Mas não se preocupe, vamos explicar o que essas 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. Nós podemos dar um nome diferente (mas precisamos evitar caracteres especiais e espaços em branco). Sempre inicie o nome de uma classe com uma letra em maiúsculo.
- `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 definimos as propriedades que comentamos acima: `title`, `text`, `created_date`, `published_date` e `author`. Para fazer isso, é necessário definir um tipo para cada campo (É um texto? Um número? Uma data? Uma relação com outro objeto, por exemplo, um usuário?)

- `models.CharField` - é assim que definimos um texto com um número limitado de caracteres.
- `models.TextField` - este campo é para textos sem um limite fixo. Parece ideal para o conteúdo de um blog, né?
- `models.DateTimeField` - este é uma data e hora.

- `models.ForeignKey` - este é um link para outro modelo.

Nós não explicaremos cada pedaço de código aqui pois isso levaria muito tempo. Você deve dar uma olhada na documentação do Django se quiser saber mais sobre campos de modelos e como definir outras coisas além das descritas acima (<https://docs.djangoproject.com/en/2.0/ref/models/fields/#field-types>).

E `def publish(self):` ? Esse é justamente o método `publish` de que falamos anteriormente. `def` significa que se trata de uma função/método e que `publish` é seu nome. Você pode mudar o nome do método, se quiser. A regra para nomes é sempre usar letras minúsculas e no lugar dos espaços em branco, usar o caractere sublinhado (`_`). Por exemplo, um método que calcula o preço médio poderia se chamar `calculate_average_price` (do inglês, `calcular_preco_medio`).

Métodos muitas vezes retornam (`return`) algo. Um exemplo disto é o método `__str__`. Neste caso, quando chamarmos `__str__()`, obteremos um texto (`string`) com o título do Post.

Lembre-se também de que tanto `def publish(self):` quanto `def __str__(self):` são endentados para dentro da classe. E porque Python é sensível a espaços em branco, precisamos endentar todos os nossos métodos para dentro da classe. Caso contrário, os métodos não pertencerão à classe e você poderá obter um comportamento inesperado.

Se algo ainda não está claro sobre modelos, sinta-se livre para perguntar para sua monitora! Sabemos que é complicado, especialmente porque você está aprendendo o que são objetos e funções ao mesmo tempo. Mas esperamos que isto se pareça um pouco menos com mágica agora!

Criando tabelas para nossos modelos no banco de dados

O último passo é adicionar nosso novo modelo ao banco de dados. Primeiramente, precisamos fazer com que o Django entenda que fizemos algumas alterações nos nossos modelos. (Nós acabamos de criar um modelo de Post!) Vá para o console e digite `python manage.py makemigrations blog`. Deve ficar assim:

command-line

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

Observação: Lembre-se de salvar os arquivos que você editar. Caso contrário, o computador executará uma versão antiga que pode gerar mensagens de erro inesperadas.

O Django preparou um arquivo de migração que precisamos aplicar ao nosso banco de dados. Digite `python manage.py migrate blog` e a saída deve ser:

command-line

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

Uhuu! Nosso modelo Post já está no banco de dados! Seria legal vê-lo, né? Vá para o próximo capítulo para descobrir como nosso Post se parece!

Django Admin

Para adicionar, editar e deletar os posts que acabamos de modelar, nós usaremos o admin do Django.

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

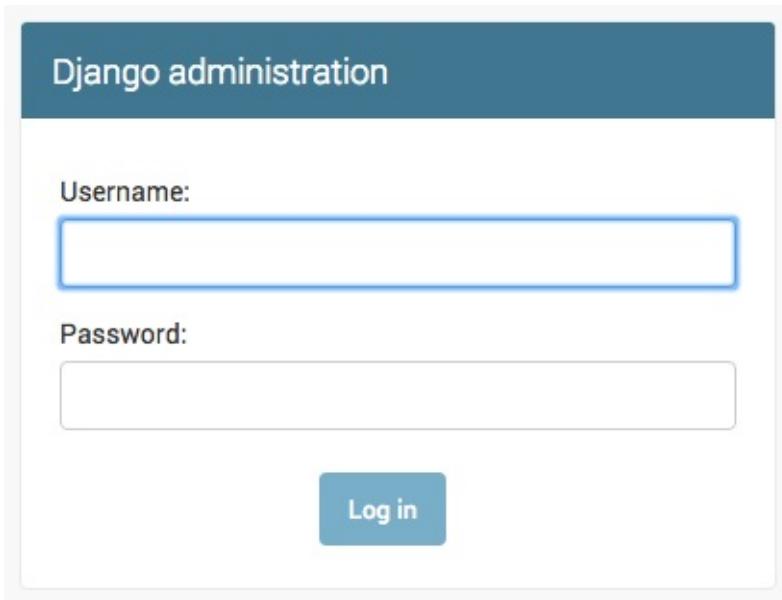
`blog/admin.py`

```
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, 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 iniciar o servidor web. Vá para o seu navegador e digite o endereço <http://127.0.0.1:8000/admin/>. Você verá uma página de login como essa:



Para fazer login, você precisa criar um *superusuário (superuser)* - uma conta de usuário que pode controlar tudo no site. Volte à linha de comando, digite `python manage.py createsuperuser` e aperte Enter.

Lembre-se: para escrever novos comandos enquanto o servidor web estiver rodando, abra uma nova janela do terminal e ative seu virtualenv. Nós revisamos como escrever novos comandos no capítulo **Seu primeiro projeto Django!**, na seção **Iniciando o servidor web**.

Mac OS X ou Linux:

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

Windows:

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

Quando for solicitado, insira seu nome de usuário (letras minúsculas, sem espaços), e-mail e senha. **Não se preocupe se você não conseguir ver a senha que está digitando – é assim que tem ser.** Basta digitá-la e clicar `enter` para continuar. A saída deve parecer com isso (onde o nome de usuário e o email devem ser os seus):

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

Volte ao seu navegador. Faça login com as credenciais de superusuário que você escolheu; você deverá ver o painel de controle de administração do Django.

Django administration

Site administration

AUTHENTICATION AND AUTHORIZATION

Groups

[+ Add](#) [Change](#)

Users

[+ Add](#) [Change](#)

BLOG

Posts

[+ Add](#) [Change](#)

Vá para Posts e brinque um pouco por lá. Adicione cinco ou seis posts. Não se preocupe com o conteúdo - você pode simplesmente copiar e colar algum texto desse tutorial para economizar tempo. :)

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

Django administration

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

Home > Blog > Posts > Add post

Add post

Author: [▼](#) [✎](#) [+](#)

Title:

Text:

Created date: Date: Today [▼](#) Time: Now [▼](#)

Published date: Date: Today [▼](#) Time: Now [▼](#)

[Save and add another](#) [Save and continue editing](#) [SAVE](#)

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

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

Esse provavelmente é um bom momento para pegar um café (ou chá) ou algo para comer para recuperar as energias.

Você criou seu primeiro modelo em Django e merece uma pausa!

Deploy!

Observação: Pode ser um pouco difícil chegar ao final desse capítulo. Persista e termine-o; o deploy é uma parte importante do processo de desenvolvimento de um website. Colocar o seu site no ar é um pouco mais complicado, então esse capítulo está no meio do tutorial para que sua monitora possa lhe ajudar nessa tarefa. Isto significa que você pode terminar o tutorial por conta própria se o tempo acabar.

Até agora, seu site só estava disponível no seu computador. Agora você aprenderá como implantá-lo (fazer o 'deploy')! O deploy é o processo de publicação da sua aplicação na Internet de forma que as pessoas possam, finalmente, vê-la. :)

Como você aprendeu, um website precisa estar em um servidor. Existem vários provedores de servidores na internet, nós vamos usar o [PythonAnywhere](#). O PythonAnywhere é gratuito para pequenas aplicações que não recebem muitos visitantes, então vai ser suficiente para você por enquanto.

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

Estes três lugares serão importantes para você. Seu computador local é o lugar onde você fará o desenvolvimento e testes. Quando estiver satisfeita com as mudanças, você colocará uma cópia de seu programa no GitHub. Seu site estará na PythonAnywhere e você irá atualizá-lo ao subir uma nova cópia do seu código para o GitHub.

Git

Observação: Se você já fez os passos de instalação, não precisa fazer novamente - você pode pular para a próxima seção e comece a criar seu repositório Git.

O Git é um "sistema de controle de versão" usado por muitos programadores. Este software pode acompanhar mudanças em arquivos ao longo do tempo para que você possa recuperar versões específicas mais tarde. Um pouco parecido com o recurso "controlar alterações" do Microsoft Word, mas muito mais poderoso.

Instalando o Git

Installing Git: Windows

Você pode baixar o Git em [git-scm.com](#). Clique em "next" em todos os passos, exceto em um: no passo intitulado "Ajustando o PATH do seu ambiente" (em inglês "Adjusting your PATH environment"), escolha "Use Git e ferramentas Unix opcionais do Prompt de Comando do Windows" (em inglês "Use Git and optional Unix tools from the Windows Command Prompt"), que é a opção mais abaixo. Fora isso, as configurações padrão estão ótimas. "Checkout Windows-style, commit Unix-style line endings" está bom.

Não se esqueça de reiniciar o prompt de comando ou o powershell depois que a instalação terminar com sucesso.

Installing Git: OS X

Baixe o Git em [git-scm.com](#) e siga as instruções.

Observação: Se estiver rodando o OS X 10.6, 10.7, ou 10.8, você precisará instalar essa versão do git: [Instalado Git para o OS X Snow Leopard](#)

Installing Git: Debian or Ubuntu

command-line

```
$ sudo apt install git
```

Installing Git: Fedora

command-line

```
$ sudo dnf install git
```

Installing Git: openSUSE

command-line

```
$ sudo zypper install git
```

Começando o seu repositório no Git

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

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

command-line

```
$ git init
Initialized empty Git repository in ~/djangogirls/.git/
$ git config --global user.name "Seu Nome"
$ git config --global user.email voce@exemplo.com
```

Só é necessário iniciar o repositório Git uma vez por projeto (e você não vai precisar preencher seu nome de usuário e e-mail nunca mais).

O Git irá controlar as alterações em todos os arquivos e pastas neste diretório, mas existem alguns arquivos que queremos que ele ignore. 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:

`.gitignore`

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

E salve-o como `.gitignore` na pasta "djangogirls".

Observação: O ponto no início do nome do arquivo é importante! Se você estiver tendo alguma dificuldade em criá-lo (Macs, por exemplo, não gostam quando você tenta criar arquivos que começam com um ponto por meio do Finder), use a função "Salvar Como..." no seu editor; não tem como errar.

Observação: Um dos arquivos especificados no seu `.gitignore` é o `db.sqlite3`. Este arquivo é o seu banco de dados local, onde todos os seus posts ficarão guardados. Não queremos que você adicione este arquivo ao repositório porque o seu site no PythonAnywhere vai utilizar um banco de dados diferente. Esse banco poderia ser SQLite, como na sua máquina de desenvolvimento, mas normalmente você utilizará um chamado MySQL, que consegue lidar com bem mais visitantes ao site do que o SQLite. De qualquer forma, ao ignorar o banco de dados SQLite para a cópia do GitHub, todos os posts que você criou até agora vão estar disponíveis somente no seu ambiente local, e você terá que adicioná-los novamente durante a produção. Pense no seu banco de dados local como um bom parque de diversões onde você pode testar coisas diferentes e não ter medo de apagar os posts reais do seu blog.

É uma boa idéia usar um comando `git status` antes de `git add` ou sempre que você não tiver certeza do que mudou. Isso evitará quaisquer surpresas, como os arquivos errados serem adicionados ou "commitados". O comando `git status` mostra informações sobre arquivos que não estão sendo controlados, arquivos que foram modificados ou preparados (staged), o status do branch, e muito mais. A saída do comando deve ser parecida com o seguinte:

linha de comando

```
$ git status
On branch master

Initial commit

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

    .gitignore
    blog/
    manage.py
    mysite/
    requirements.txt

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

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

linha de comando

```
$ 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
```

Subindo o seu código para o GitHub

Vá para [GitHub.com](#) e crie uma conta nova, gratuita. (Se você já fez isso em preparação para o workshop, ótimo!)

Em seguida, crie um novo repositório chamado "my-first-blog". Deixe a caixa "Initialize with a README" desmarcada, deixe a opção do `.gitignore` em branco (nós já fizemos isso manualmente) e deixe 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

Observação: O nome `my-first-blog` é importante - você poderia escolher qualquer outra coisa, mas ele vai aparecer várias vezes nas instruções abaixo, e você teria que substituir todas as vezes. É mais fácil simplesmente manter o nome `my-first-blog`.

Na próxima tela, você verá a URL pra clonar o seu repo. Escolha a versão "HTTPS", e copie-a e cole no terminal em seguida:

https://github.com/hjwp/my-first-blog

hjwp / my-first-blog

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

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
```

Code Issues Pull requests Wiki Pulse Graphs Settings

Agora precisamos conectar o repositório Git no seu computador com o que existe no GitHub.

Digite o seguinte no seu terminal (Substitua `<your-github-username>` pelo nome de usuário que você colocou quando criou a sua conta no GitHub, sem os sinais de menor e maior):

linha de comando

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

Digite o seu nome e senha do GitHub e você deverá ver algo parecido com isso:

linha de comando

```
Username for 'https://github.com': ola
Password for 'https://ola@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/ola/my-first-blog.git

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

O seu código agora está no GitHub. Vai lá ver! Você perceberá que ele esta em ótima companhia - o [Django](#), o [Tutorial Django Girls](#) e vários outros incríveis projetos open source também hospedam seu código no GitHub. :)

Configurando o seu blog no PythonAnywhere

Crie uma conta no PythonAnywhere

Observação: Você talvez já tenha criado uma conta no PythonAnywhere durante os passos de instalação. Nesse caso, não precisa criar outra.

Cadastre uma conta gratuita de "Beginner" (Iniciante) no PythonAnywhere:

- www.pythonanywhere.com

Observação: Quando escolher seu nome de usuário, lembre-se de que a URL do seu blog vai ter o formato `nomedeusuario.pythonanywhere.com`, então escolha seu próprio apelido ou um nome que represente o tema do seu blog.

Criando um token de API do PythonAnywhere

Você só precisa fazer isso uma vez. Quando estiver inscrita no PythonAnywhere, você será levada ao seu dashboard. Encontre o link no lado direito superior da página "Accounts", e então selecione a aba "API token" e clique no botão que diz "Create new API token".



Your API token

You do not have an API token yet.

[Create a new API token](#)

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

Configurando o seu site no PythonAnywhere

Navegue para a [Dashboard do PythonAnywhere](#) clicando no logo e escolha a opção de iniciar um console "Bash" -- essa é a versão do PythonAnywhere da linha de comando, igual à que existe no seu computador.

Recent

Consoles

+	5	-
---	---	---

You have no recent consoles.

[View all](#)

New console:

\$ Bash

>>> Python ▾

[More...](#)

Observação: O PythonAnywhere é baseado em Linux, então se você estiver no Windows, o console terá uma cara um pouco diferente do que aparece no seu computador.

Fazer o deploy de uma aplicação web no PythonAnywhere envolve baixar o seu código do GitHub e configurar o PythonAnywhere para reconhecê-lo e começar a servi-lo como uma aplicacão web. Existem formas manuais de fazer isso, mas o PythonAnywhere fornece uma ferramenta que vai fazer tudo pra você. Vamos instalar ela primeiro:

linha de comando do PythonAnywhere

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

Isso deve mostrar na tela coisas como `Collecting pythonanywhere`, e depois de algum tempo finalizar com uma linha dizendo `Successfully installed (...) pythonanywhere (...)`.

Agora vamos executar a ferramenta para configurar a nossa aplicação a partir do GitHub automaticamente. Digite os seguintes comandos no console do PythonAnywhere (não se esqueça de usar o seu nome de usuário no GitHub ao invés de `<your-github-username>`):

linha de comando do PythonAnywhere

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

Enquanto assiste a execução da ferramenta, você pode ver o que ela está fazendo:

- Baixando o seu código do GitHub;
- Criando um virtualenv no PythonAnywhere, igual ao que existe no seu computador;
- Atualizando o seu arquivo de configuração com algumas configurações sobre o deploy;
- Criando um banco de dados no PythonAnywhere usando o comando `manage.py migrate` ;
- Criando os seus arquivos estáticos (nós aprenderemos sobre eles mais tarde)
- E configurando o PythonAnywhere para servir a sua web app através da sua API.

No PythonAnywhere, todos esses passos são automatizados, mas são os mesmos que você executaria ao utilizar qualquer outro provedor. O importante agora é reparar que o seu banco de dados no PythonAnywhere é na verdade completamente separado do banco de dados no seu computador — isso significa que eles têm posts e contas de admin completamente diferentes.

Por causa disso, da mesma forma que tivemos que fazer no nosso computador, precisamos criar a conta de admin com `createsuperuser`. O PythonAnywhere já ativou o seu virtualenv automaticamente, então tudo o que você precisa fazer é executar o comando:

linha de comando do PythonAnywhere

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

Digite as informações sobre a sua conta de admin. É mais fácil usar as mesmas que usou no seu computador pra evitar qualquer confusão, a menos que você queira criar uma senha mais segura para a conta no PythonAnywhere.

Agora, se quiser, você pode dar uma olhada no seu código no PythonAnywhere usando `ls`:

linha de comando do PythonAnywhere

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

Você também pode visitar a aba "Files" e dar uma olhada usando o gerenciador de arquivos do PythonAnywhere.

Estamos no ar!

Agora o seu site deve estar no ar, na internet! Clique na aba "Web" do PythonAnywhere para pegar o link dele. Você pode compartilhar esse link com quem quiser :)

Observação: Este é um tutorial para iniciantes e ao fazer o deploy do site desta forma, nós tomamos alguns atalhos que não são ideais do ponto de vista de segurança. Se e quando você decidir continuar trabalhando nesse projeto ou começar um novo, você deve revisar a [checklist de implantação do Django](#) para pegar algumas dicas de como tornar seu site seguro.

Dicas de debugging

Se você vir um erro ao rodar o script `pa_autoconfigure_django.py`, aqui vão algumas causas comuns:

- Esquecer de criar um token de API do PythonAnywhere.
- Digitar a URL do seu GitHub incorretamente.
- Se você vir um erro dizendo "*Could not find your settings.py*" (`settings.py` não encontrado), você provavelmente não adicionou todos os seus arquivos ao Git e/ou você não fez o push deles para o GitHub. Dê uma revisada na sessão sobre Git acima

Se você vir um erro ao visitar o seu site, o primeiro lugar para procurar informações sobre ele é o **log de erros**. Você vai encontrar um link para o log na aba [Web](#) do PythonAnywhere. Verifique se há alguma mensagem de erro no log; as mais recentes estarão no final.

Há também algumas [dicas de debugging no site de ajuda da PythonAnywhere](#).

E lembre-se, a sua monitora está aí pra ajudar!

Visite o seu site!

A página padrão no seu site deverá dizer "It worked!" ("Funcionou!"), igual ao seu ambiente local. Adicione `/admin` ao final da URL e você será levada ao site de administração. Faça login com o nome de usuário e a senha e você poderá adicionar novos Posts ao servidor.

Depois de criar alguns posts, você pode voltar para o seu ambiente local (não o PythonAnywhere). Daqui pra frente você deve trabalhar no seu ambiente local para fazer alterações. Este workflow é comum no desenvolvimento web – fazer alterações locais, subir essas alterações pro GitHub, e baixar essas alterações para o seu servidor web de produção. Isto permite que você desenvolva e experimente sem quebrar o seu site que está no ar. Bem legal, né?

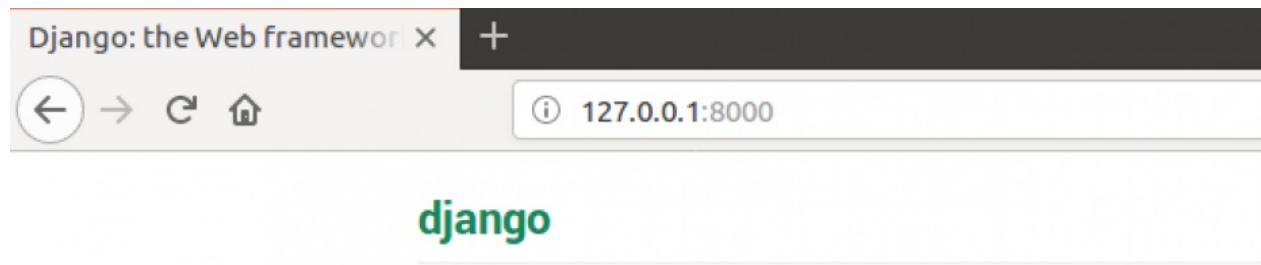
Você merece *MUITOS* parabéns! Deployes em servidores são uma das partes mais complicadas do desenvolvimento web e não é incomum levar vários dias até conseguir fazer com que isso funcione. Mas seu site está no ar, na internet de verdade! Simples assim!

URLs

Estamos prestes a construir nossa primeira página Web: uma página inicial para o seu blog! Mas primeiro, vamos aprender um pouco mais sobre as URLs do Django.

O que é uma URL?

Uma URL é simplesmente um endereço da web. Você pode ver uma URL toda vez que você visita um website - ela aparece na barra de endereços do seu navegador. (Sim! `127.0.0.1:8000` é uma URL! E `https://djangogirls.org` também é uma URL.)



Cada página na Internet precisa de sua própria URL. Desta forma, sua aplicação sabe o que deve mostrar a um usuário que abre uma URL. Em Django, usamos algo chamado `URLconf` (configuração de URLs). URLconf é um conjunto de padrões que o Django vai usar para comparar com a URL recebida para encontrar a resposta correta.

Como funcionam as URLs em Django?

Abra o arquivo `mysite/urls.py` no seu editor de código preferido e veja o que aparece:

`mysite/urls.py`

```
"""mysite URL Configuration

[...]
from django.urls import path, include
from django.contrib import admin

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

Como você pode ver, o Django já colocou alguma coisa aqui para nós.

Linhas entre aspas triplas (`'''` ou `"""`) são chamadas de docstrings -- você pode escrevê-las no topo de um arquivo, classe ou método para descrever o que ele faz. Elas não serão executadas pelo Python.

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

`mysite/urls.py`

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

Isso significa que para cada URL que começa com `admin/`, o Django irá encontrar uma `view` correspondente. Neste caso nós estamos incluindo várias URLs de admin de uma vez a partir de uma lista criada pelo próprio Django em `<0>admin.site.urls`. Desta forma, não temos que repetir todas URLs no nosso modesto arquivo -- é mais legível e mais limpo.

Sua primeira URL no Django!

É hora de criar nossa primeira URL! Queremos que <http://127.0.0.1:8000/> seja a página inicial do nosso blog e exiba uma lista de posts.

Também queremos manter o arquivo de `mysite/urls.py` limpo, e portanto importaremos as URLs da nossa aplicação `blog` no arquivo principal `mysite/urls.py`.

Adicione uma linha para importar `blog.urls`. Note que estamos usando a função `include`, então você também precisará importar esta função.

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

`mysite/urls.py`

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

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

blog.urls

Crie um novo arquivo vazio chamado `urls.py` no diretório `blog`. É fácil! Basta adicionar essas duas linhas:

`blog/urls.py`

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

Aqui, estamos importando do Django a função `url` e todas as nossas `views` do aplicativo `blog`. (Não temos nenhuma ainda, mas chegaremos a isso em um minuto!)

Depois disso podemos adicionar nosso primeiro padrão de URLs:

`blog/urls.py`

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

Como você pode ver, estamos agora atribuindo uma `view` chamada `post_list` à URL raiz. Este padrão de URL corresponde a uma sequência de caracteres vazia, e o resolvidor de URLs do Django irá ignorar o nome de domínio (ou seja, <http://127.0.0.1:8000/>) que antecede o caminho completo da URL. Este padrão dirá ao Django que `views.post_list` é o lugar correto aonde ir se alguém entra em seu site pelo endereço '<http://127.0.0.1:8000/>'.

A última parte, `name='post_list'`, é o nome da URL que será usado para identificar a view. Pode ser o mesmo nome da view, mas também pode ser algo completamente diferente. Nós vamos usar URLs nomeadas mais à frente, então é importante nomearmos agora todas as URLs de nossa aplicação. Também devemos fazer com que os nomes das URLs sejam únicos e fáceis de lembrar.

Se você tentar visitar <http://127.0.0.1:8000/> agora, vai encontrar alguma mensagem do tipo 'página web não disponível'. Isso ocorre porque o servidor (lembre-se de digitar `runserver`?) não está mais funcionando. Dê uma olhada na sua janela de console do servidor para descobrir o porquê.

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

Seu console está mostrando um erro, mas não se preocupe -- este erro é bastante útil: ele está dizendo que não existe nenhum atributo `post_list` no módulo de `views`. Esse é o nome da `view` que Django está tentando encontrar e usar, mas ainda não a criamos. Por enquanto, seu `/admin/` também não funcionará. Mas não se preocupe, nós chegaremos lá.

Se você quer saber mais sobre a configuração de URLs no Django, veja a documentação oficial:

<https://docs.djangoproject.com/en/2.0/topics/http/urls/>

Django views -- hora de criar!

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

Uma `view` é o lugar onde nós colocamos a "lógica" da nossa aplicação. Ela vai extrair informações do `model` que você criou e entregá-las a um `template`. Nós vamos criar um template no próximo capítulo. Views são apenas funções Python um pouco mais complicadas do que aquelas que criamos no capítulo **Introdução ao Python**.

As views são colocadas no arquivo `views.py`. Nós vamos adicionar nossas `views` ao arquivo `blog/views.py`.

blog/views.py

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

blog/views.py

```
from django.shortcuts import render

# Create your views here.
```

Ainda não tem muita coisa aqui.

Lembre-se de que as linhas começando com `#` são comentários -- isto significa que elas não serão executadas pelo Python.

Vamos criar uma `view` (como o comentário em inglês acima sugere). Vamos criar uma view simples agora:

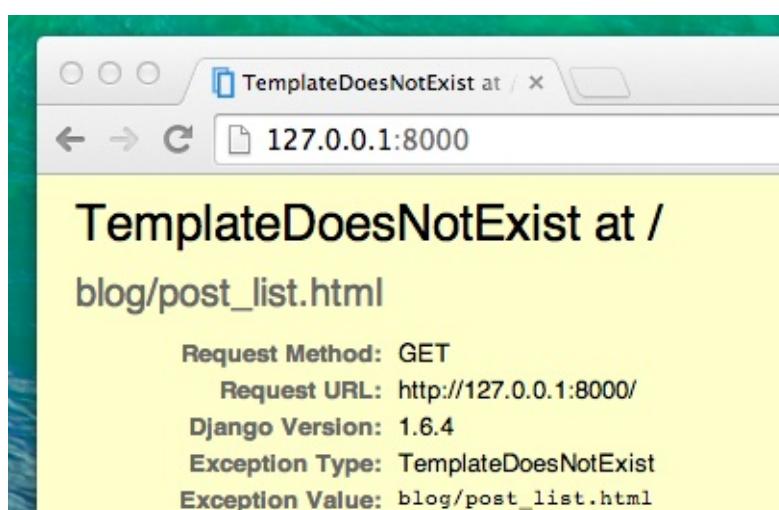
blog/views.py

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

Como você pode ver, nós criamos uma função (`def`) chamada `post_list` que recebe um `request`, executa a função `render` que irá renderizar (montar) nosso modelo de acordo com o template `blog/post_list.html`, e retorna (`return`) o resultado.

Salve o arquivo e abra a página <http://127.0.0.1:8000/> para ver o que acontece.

Outro erro! Veja o que está acontecendo agora:



O erro mostra que o servidor pelo menos está sendo executado, mas ainda não parece certo, né? Não se preocupe, é apenas uma página de erro: nada a temer! Como as mensagens de erro no console, estas também são muito úteis. Você pode ler *TemplateDoesNotExist*, que significa "template não existe", em inglês. 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/2.0/topics/http/views/>

Introdução ao HTML

Você pode se perguntar: o que é um template?

Template é um arquivo que nós podemos reutilizar para apresentar diferentes informações em um formato consistente – por exemplo, você pode usar um template para te ajudar a escrever uma carta, pois mesmo que cada carta contenha mensagens diferentes e possa estar endereçada a pessoas diferentes, elas compartilharão o mesmo formato.

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

O que é HTML?

HTML é um código interpretado pelo seu navegador - como Chrome, Firefox ou Safari - para exibir uma página web ao 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. Códigos em HTML são feitos com **tags**, cada uma começando com `<` e terminando com `>`. Estas tags representam **elementos de marcação**.

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`. Então, crie um diretório chamado `templates` dentro do diretório do seu blog. Em seguida, crie outro diretório chamado `blog` dentro do diretório `templates`:

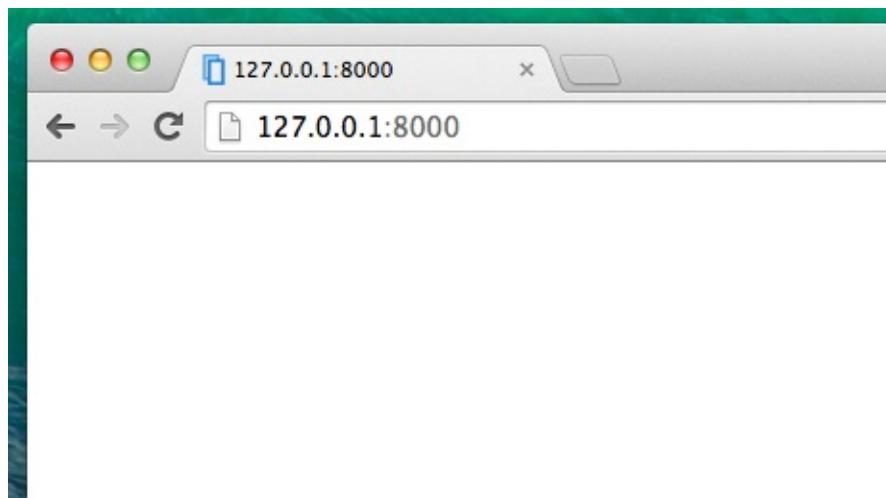
```
blog
└── templates
    └── blog
```

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

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

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

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



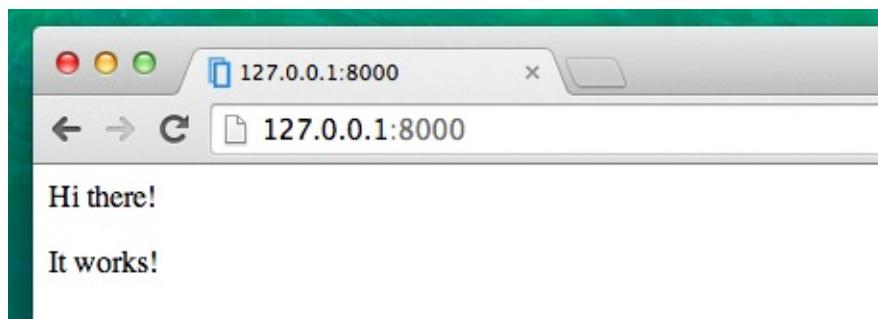
Acabaram-se os erros! Uhuu :) Entretanto, nosso site não mostra nada a não ser uma página em branco. Isso acontece porque o nosso template está vazio. Precisamos consertar isso.

Adicione a seguinte linha ao template:

blog/templates/blog/post_list.html

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

Como está o nosso site agora? Visite a página 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. Todo o conteúdo de um website se encontra entre a tag de início `<html>` e a tag de fim `</html>`
- `<p>` é a tag que inicia um parágrafos; `</p>` determina o fim de um parágrafo

"Head" e "body"

Cada página HTML também é dividida em dois elementos: **head** (cabecalho) 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 da web.

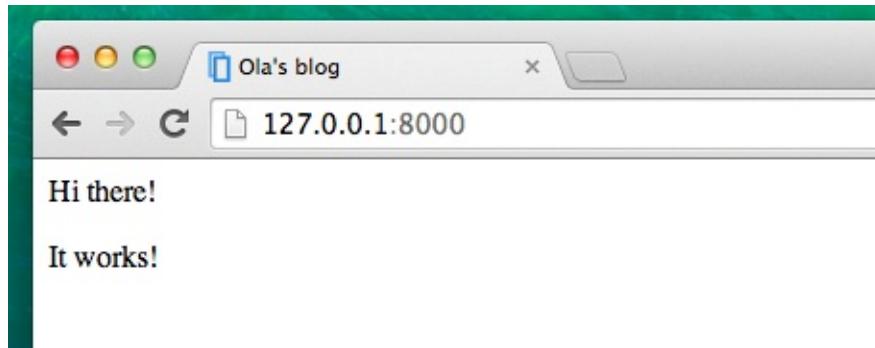
Nós usamos a tag `<head>` para dizer ao navegador quais são as configurações da página. Por sua vez, a tag `<body>` diz ao navegador qual é o conteúdo de fato da página.

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

blog/templates/blog/post_list.html

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

Salve o arquivo e atualize a 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 (o texto também será usado nos 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 uma tag específica antes que todas as outras tags dentro dela estejam fechadas).

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

Você precisa seguir essas regras de *fechamento* de tags, e de *aninhamento* de elementos - se não fizer isso, o navegador provavelmente não interpretará seu código da maneira correta e sua página será exibida incorretamente.

Personalize seu template

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

- `<h1>Um título</h1>` para o título da seção principal exibido na página
- `<h2>Um sub-título</h2>` para um título um nível abaixo
- `<h3>Um sub-sub-título</h3>` ... e por aí vai, até `<h6>`
- `<p>Um parágrafo de texto</p>`
- `texto` enfatiza seu texto
- `text` enfatiza fortemente seu texto
- `
` quebra a linha (você não pode digitar nada dentro da tag `br` e ela não possui uma tag de fechamento correspondente)
- `link` cria um link
- `primeiro itemsegundo item` cria uma lista, exatamente como essa!
- `<div></div>` define uma seção da página

Aqui vai um exemplo de um template completo; copie e cole dentro de `blog/templates/blog/post_list.html`:

`blog/templates/blog/post_list.html`

```

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

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

    <div>
      <p>publicado: 14.06.2014, 12:14</p>
      <h2><a href="">Meu segundo post</a></h2>
      <p>Aenean eu leo quam. Pellentesque ornare sem lacinia quam venenatis vestibulum. Donec id elit non mi porta 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 que contém um link
- Os outros dois elementos `div` possuem nossas postagens com a data de publicação, `h2` com o título clicável da postagem e dois `p`s (parágrafos) de texto, um para a data e outro para o texto da postagem.

Isso nos dá o seguinte efeito:



Uhuu! Mas, até o momento, nosso template mostra **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 as postagens reais que foram adicionadas no Django admin - e isso é o que faremos em seguida.

Mais uma coisa: implantação (deploy)!

Seria bom ver tudo isto na Internet, certo? Vamos fazer mais um deploy no PythonAnywhere:

Commit, e dê push para subir seu código no GitHub

Antes de tudo, vamos ver que arquivos foram modificados desde que nós fizemos a última implantação (execute esses comandos localmente, não no PythonAnywhere):

command-line

```
$ git status
```

Assegure-se de que você está no diretório `djangogirls` e vamos dizer ao `git` para incluir todas as mudanças feitas nele:

command-line

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

Observação: `--all .` significa que o `git` também vai reconhecer se você tiver excluído arquivos (por padrão, ele só reconhece arquivos novos/modificados). Lembre-se também (do capítulo 3) que `.` significa o diretório atual.

Antes de fazermos o upload de todos os arquivos, vamos verificar quais deles o `git` enviará (todos os arquivos marcados para upload pelo `git` aparecerão em verde):

command-line

```
$ git status
```

Estamos quase lá! Agora é hora de dizer ao `git` para salvar essa modificação em seu histórico. Daremos a ele uma "mensagem de commit" em que descrevemos as modificações que foram feitas. Você pode escrever o que quiser agora, mas é mais útil escrever algo descritivo que te ajude no futuro a lembrar das coisas que fez.

command-line

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

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

Uma vez feito isso, faremos o upload (push) das nossas mudanças para o Github:

command-line

```
$ git push
```

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

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

PythonAnywhere command-line

```
$ cd ~/<your-pythonanywhere-username>.pythonanywhere.com  
$ git pull  
[...]
```

(Lembre-se de substituir o `<your-pythonanywhere-username>` pelo seu username do PythonAnywhere, sem os símbolos `<` e `>`).

E veja seu código sendo baixado. Se você quiser verificar se ele já baixou, pode ir para a **aba Files** e ver seu código no PythonAnywhere.

- Finalmente, pule para a aba **Web** e aperte **Reload** em sua aplicação.

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

QuerySets e ORM do Django

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

O que é um QuerySet?

Um QuerySet (conjunto de busca) é, em essência, uma lista de objetos de um dado modelo. QuerySet permite que você leia os dados a partir de uma base de dados, filtre e ordene.

É mais fácil aprender usando exemplos. Vamos lá?

O Shell do Django

Abra o seu terminal (não o PythonAnywhere) e digite o seguinte comando:

command-line

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

O resultado deve ser:

command-line

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

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

Todos os objetos

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

command-line

```
>>> 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 antes!

command-line

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

Mas isso é simples: basta importar o modelo `Post` de dentro do `blog.models`. Vamos tentar mostrar todas as postagens novamente:

command-line

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

É uma lista dos posts que criamos mais cedo! Nós criamos estes posts utilizando a interface do Django admin. No entanto, agora queremos criar novos posts utilizando Python. Como fazemos isso?

Criando um objeto

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

command-line

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

Mas aqui temos um ingrediente faltando: `me`. Precisamos passar uma instância do modelo `User` como autor. Como fazemos isso?

Primeiro vamos importar o modelo User:

command-line

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

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

command-line

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

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

command-line

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

Como podemos ver, agora usamos o método `get` para selecionar um `User` com o campo `username` igual a 'ola'. Claro, você tem que adaptar este comando ao seu nome de usuário.

Agora finalmente podemos criar nosso post:

command-line

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

Uhuu! Quer ver se funcionou?

command-line

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

É isso aí, mais um post na lista!

Adicionando mais postagens

Agora você pode se divertir um pouco e adicionar algumas postagens para ver como funciona. Adicione mais uns 2 ou 3 posts pelo Python e siga para a próxima parte.

Filtrando objetos

Um recurso importante dos QuerySets é a possibilidade de filtrá-los. Digamos que queremos encontrar todos as postagens escritas pela usuária ola. Para isto, usamos `filter` ao invés de `all` em `Post.objects.all()`. Entre parênteses, indicamos quais condições precisam ser atendidas por um post para que ele entre no nosso queryset. No nosso caso, a condição é: `author` é igual a `me`. A maneira de escrever isso no Django é: `author=me`. Agora o nosso trecho de código ficará assim:

command-line

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

E se quisermos ver todos os posts que contenham a palavra 'title' no campo `title` ?

command-line

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

Observação: Existem dois caracteres de sublinhado (`_`) entre `title` e `contains`. O ORM do Django utiliza esta sintaxe para separar nomes de campo ("title") e operações ou filtros (como "contains"). Se você usar apenas um sublinhado, 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 uma `published_date` definida no passado:

command-line

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

Infelizmente, o post que nós criamos pelo console do Python não está publicado ainda. Podemos mudar isso! Primeiro, busque a instância do post que queremos publicar:

command-line

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

Então vamos publicá-lo com o nosso método `publish`:

command-line

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

Agora, busque novamente a lista de posts publicados (aperte a seta para cima algumas vezes e pressione `enter`):

command-line

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

Ordenando objetos

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

`created_date :`

command-line

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

Também podemos inverter a ordem adicionando `-` no início:

command-line

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

Encadeando QuerySets

Você também pode combinar QuerySets **encadeando-os**:

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

Isso é muito poderoso e nos permite criar consultas bastante complexas.

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

command-line

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

Dados dinâmicos em templates

Até o momento, temos diferentes peças: o modelo `Post` está definido em `models.py`, temos `post_list` em `views.py` e o template adicionado. Mas como faremos de fato para que as postagens apareçam no nosso template em HTML? Porque é isso que nós queremos: pegar algum conteúdo (modelos 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 modelos e templates. Vamos precisar pegar os modelos que queremos exibir e passá-los para o template na nossa lista de postagens `post_list view`. Em uma `view`, nós decidimos o que (qual modelo) será exibido em um template.

Tudo bem, e como vamos fazer isso?

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

`blog/views.py`

```
from django.shortcuts import render

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

Lembra de quando falamos sobre a inclusão de código que foi escrito em arquivos diferentes? Agora é o momento em que precisamos que incluir o modelo que temos escrito em `models.py`. Vamos adicionar a linha `from .models import Post` assim:

`blog/views.py`

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

O ponto antes de `models` significa *diretório atual* ou *aplicativo atual*. Tanto `views.py` como `models.py` estão no mesmo diretório. Isto significa que podemos usar `.` e o nome do arquivo (sem `.py`). Em seguida, importamos o nome do modelo (`Post`).

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

QuerySet

Você já deve estar familiarizada com o modo que os QuerySets funcionam. Nós conversamos sobre isso no [capítulo QuerySets e ORM do Django](#).

Agora queremos classificar as postagens publicadas por `published_date`, certo? Nós já fizemos isso no capítulo sobre QuerySets!

`blog/views.py`

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

Agora vamos colocar esse pedaço de código dentro do arquivo `blog.views.py` adicionando-o à função `def post_list(request)`. Não esqueça de adicionar `from django.utils import timezone` antes!

`blog/views.py`

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

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

A última parte que falta é passar a QuerySet `posts` para o template. Não se preocupe com isso agora, vamos falar sobre como exibi-lo em um próximo capítulo.

Note que criamos uma *variável* para nosso o QuerySet: `posts`. Esse é o nome do nosso QuerySet. De agora em diante, podemos nos referir a ele por este nome.

Na função `render` já temos um parâmetro `request` (tudo o que recebemos do usuário através da Internet) e um arquivo de template (`'blog/post_list.html'`). O último parâmetro -- `{}` -- é um lugar em que podemos acrescentar algumas coisas para o template usar. Precisamos nomear os parâmetros (continuaremos com `'posts'`, por enquanto). :) Deve ficar assim: `{'posts': posts}`. Note que a parte antes de `:` é uma string; por isso você precisa colocá-la entre aspas:

```
" .
```

Agora, nosso arquivo `blog/views.py` deve ter essa cara:

`blog/views.py`

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

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

Pronto! 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/2.0/ref/models/querysets/>

Templates do Django

Hora de exibir dados! O Django já vem com algumas **tags de template** que são úteis pra isso.

O que são tags de template?

Sabe, em HTML não podemos incluir código Python porque os browsers só entendem HTML. Sabemos que o HTML é bem estático, enquanto o Python é muito mais dinâmico.

Tags de template Django nos permitem transformar código similar a Python em código HTML para que você possa construir sites dinâmicos mais rápido e mais facilmente. Legal!

Templates para lista de posts

No capítulo anterior, fornecemos para o nosso template uma lista de postagens e a variável `posts`. Agora vamos exibi-las em HTML.

Pra mostrar uma variável em um template do Django, usamos chaves duplas com o nome da variável, assim:

`blog/templates/blog/post_list.html`

```
{{ posts }}
```

Tente fazer isso no seu template `blog/templates/blog/post_list.html`. Substitua tudo desde o segundo `<div>` até o terceiro `</div>` por `{{ posts }}`. Salve o arquivo e atualize a página para ver o resultado:



Como você pode ver, obtivemos apenas:

`blog/templates/blog/post_list.html`

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

Isto significa que o Django entende essa variável como uma lista de objetos. Em **Introdução ao Python** aprendemos como exibir listas, lembra? Sim, com laços "for"! Em um template do Django você pode criá-los assim:

`blog/templates/blog/post_list.html`

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

Tente fazer isso no seu template.



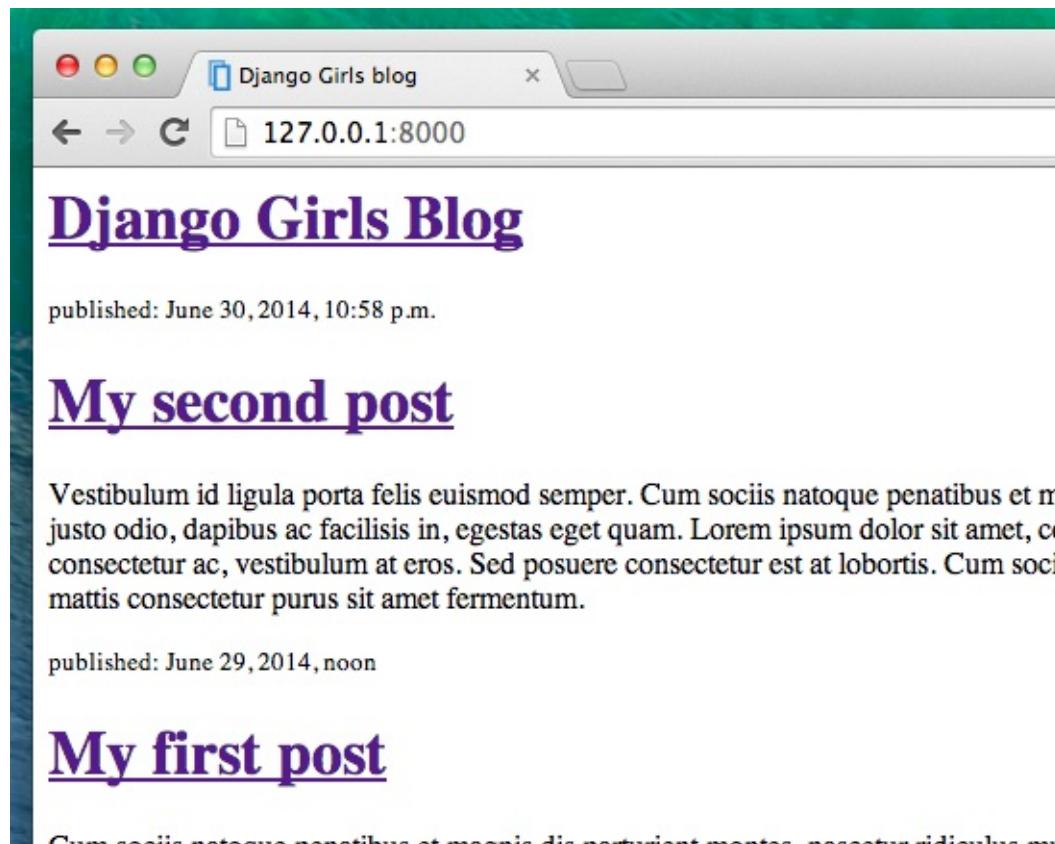
Funciona! Mas nós queremos que eles sejam exibidos como os posts estáticos que criamos anteriormente no capítulo de **Introdução a HTML**. Nós podemos misturar HTML com tags de template. O conteúdo da tag `body` ficará assim:

`blog/templates/blog/post_list.html`

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

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

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



Você notou que dessa vez nós usamos uma notação um pouco diferente (`{{ post.title }}` ou `{{ post.text }}`)? Estamos acessando os dados em cada um dos campos que definimos no modelo do `Post`. Além disso, `|linebreaks` 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 funciona na internet, né? Vamos tentar implantar a PythonAnywhere novamente. Aqui está um resumo dos passos...

- Primeiro, envie seu código para o Github

command-line

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

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

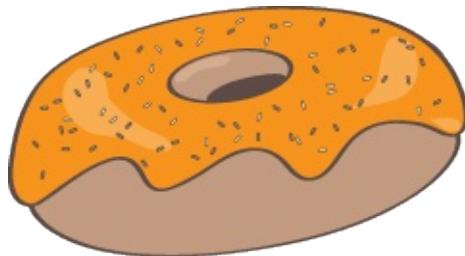
PythonAnywhere command-line

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

- Por fim, vá para a aba [Web app setup](#) e clique em **Reload** em sua aplicação Web. Sua atualização deve ter aparecido! Tudo bem se as postagens em seu site PythonAnywhere não coincidirem com as postagens que aparecem no blog hospedado no seu servidor local. Os bancos de dados em seu computador local e no Python Anywhere não sincronizam com o resto de seus arquivos.

Parabéns! Agora vá em frente e tente adicionar um novo post em seu Django admin (Lembre-se de adicionar `published_date!`). Certifique-se de que você está no Django admin do seu site `pythonanywhere`, <https://yourname.pythonanywhere.com/admin>. Em seguida, atualize a página para ver se o post aparece.

Funcionou como mágica? Estamos orgulhosas! Afaste-se do seu computador um pouco, você merece uma pausa. :)



CSS - deixe tudo mais bonito!

Nosso blog ainda está um pouco feio, né? Está na hora de deixar ele mais bonitinho! Para isso, nós usaremos o CSS.

O que é CSS?

Cascading Style Sheets (CSS - Folhas de Estilo em Cascata, em português) é uma linguagem utilizada para descrever o visual e a formatação de um website escrito numa linguagem de marcação (como HTML). Considere como uma maquiagem para a nossa página web. :)

Mas não queremos começar do zero de novo, né? Mais uma vez, usaremos algo que outros programadores lançaram na Internet de graça. Você sabe, reinventar a roda não é divertido.

Vamos usar o Bootstrap!

Bootstrap é um dos frameworks de HTML e CSS mais populares para desenvolver websites mais bonitinhos:

<https://getbootstrap.com/>

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

Instalar o Bootstrap

Para instalar o Bootstrap, você precisa adicionar o seguinte código no `<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, apenas aponta para arquivos que existem na Internet. Vá em frente, abra seu website e atualize a página. Aqui está!



Já está mais bonita!

Arquivos estáticos no Django

Finalmente, vamos dar uma olhada mais de perto nessas coisas que chamamos de **arquivos estáticos**. Arquivos estáticos são todos os seus CSS e imagens. Seu conteúdo não depende do contexto de requisição e será o mesmo para todos os usuários.

Onde colocar os arquivos estáticos para o Django

O Django já sabe onde encontrar os arquivos estáticos para o app pré-instalado "admin". Agora só precisamos adicionar alguns arquivos estáticos para o nosso próprio app, `blog`.

Fazemos isso criando uma pasta chamada `static` dentro da aplicação `blog`:

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

O Django vai encontrar automaticamente quaisquer pastas chamadas "static" dentro de todas as pastas dos seus apps. Então ele será capaz de usar seu conteúdo como arquivos estáticos.

Seu primeiro arquivo CSS!

Vamos criar um arquivo CSS agora, para adicionar seu próprio estilo à sua página 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`. Pronta?

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

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

Nós não vamos nos aprofundar muito no aprendizado do CSS aqui. No final desta página há uma recomendação para um curso gratuito de CSS se você quiser aprender mais.

Mas vamos fazer pelo menos um pouco. Vamos mudar a cor do nosso cabeçalho? Para entender as cores, os computadores usam códigos especiais. Esses códigos começam com `#` e são seguidos de 6 letras (A-F) e números (0-9). Por exemplo, o código para o azul é `#0000FF`. Você pode encontrar os códigos de diversas cores aqui:

<http://www.colorpicker.com/>. Você pode também usar **cores predefinidas**, como `red` e `green`.

Em seu arquivo `blog/static/css/blog.css`, adicione o seguinte código:

`blog/static/css/blog.css`

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

`h1 a` é um seletor CSS. Isto significa que estamos aplicando nossos estilos a qualquer elemento `a` dentro de um elemento `h1`. Então quando tivermos algo como um `<h1>link</h1>`, o estilo `h1 a` será aplicado. Neste caso, nós estamos dizendo para mudar a cor para `#FCA205`, que é laranja. Mas é claro que você pode colocar a cor que quiser aqui!

Em um arquivo CSS, nós determinamos estilos para elementos do arquivo HTML. A primeira maneira de identificar elementos é usando seus nomes. Você pode se lembrar desses nomes porque são a mesma coisa que as tags da seção HTML. `a`, `h1` e `body` são exemplos de nomes de elementos. Também identificamos elementos pelo atributo `class` ou pelo atributo `id`. `Class` e `id` são nomes que você mesma dá ao elemento. Classes definem grupos de elementos e ids apontam para elementos específicos. Por exemplo, a tag a seguir pode ser identificada usando a tag de nome `a`, a classe `external_link` ou o id de `link_to_wiki_page`:

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

Você pode ler mais sobre [Seletores CSS no w3schools](#).

Nós também precisamos dizer ao nosso template HTML que adicionamos algum CSS. Abra o arquivo `blog/templates/blog/post_list.html` e adicione esta linha bem no começo dele:

`blog/templates/blog/post_list.html`

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

Estamos apenas carregando arquivos estáticos aqui. :) Adicione a seguinte linha entre as tags `<head>` e `</head>` e depois dos links para os arquivos CSS do Bootstrap:

`blog/templates/blog/post_list.html`

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

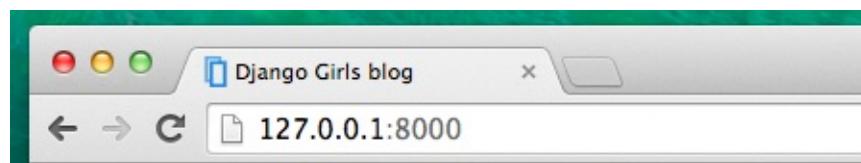
O navegador lê os arquivos na ordem em que são apresentados, então precisamos ter certeza de que eles estão no lugar certo. Do contrário, o código no nosso arquivo poderia ser sobreescrito pelo código dos arquivos do Bootstrap. Acabamos de dizer ao nosso template onde está o nosso arquivo CSS.

Agora, seu arquivo deve estar assim:

`blog/templates/blog/post_list.html`

```
{% 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!



Django Girls Blog

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

My second post

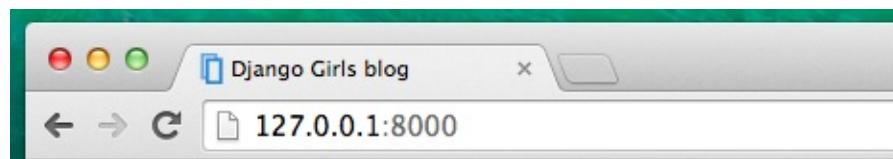
Vestibulum id ligula porta felis euismod semper. Cum sociis natoque

Bom trabalho! Que tal a gente dar um pouco de espaço ao nosso site e aumentar a margem do lado esquerdo? Vamos tentar!

blog/static/css/blog.css

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

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



Django Girls Blog

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

My second post

Vestibulum id ligula porta felis euismod semper. Cum sociis natoque

E que tal customizar a fonte no nosso cabeçalho? Cole o seguinte na seção `<head>` do arquivo

blog/templates/blog/post_list.html :

blog/templates/blog/post_list.html

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

Assim como antes, cheque a ordem e a posição antes do link para `blog/static/css/blog.css`. Esta linha importará do Google Fonts (<https://www.google.com/fonts>) uma fonte chamada *Lobster*.

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

`blog/static/css/blog.css`

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



Incrível!

Como mencionado acima, o CSS tem um conceito de classes. Essas classes permitem que você nomeie uma parte do código HTML e aplique estilos apenas a esta parte sem afetar nenhuma outra. Isto pode ser super útil! Talvez você tenha duas divs que estão fazendo coisas diferentes (como o seu cabeçalho e seu post). Uma classe pode ajudá-la a diferenciá-los.

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

`blog/templates/blog/post_list.html`

```
<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.

`blog/templates/blog/post_list.html`

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

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

`blog/static/css/blog.css`

```
.page-header {  
    background-color: #ff9400;  
    margin-top: 0;  
    padding: 20px 20px 20px 40px;  
}  
  
.page-header h1, .page-header h1 a, .page-header h1 a:visited, .page-header h1 a:active {  
    color: #ffffff;  
    font-size: 36pt;  
    text-decoration: none;  
}  
  
.content {  
    margin-left: 40px;  
}  
  
h1, h2, h3, h4 {  
    font-family: 'Lobster', cursive;  
}  
  
.date {  
    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;  
}
```

Agora inclua declarações de classes no código HTML que exibe os posts. No arquivo blog/templates/blog/post_list.html, substitua isto:

blog/templates/blog/post_list.html

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

por isto:

blog/templates/blog/post_list.html

```

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

```

Salve esses arquivos e atualize seu site.



Uau! Está lindo, né? Olhe para o código que acabamos de colar para encontrar os locais onde adicionamos classes no HTML e as usamos no CSS. Onde você faria uma mudança se você quisesse a data na cor turquesa?

Não tenha medo de mexer um pouco com esse CSS e tentar mudar algumas coisas. Brincar com o CSS pode ajudá-la a entender o que diferentes coisas estão fazendo. Se quebrar algo, não se preocupe - você sempre pode desfazer!

Nós realmente recomendamos fazer este curso gratuito on-line: [Curso de HTML & CSS do Codeacademy](#). Isso pode ajudá-la a aprender tudo sobre como deixar seus sites mais bonito com CSS.

Pronta para o próximo capítulo?! :)

Estendendo os templates

Outra coisa boa que o Django tem para você é o **template extending** - extensão de templates. O que isso significa? Significa que você pode usar as mesmas partes do seu HTML em diferentes páginas do seu site.

Templates ajudam quando você quer usar a mesma informação ou layout em mais de um lugar. Você não precisa se repetir em todos os arquivos. E, caso queira mudar algo, você não precisa fazê-lo em todos os templates, apenas em um!

Criando um template base

Um template base é o template mais básico sobre o qual você construirá 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
```

Em seguida, abra o arquivo criado e copie tudo que ele contém `post_list.html` para `base.html`, desse jeito:

`blog/templates/blog/base.html`

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

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

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

`blog/templates/blog/base.html`

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

Você pode ver que essa ação substituiu tudo a partir de `{% for post in posts %}` até `{% endfor %}` por:

`blog/templates/blog/base.html`

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

Mas por quê? Você acabou de criar um `bloco`! Você usou o a etiqueta de template (template tag) `{% block %}` para criar uma área que terá HTML inserido nela. Esse HTML virá de outro template que vai estender este template (`base.html`). Nós vamos te mostrar como fazer isso já já.

Agora salve `base.html` e abra seu `blog/templates/blog/post_list.html` novamente. Remova tudo acima de `{% for post in posts %}` e abaixo de `{% endfor %}`. Quando terminar, o arquivo deve estar parecido com isso:

`blog/templates/blog/post_list.html`

```
{% 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 %}
```

Queremos usar isso como parte do nosso template para todos os blocos de conteúdo. É hora de adicionar as tags (etiquetas) de blocos neste arquivo!

Você quer que sua etiqueta de bloco coincida com a etiqueta no seu arquivo `base.html`. Você também quer que inclua todo o código que pertence aos seus blocos de conteúdo. Para isso, coloque tudo entre `{% block content %}` e `{% endblock %}`. Assim:

`blog/templates/blog/post_list.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 %}
```

Só falta uma coisa. Precisamos juntar estes dois templates. Isto é o que significa 'estender templates'! Vamos fazer isso adicionando uma etiqueta de extensão ao início do arquivo. Assim:

blog/templates/blog/post_list.html

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

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

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

Se você receber o erro `TemplateDoesNotExist`, significa que não há um arquivo em `blog/base.html` e você está rodando o `runserver` na janela do terminal. Tente encerrá-lo (apertando Ctrl+C -- as teclas Control e C juntas) e reiniciá-lo 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 deixá-lo bonitinho.

Hora de praticar!

A primeira coisa que precisamos em nosso blog é, obviamente, uma página para mostrar uma postagem, né?

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

Criando um link para os detalhes de um post

Vamos começar adicionando um link dentro do arquivo `blog/templates/blog/post_list.html`. Por enquanto, ele deve se parecer com isto:

`blog/templates/blog/post_list.html`

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

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

Queremos um link no título do post dentro da página de lista de posts apontando para a página de detalhes do post respectivo. Vamos mudar `<h1>{{ post.title }}</h1>` e adicionar um link para a página de detalhe:

`blog/templates/blog/post_list.html`

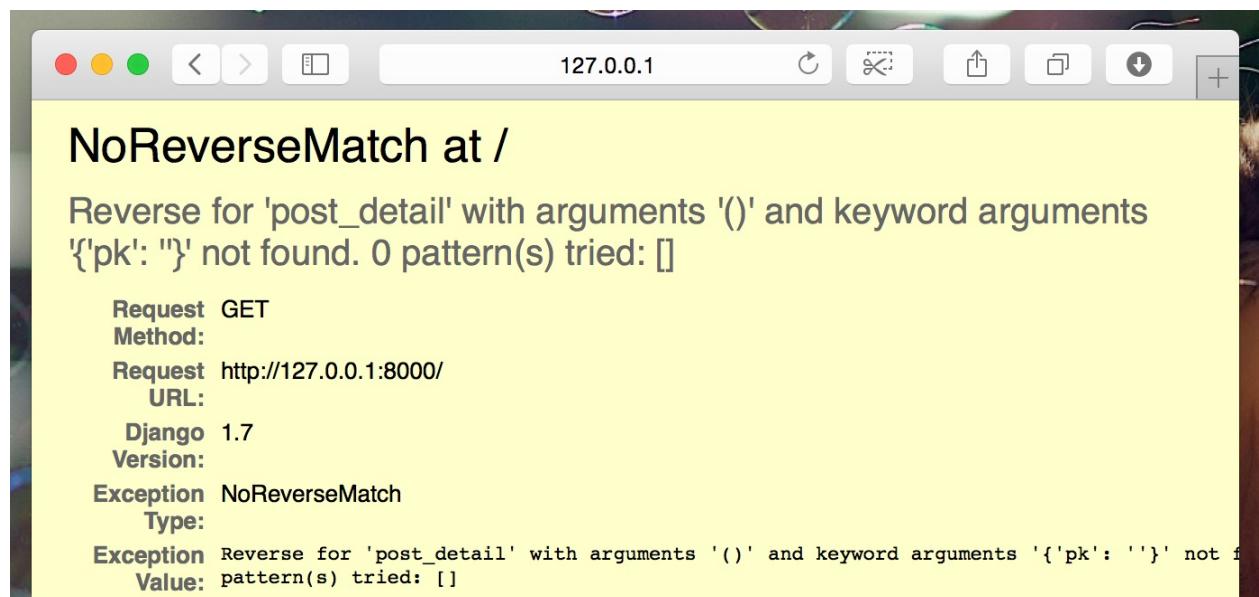
```
<h1><a href="{% url 'post_detail' pk=post.pk %}">{{ post.title }}</a></h1>
```

Hora de explicar o misterioso `{% url 'post_detail' pk=post.pk %}`. Como você deve suspeitar, a notação `{% %}` significa que estamos usando as tags de template do Django. Dessa vez, usamos uma que cria uma URL para nós!

A parte `post_detail` significa que o Django espera uma URL no arquivo `blog/urls.py` com o nome definido como `name='post_detail'`

E quanto ao `pk=post.pk`? `pk` é uma abreviação de "primary key" (do inglês chave primária), que é um identificador único de cada entrada em um banco de dados. Uma vez que não especificamos a chave primária em nosso modelo de `Post`, o Django cria uma para nós (que por padrão, é um número que incrementa sequencialmente a partir de 1, 2, 3, etc) e a adiciona como um campo chamado `pk` em cada um dos nossos posts. Acessamos a chave primária escrevendo `post.pk`, do mesmo modo que podemos acessar outros campos (`title`, `author`, etc.) no nosso objeto de `Post`!

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



Criando uma URL para a página de detalhes de um post

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

Queremos que a página de detalhes do nosso primeiro post seja exibida por essa URL: <http://127.0.0.1:8000/post/1/>

Vamos criar uma URL no arquivo `blog/urls.py` que aponta para uma view chamada `post_detail`, que vai nos mostrar o post completo. Adicione a linha `url(r'^post/(?P<int:pk>+)$', views.post_detail, name='post_detail')` ao arquivo `blog/urls.py`. O arquivo deverá ficar assim:

`blog/urls.py`

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

urlpatterns = [
    path('', views.post_list, name='post_list'),
    path('post/<int:pk>/', views.post_detail, name='post_detail'),
]
```

A parte `post/<int:pk>/` especifica um padrão de URL – vamos explicar:

- `post/` significa apenas que a URL deve começar com a palavra `post` seguida por `/`. Até aqui, tudo bem.
- `<int:pk>` – essa parte é um pouco mais complicada. Ela nos diz que o Django espera um objeto do tipo inteiro e que vai transferi-lo para a view como uma variável chamada `pk`.
- `/` – por fim, precisamos adicionar uma `/` ao final da nossa URL.

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

Legal, adicionamos um novo padrão de URL para `blog/urls.py`! Vamos atualizar a página: <http://127.0.0.1:8000/> Boom! O servidor parou de funcionar de novo. Dê um olhada no console -- como esperado, há ainda outro erro!

```

    return _bootstrap._gcd_import(name[level:], package, level)
File "<frozen importlib._bootstrap>", line 2231, in _gcd_import
File "<frozen importlib._bootstrap>", line 2214, in _find_and_load
File "<frozen importlib._bootstrap>", line 2203, in _find_and_load_unlocked
File "<frozen importlib._bootstrap>", line 1200, in _load_unlocked
File "<frozen importlib._bootstrap>", line 1129, in _exec
File "<frozen importlib._bootstrap>", line 1448, in _exec_module
File "<frozen importlib._bootstrap>", line 321, in _call_with_frames_removed
File "/home/hel/code/djangogirls/workthrough/blog/urls.py", line 6, in <module>
    url(r'^post/(?P<pk>[0-9]+)/$', views.post_detail, name='post_detail'),
AttributeError: 'module' object has no attribute 'post_detail'

```

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

Adicionando a view de detalhes do post

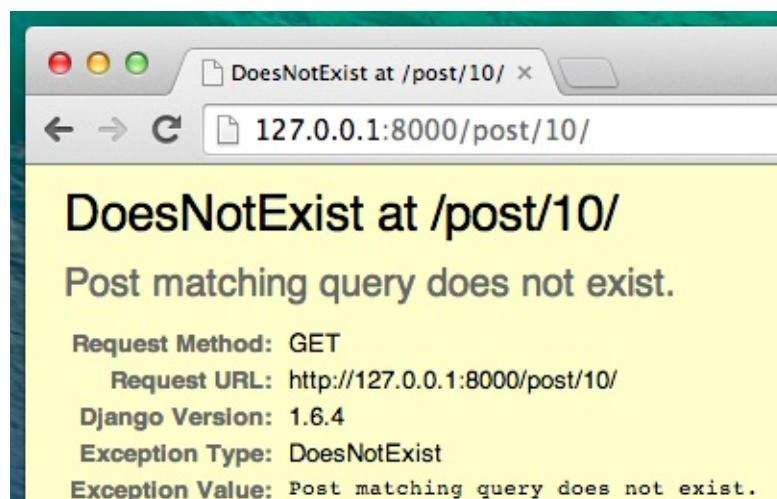
Desta vez, a nossa `view` recebe um parâmetro extra: `pk`. Nossa `view` precisa pegá-lo, certo? Então vamos definir nossa função como `def post_detail (request, pk):`. Precisamos usar exatamente o mesmo nome que especificamos em `urls` (`pk`). Omitir essa variável é incorreto e resultará em um erro!

Agora, queremos receber apenas um post do blog. Para isso, podemos usar queries (buscas) como esta:

`blog/views.py`

```
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 horroroso!



Não queremos isso! Mas é claro que o Django tem algo para lidar com isso por nós: `get_object_or_404`. Caso não haja nenhum `Post` com o `pk`, o Django exibirá uma página muito mais agradável que aquela mensagem de erro -- `Page Not Found 404` (página não encontrada).



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

Hora de adicionar uma view ao nosso arquivo `views.py` !

Em `blog/urls.py`, criamos uma regra de URL chamada `post_detail` que se refere a uma view chamada `views.post_detail`. Isto significa que o Django espera uma função chamada `post_detail` dentro de `blog/views.py`.

Vamos abrir `blog/views.py` e adicionar o seguinte código perto das outras linhas `from`:

`blog/views.py`

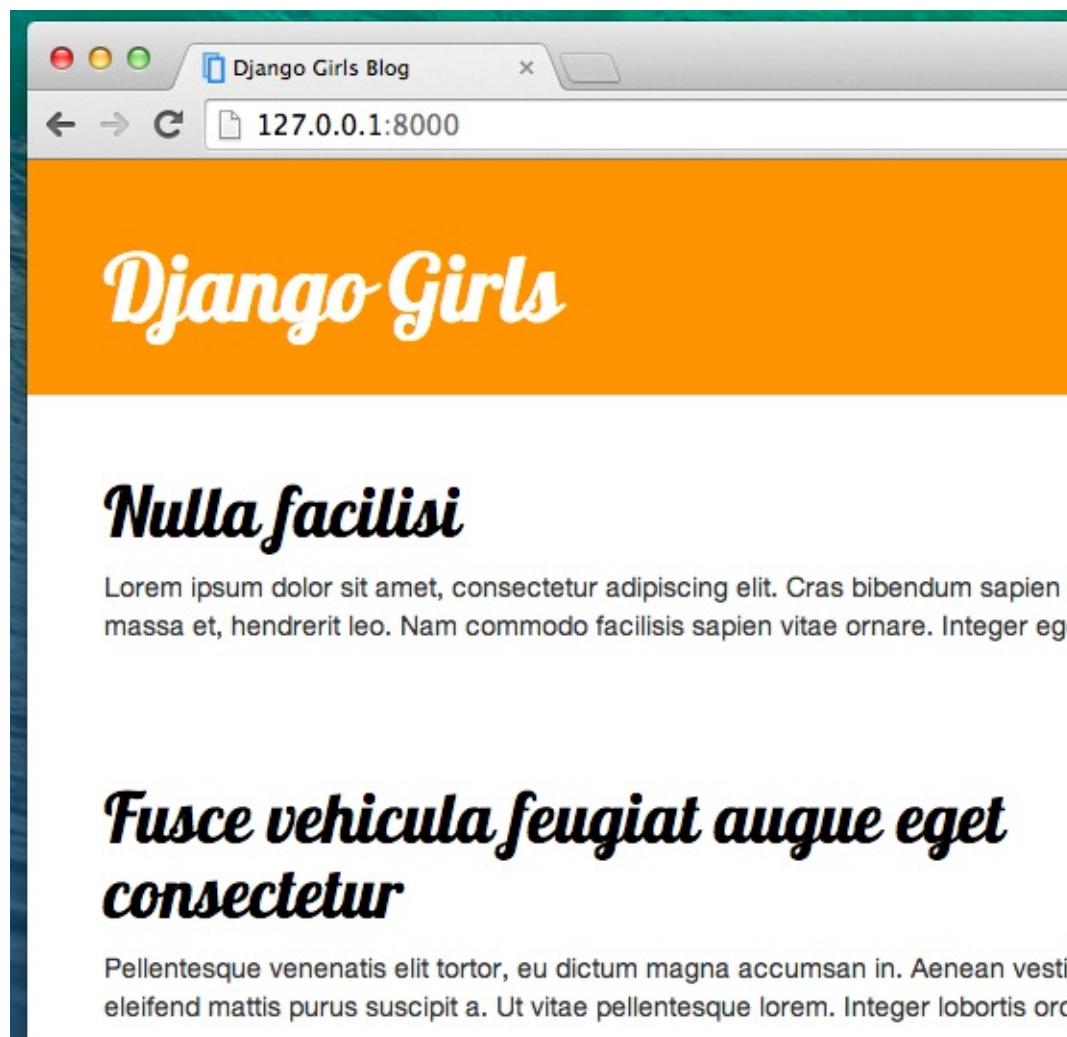
```
from django.shortcuts import render, get_object_or_404
```

E no final do arquivo, vamos adicionar a nossa view:

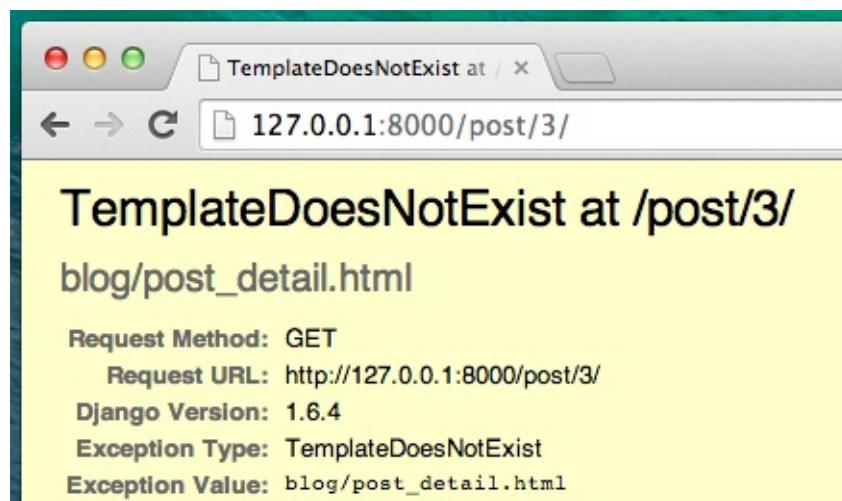
`blog/views.py`

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

É isso aí, 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 de um post do blog?



Ah não! Outro erro! Mas nós já sabemos como lidar com isso, né? Precisamos adicionar um template!

Criando um template para os detalhes do post

Vamos criar um arquivo em `blog/templates/blog` chamado `post_detail.html`.

Ele vai ter essa cara:

```
blog/templates/blog/post_detail.html
```

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

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

Mais uma vez estamos estendendo `base.html`. No bloco `content`, queremos exibir a data de publicação (`published_date`) do post (se houver), título e texto. Mas ainda temos algumas coisas importantes para discutir, certo?

`{% if ... %} ... {% endif %}` é uma tag de template que podemos usar quando queremos conferir alguma coisa. (Lembra de `if ... else ..` que vimos no capítulo **Introdução ao Python?**) Aqui queremos conferir se a `published_date` de um post não está vazia.

Pronto, podemos atualizar nossa página e ver se aquele `Page not found` sumiu.



Uhuu! Funcionou!

Hora do Deploy!

Seria bom ver se seu site ainda estará trabalhando no PythonAnywhere, né? Vamos tentar fazer a implantação novamente.

command-line

```
$ git status  
$ git add --all .  
$ git status  
$ git commit -m "Added view and template for detailed blog post as well as CSS for the site."  
$ git push
```

Agora, em um [console Bash do PythonAnywhere](#):

command-line

```
$ cd ~/<your-pythonanywhere-username>.pythonanywhere.com  
$ git pull  
[...]
```

(Lembre-se de substituir o `<your-pythonanywhere-username>` pelo seu username do PythonAnywhere, sem os símbolos < e >).

Atualizando os arquivos estáticos no servidor

Servidores como o PythonAnywhere tratam arquivos estáticos (como os arquivos CSS) de forma diferente dos arquivos em Python, por que assim podem otimizar para que eles carreguem mais rápido. Como resultado, sempre que alteramos nossos arquivos CSS, precisamos rodar um comando extra no servidor para dizer a ele que os atualize. O comando se chama `collectstatic`.

Comece ativando seu `virtualenv`, se ele já não estiver ativo (para isso, o PythonAnywhere usa um comando chamado `workon` que é bem parecido com o comando `source myenv/bin/activate` que você usa no seu computador):

command-line

```
$ workon <your-pythonanywhere-username>.pythonanywhere.com  
(ola.pythonanywhere.com)$ python manage.py collectstatic  
[...]
```

O comando `manage.py collectstatic` é mais ou menos como `manage.py migrate`. Agora, fazemos algumas mudanças no nosso código e dizemos ao Django que as aplique (*apply*) à coleção de arquivos estáticos, ou ao banco de dados.

De qualquer forma, estamos prontas para ir para a [aba Web](#) e clicar em **Reload** (atualizar).

Deve estar pronto! Arrasou :)

Formulários do Django

Por último, queremos uma forma legal de adicionar e editar as postagens do nosso blog. A ferramenta de administração do Django é legal, mas é um pouco difícil de personalizar e de deixar mais bonita. Com formulários, temos poder absoluto sobre nossa interface - podemos fazer quase tudo que pudermos imaginar!

Uma coisa legal do Django é que podemos tanto criar um formulário do zero, como criar um ModelForm que salva o resultado do formulário em um determinado modelo.

É exatamente isso que queremos fazer: criar um formulário para o nosso modelo `Post`.

Assim como todas as partes importantes do Django, forms têm seu próprio arquivo: `forms.py`.

Precisamos criar um arquivo com este nome dentro da pasta `blog`.

```
blog
└── forms.py
```

Agora vamos abri-lo e digitar o seguinte código:

`blog/forms.py`

```
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, o nosso modelo `Post` (`from .models import Post`).

`PostForm`, como você já deve suspeitar, é o nome do nosso formulário. Precisamos dizer ao Django que esse form é um `ModelForm` (pro Django fazer algumas mágicas para nós) – `forms.ModelForm` é o responsável por essa parte.

Em seguida, temos a `class Meta` em que dizemos ao Django qual modelo deverá ser usado para criar este formulário (`model = Post`).

Por fim, podemos dizer quais campos devem entrar no nosso formulário. Neste cenário, queremos que apenas o `title` e o `text` sejam expostos -- `author` deve ser a pessoa que está logada no sistema (nesse caso, você!) e `created_date` deve ser configurado automaticamente quando criamos um post (no código), correto?

E é isso! Tudo o que precisamos fazer agora é usar o formulário em uma view e mostrá-lo em um template.

Novamente, criaremos 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` chamado `page-header`:

`blog/templates/blog/base.html`

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

Note que queremos chamar nossa nova view de `post_new`. A classe `"glyphicon glyphicon-plus"` é fornecida pelo tema (bootstrap) que estamos usando, e nos mostrará um sinal de mais.

Depois de adicionar essa linha, o seu HTML vai ficar assim:

`blog/templates/blog/base.html`

```
{% load static %}
<html>
  <head>
    <title>Django Girls blog</title>
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap-theme.min.css">
    <link href="//fonts.googleapis.com/css?family=Lobster&subset=latin,latin-ext" rel='stylesheet' type='text/css'>
  >
    <link rel="stylesheet" href="{% static 'css/blog.css' %}">
  </head>
  <body>
    <div class="page-header">
      <a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a></h1>
    </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, o familiar erro `NoReverseMatch`, certo?

URL

Vamos abrir o arquivo `blog/urls.py` e escrever:

`blog/urls.py`

```
path('post/new', views.post_new, name='post_new'),
```

O código final deve se parecer com isso:

`blog/urls.py`

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

urlpatterns = [
  path('/', views.post_list, name='post_list'),
  path('post/<int:pk>/', views.post_detail, name='post_detail'),
  path('post/new/', views.post_new, name='post_new'),
]
```

Após recarregar a página, veremos um `AttributeError` por que não temos a view `post_new` implementada. Vamos adicioná-la agora.

View post_new

Hora de abrir o arquivo `blog/views.py` e adicionar as linhas seguintes com o resto das linhas `from`:

`blog/views.py`

```
from .forms import PostForm
```

E então a nossa `view`:

`blog/views.py`

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

Para criar um novo formulário `Post`, devemos chamar `PostForm()` e passá-lo para o template. Voltaremos a esta `view` depois, mas por enquanto, vamos criar um template para o formulário.

Template

Precisamos criar um arquivo `post_edit.html` na pasta `blog/templates/blog`. Para fazer o formulário funcionar, precisamos de muitas coisas:

- Temos que exibir o formulário. Podemos fazer isso com (por exemplo) `{{ form.as_p }}`.
- 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 `{% csrf_token %}`. Isso é muito importante, pois é isso que torna o nosso formulário seguro! Se você esquecer esta parte, o Django vai reclamar quando você tentar salvar o formulário:



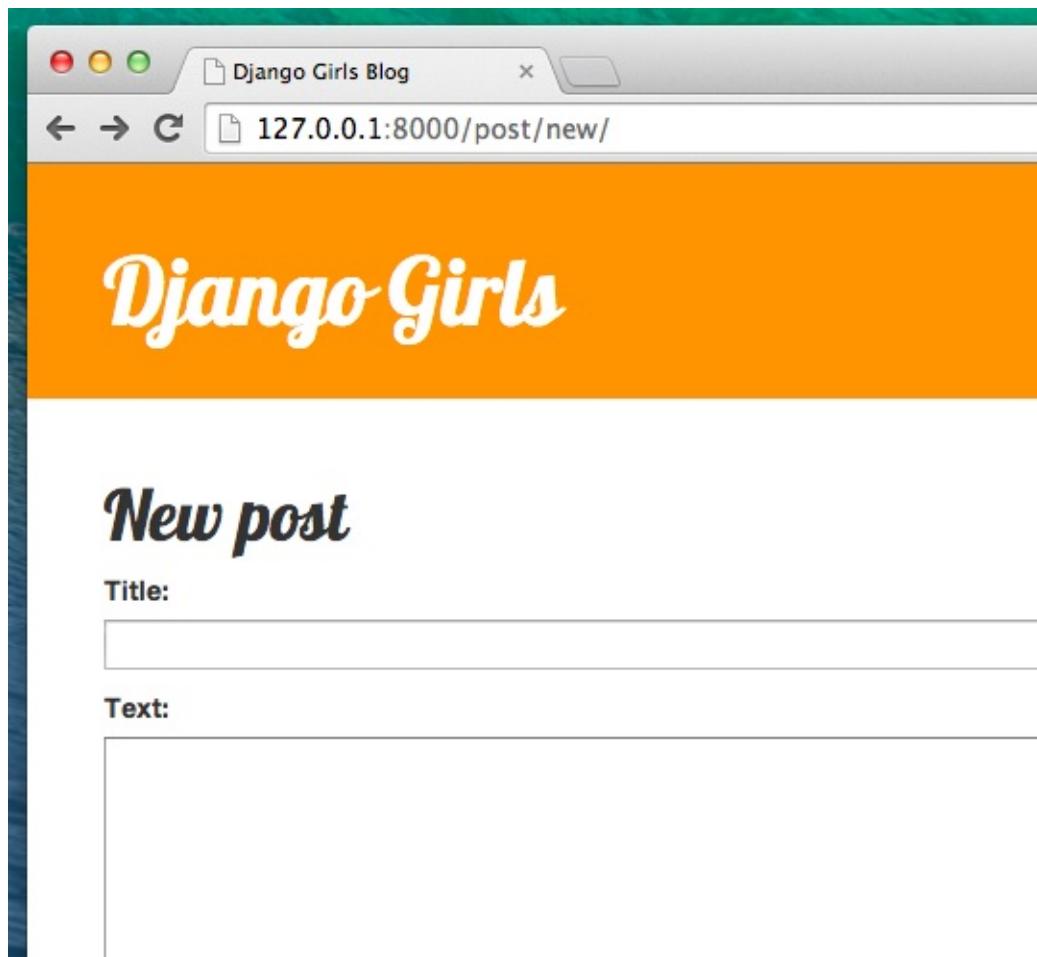
Legal, então vamos ver como ficou o HTML `post_edit.html`:

`blog/templates/blog/post_edit.html`

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

{% block content %}
    <h1>Nova postagem</h1>
    <form method="POST" class="post-form">{% csrf_token %}
        {{ form.as_p }}
        <button type="submit" class="save btn btn-default">Save</button>
    </form>
{% endblock %}
```

Hora de atualizar! Uhuu! Seu formulário apareceu!



Mas espere um minuto! O que vai acontecer quando você digitar alguma coisa nos campos `title` e `text` e tentar salvar?

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 view `post_new` é:

`blog/views.py`

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

Quando enviamos o formulário, somos trazidas de volta à mesma view, mas desta vez temos mais alguns dados no `request`, especificamente em `request.POST` (o nome não tem nada a ver com "post" de blog; tem a ver com o fato que estamos "postando" dados). Lembra que no arquivo HTML, nossa definição de `form` incluiu a variável `method="POST"`? Todos os campos vindos do "form" estarão disponíveis agora em `request.POST`. Não renomeie `POST` para nada diferente disso (o único outro valor válido para `method` é `GET`, mas não temos tempo para explicar a diferença).

Então em nossa `view` temos duas situações diferentes com as quais lidar: primeiro, quando acessamos a página pela primeira vez e queremos um formulário em branco e segundo, quando voltamos para a `view` com todos os dados do formulário que acabamos de digitar. Desse modo, precisamos adicionar uma condição (usaremos `if` para isso):

`blog/views.py`

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

É hora de preencher os pontos [...] . Se `method` é `POST` , queremos construir o `PostForm` com dados do formulário, certo? Faremos assim:

blog/views.py

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

O próximo passo é checar se o formulário está correto (todos os campos requeridos estão prontos 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!

blog/views.py

```
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 (já que houve um campo `author` em `PostForm`, e este campo é obrigatório). `commit=False` significa que não queremos salvar o modelo de `Post` ainda - queremos adicionar o autor primeiro. Na maioria das vezes você irá usar `form.save()`, sem `commit=False`, mas neste caso, precisamos fazer isso. `post.save()` vai preservar as alterações (adicionando o autor) e é criado um novo post no blog!

Finalmente, seria fantástico se pudéssemos ir à página `post_detail` , direto para o nosso recém-criado post no blog, né? Para fazer isso, precisaremos de mais uma importação:

blog/views.py

```
from django.shortcuts import redirect
```

Adicione isso logo no início do seu arquivo. Agora podemos dizer: "vá para a página `post_detail` para o post recém-criado":

blog/views.py

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

`post_detail` é o nome da visualização (view) à qual queremos ir. Lembra que essa view exige uma variável `pk`? Para passar isso para as `views` , usamos `pk=post.pk` , em que `post` é o recém-criado post do blog!

Ok, nós falamos muito, e agora queremos ver a cara da view completa, né?

blog/views.py

```

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

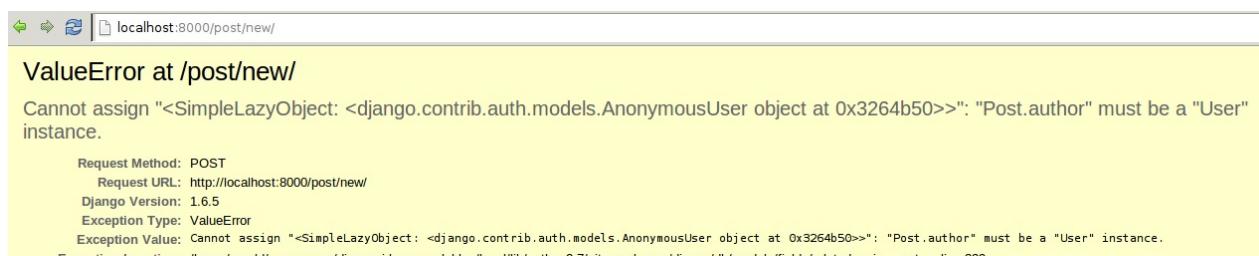
```

Vamos ver se funciona. Vá para a página <http://127.0.0.1:8000/post/new/>, adicione um `title` e o `text`, salve... e pronto! O novo post do blog é adicionado e somos redirecionadas à página de `post_detail`!

Você deve ter percebido que estamos estabelecendo a data de publicação antes de salvar o post. Mais tarde, vamos introduzir um botão de *Publicar* em **Django Girls Tutorial: Extensões**.

Isso é incrível!

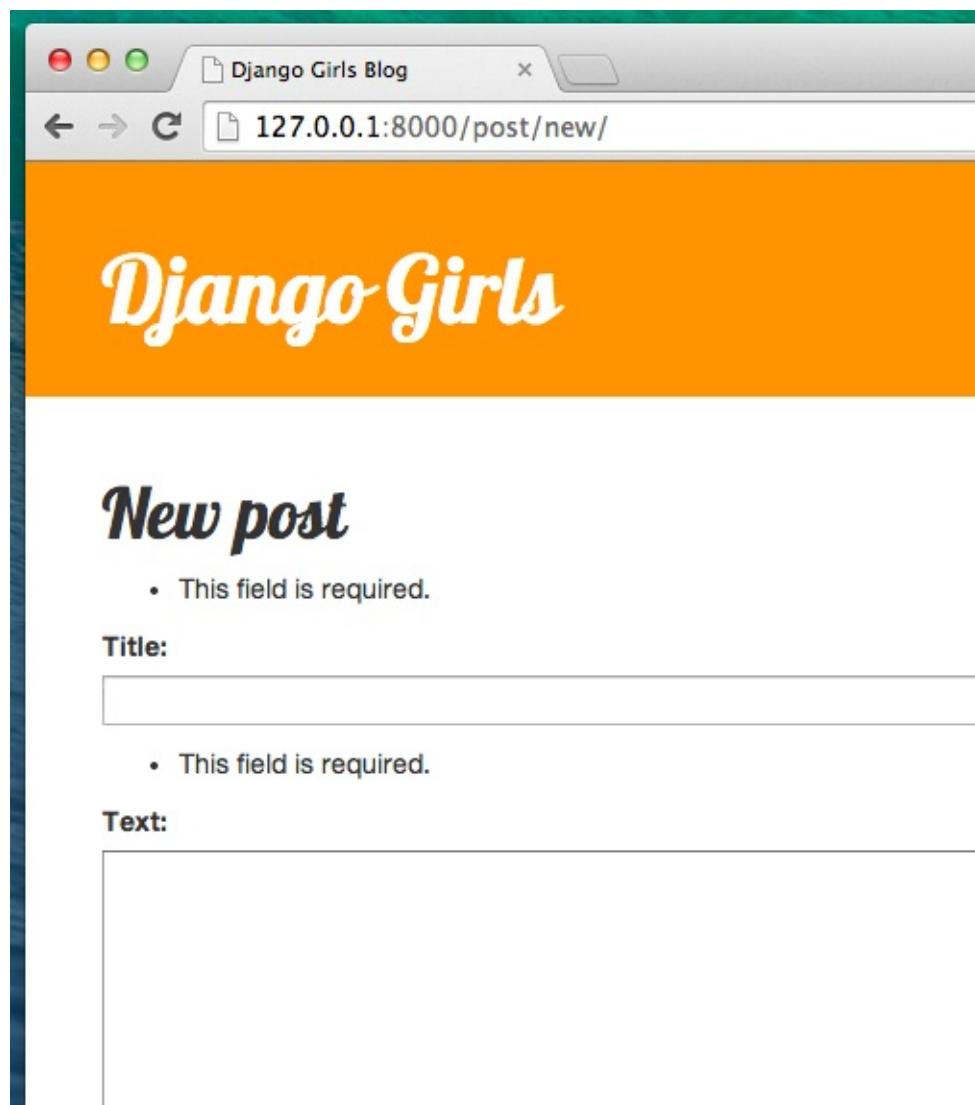
Como recentemente usamos a interface de administração do Django, o sistema entende que estamos logadas. Existem algumas situações que poderiam nos desligar do sistema (fechar o navegador, reiniciar banco de dados etc.). Se ao criar um post você receber erros que se referem à ausência de um usuário logado, vá até a página de admin <http://127.0.0.1:8000/admin> e faça login novamente. Isso vai resolver o problema temporariamente. Há um ajuste permanente esperando por você em **lição de casa: adicionar segurança ao seu site!**, um capítulo posterior ao tutorial principal.



Validação de formulários

Agora, mostraremos como os formulários do Django 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 são opcionais, então o Django, por padrão, espera que sejam definidos.

Tente salvar o formulário sem `title` e `text`. Adivinhe o que vai acontecer!



Django está confirmando que todos os campos de nosso formulário estão corretos. Não é incrível?

Editando o formulário

Agora sabemos como adicionar um novo formulário. Mas e se quisermos editar um que já existe? É muito parecido com o que acabamos de fazer. Vamos criar algumas coisas importantes rapidinho. (Se você não entender alguma coisa, pergunte para a sua monitora ou veja os capítulos anteriores -- já cobrimos todas essas etapas anteriormente.)

Abra `blog/templates/blog/post_detail.html` e adicione esta linha

`blog/templates/blog/post_detail.html`

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

agora, o template estará parecido com:

`blog/templates/blog/post_detail.html`

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

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

Em `blog/urls.py`, adicionamos esta linha:

`blog/urls.py`

```
path('post/<int:pk>/edit/', views.post_edit, name='post_edit'),
```

Vamos reutilizar o template `blog/templates/blog/post_edit.html`, então a última coisa que falta é uma view.

Vamos abrir `blog/views.py` e adicionar ao final do arquivo:

`blog/views.py`

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

Isso é quase igual à nossa view de `post_new`, né? Mas não inteiramente. Primeira coisa: passamos um parâmetro extra `pk` a partir da url. 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...

`blog/views.py`

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

... como quando apenas abrimos um formulário para editar esse post:

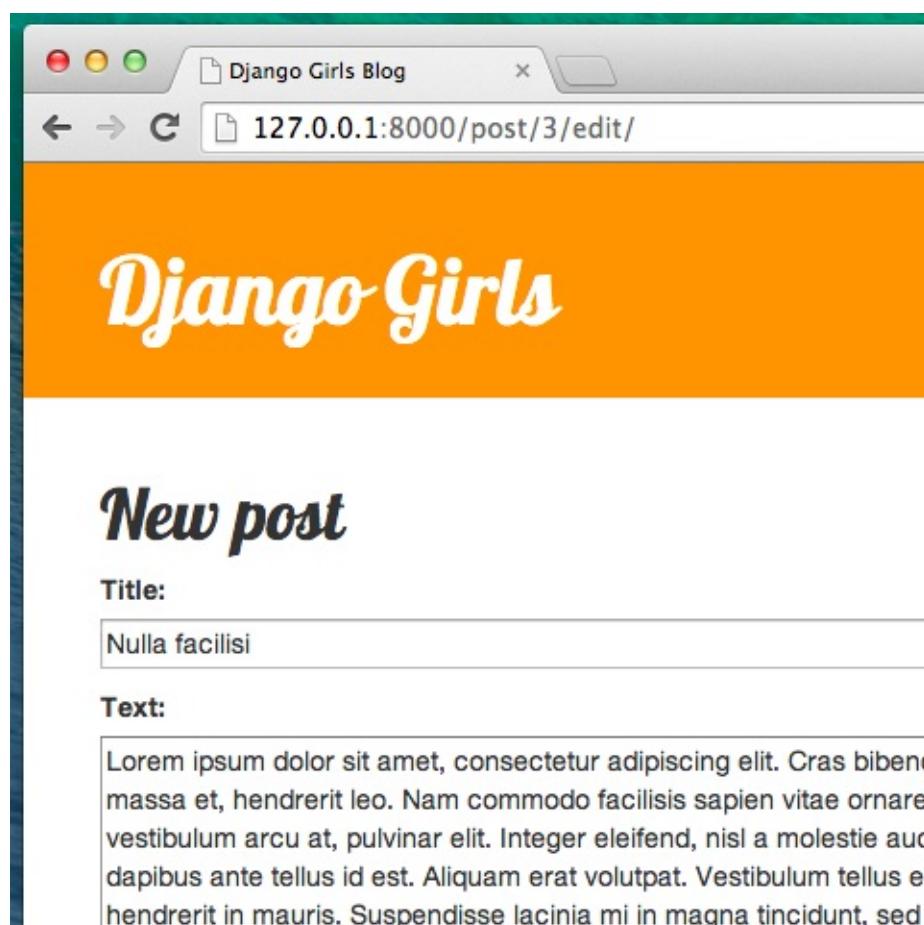
`blog/views.py`

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

Ok, vamos testar para ver se funcional! Vamos para a página `post_detail`. Deve haver um botão editar no canto superior direito:



Quando você clicar nesse botão, verá o formulário com a nossa postagem:



Sinta-se livre para mudar o título ou o texto e salvar as alterações!

Parabéns! Sua aplicação está ficando cada vez mais completa!

Se precisar de mais informações sobre formulários do Django, leia a documentação:

<https://docs.djangoproject.com/en/2.0/topics/forms/>

Segurança

Ser capaz de criar novos posts apenas clicando em um link é ótimo! Mas nesse momento, qualquer um que visitar nosso site poderá criar um novo post, e você isso provavelmente não quer isso. Vamos fazer com que o botão apareça apenas para você e para mais ninguém.

Em `blog/templates/blog/base.html`, procure nossa `div page-header` e a tag de link que você colocou mais cedo. Deve se parecer com:

`blog/templates/blog/base.html`

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

Vamos incluir outra tag `{% if %}` que irá apresentar o link somente para os usuários que estiverem logados como admin. No momento, é apenas você! Mude a tag `<a>` para que fique assim:

`blog/templates/blog/base.html`

```
{% if user.is_authenticated %}
<a href="{% url 'post_new' %}" class="top-menu"><span class="glyphicon glyphicon-plus"></span></a>
{% endif %}
```

Este `{% if %}` fará com que o link seja enviado ao navegador se o usuário que requisitou a página estiver logado. Isso não protege o blog completamente da criação de um novo post, mas é um bom começo. Vamos falar mais sobre segurança nas próximas lições.

Lembra do ícone Editar que acabamos de adicionar à nossa página de detalhes? Queremos fazer a mesma coisa com ele para que outras pessoas não possam editar as mensagens já existentes.

Abra `blog/templates/blog/post_detail.html` e encontre esta linha:

`blog/templates/blog/post_detail.html`

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

Altere-a para:

`blog/templates/blog/post_detail.html`

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

Você provavelmente está logada, então se atualizar a página, não verá nada de diferente. Carregue a página em um navegador novo ou em uma janela anônima (chamada "InPrivate" no Windows Edge), e então você verá que o link não aparece, e o ícone também não!

Mais uma coisa: hora de implantar!

Vamos ver se tudo isso funciona no PythonAnywhere. Hora de fazer outro deploy!

- Primeiro, faça o commit do seu novo código e dê o comando push para colocá-lo no Github:

command-line

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

- Então, em um [console Bash do PythonAnywhere](#):

command-line

```
$ cd ~/<your-pythonanywhere-username>.pythonanywhere.com  
$ git pull  
[...]
```

(Lembre-se de substituir o `<your-pythonanywhere-username>` pelo seu username do PythonAnywhere, sem os símbolos < e >).

- Finalmente, vá para a [aba Web](#) e clique **Reload**.

E deve ser isso! Parabéns :)

O que vem agora?

Parabéns! Você **arrasou**. Estamos orgulhosas! <3

O que fazer agora?

Antes de mais nada, faça uma pausa e relaxe. Você acabou de fazer algo realmente grande.

Depois disso, certifique-se de seguir o Django Girls no [Facebook](#) ou no [Twitter](#) para se manter atualizada.

Vocês podem recomendar outras fontes de informação?

Sim! Primeiro, vá em frente e tente nosso outro livro, chamado [Django Girls Tutorial: Extensions](#).

Mais pra frente, você pode tentar os recursos listados abaixo. Todos são muito bem recomendados!

- [Tutorial oficial do Django](#)
- [Tutorial do New Coder](#)
- [Curso de Python do Code Academy](#)
- [Curso de HTML & CSS do Code Academy](#)
- [Tutorial do Django Carrots](#)
- [Livro "Learn Python The Hard Way" \(Aprenda Python do Jeito Difícil\)](#)
- [Videoaulas "Começando com o Django"](#)
- [Livro "Two Scoops of Django 1.11: Best Practices for Django Web Framework" \(Duas Colheres de Django 1.11: Melhores Práticas para o Framework Web Django\)](#)
- ["Hello Web App: Learn How to Build a Web App" \(Aprenda Como Construir uma Aplicação Web\) -- você também pode solicitar uma licença gratuita do eBook entrando em contato com a autora Tracy Osborn em \[tracy@limedaring.com\]\(mailto:tracy@limedaring.com\)](#)