Projeto Django

1 - Criar/entrar na pasta do projeto

```
$ mkdir helloworld
$ cd helloworld
```

2 - Criar ambiente virtual

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

With this command, I'm asking Python to run the venv package, which creates a virtual environment named venv. The first venv in the command is the name of the Python virtual environment package, and the second is the virtual environment name that I'm going to use for this particular environment. If you find this confusing, you can replace the second venv with a different name that you want to assign to your virtual environment.
In general I create my virtual environments with the name venv in the project directory, so whenever I cd into a project I find its corresponding virtual environment.

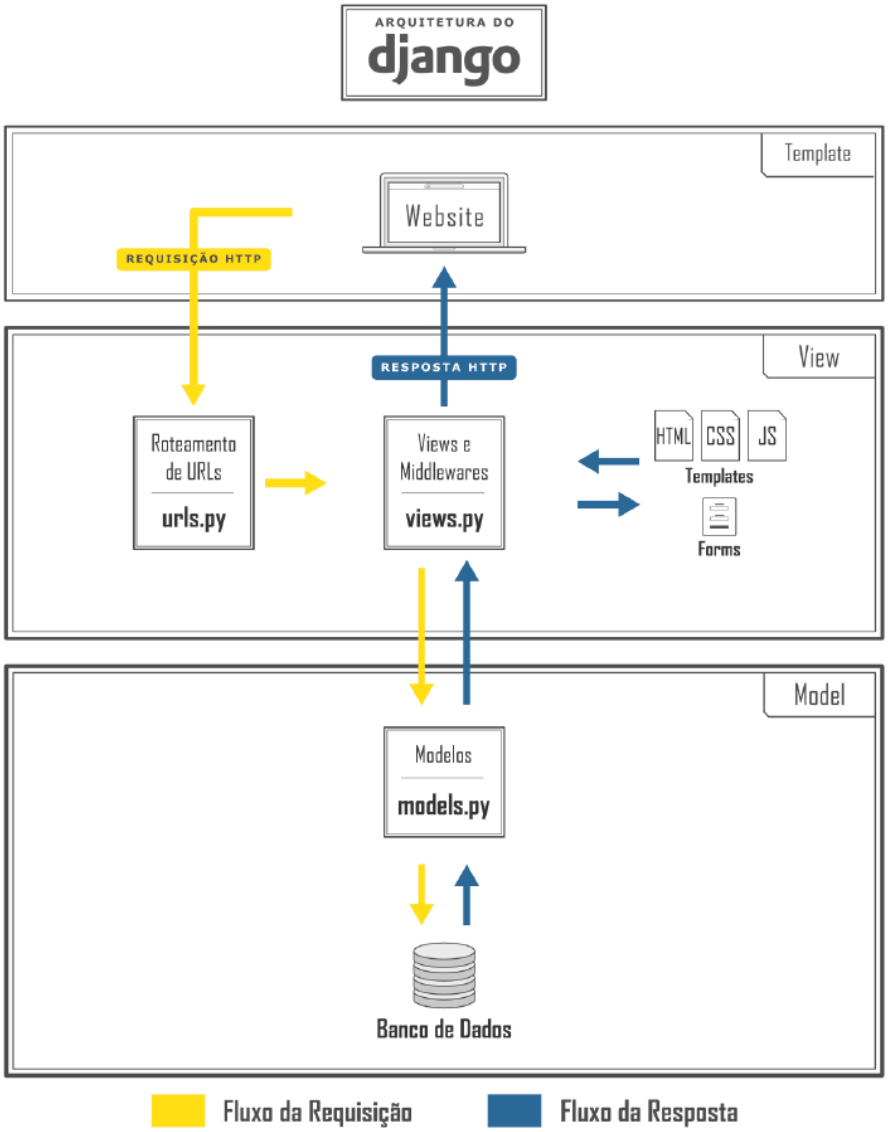
If you are using any version of Python older than 3.4 (and that includes the 2.7 release), virtual environments are not supported natively. For those versions of Python, you need to download and install a third-party tool called virtualenv before you can create virtual environments. Once virtualenv is installed, you can create a virtual environment with the following command:

```
$ virtualenv venv
```

3 - Ativar ambiente virtual

```
$ source venv/bin/activate
(venv) $
```

DJANGO



4 – Instalando DJANGO

pip install django

ou para copiar de outro projeto
na raiz do projeto original

```
pip freeze > requirements.txt
```

na raiz do novo projeto

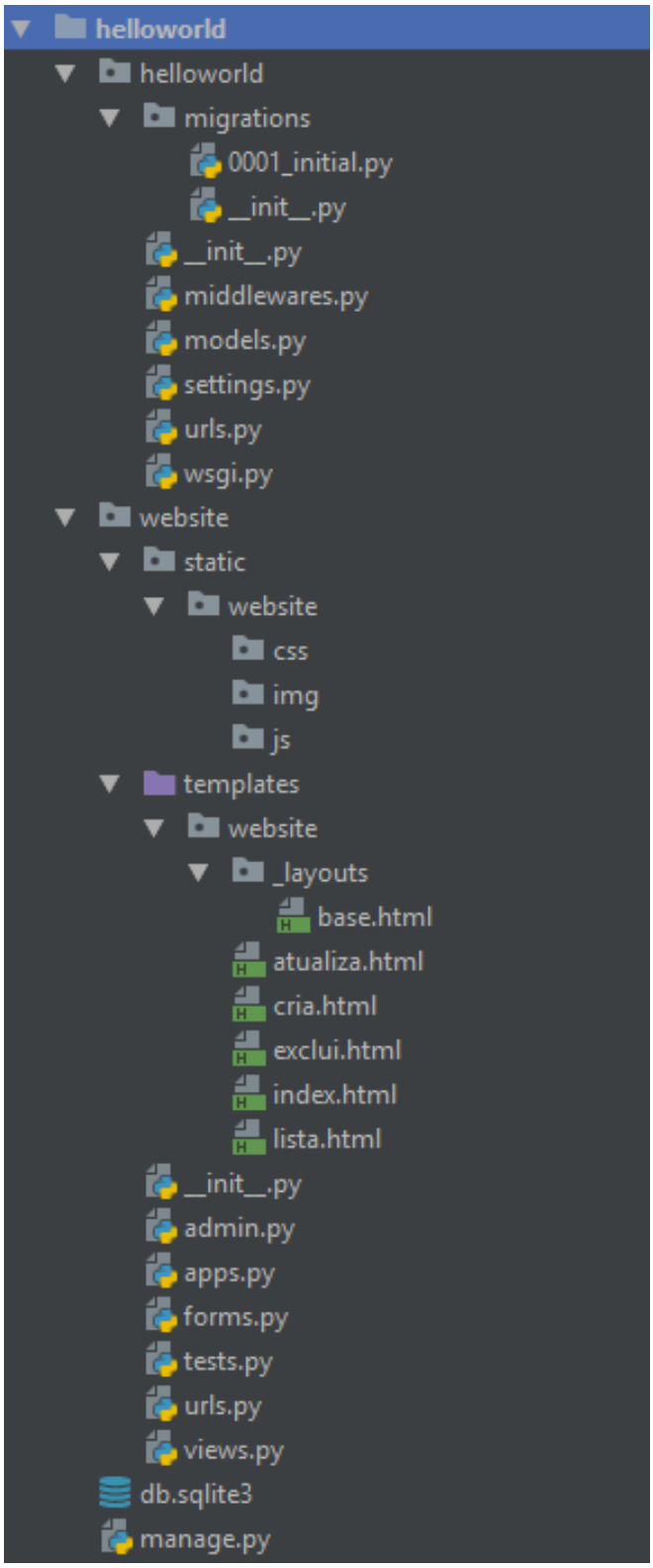
```
pip install –r requirements.txt
```

5 – Criando a estrutura do projeto

django-admin.py startproject helloworld .

6 – Iniciar o projeto
na raiz do servidor
python manage.py runserver

7 – Criando o APP neste caso o app se chama website
django-admin.py startapp `website`
Agora, vamos criar algumas pastas para organizar a estrutura da nossa aplicação.
Primeiro, crie a pasta templates dentro de website.
Dentro dela, crie uma pasta website e dentro dela, uma pasta chamada _layouts. Crie também a pasta static dentro de website, para guardar os arquivos estáticos (arquivos CSS, Javascript, imagens, fontes, etc). Dentro dela crie uma pasta website, por questões de namespace. Dentro dela, crie: uma pasta css, uma pasta img e uma pasta js.



8 - Atualizando a configuração INSTALLED_APPS no arquivo de configuração helloworld/settings.py

```
INSTALLED_APPS = [
'django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
```

```
'django.contrib.messages',
'django.contrib.staticfiles',
'helloworld',
'website'
]
```

Agora, vamos fazer algumas alterações na estrutura do projeto para organizar e centralizar algumas configurações.

Primeiro, vamos passar o arquivo de modelos `models.py` de `/website` para `/helloworld`, pois os arquivos comuns ao projeto vão ficar centralizados no app `helloworld` (geralmente temos apenas um arquivo `models.py` para o projeto todo).

Como não temos mais o arquivo de modelos na pasta `/website`, podemos, então, excluir a pasta `/migrations` e o `migrations.py`, pois estes serão gerados e gerenciados pelo app `helloworld`.

Estrutura inicial do projeto

```
/helloworld
    - __init__.py
    - settings.py
    - urls.py
    - wsgi.py
    - models.py
/website
    - __init__.py
    - admin.py
    - apps.py
    - tests.py
    - views.py
- manage.py
```

Criar modelos (TABELAS DO BANCO DE DADOS)

url.py do projeto

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

from django.urls.conf import include

urlpatterns = [
    path('', include('appp.urls', namespace='appp')),

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

apps.py do app

```
from django.apps import AppConfig


class ApppConfig(AppConfig):
    name = 'appp'
```

Admin.py do app para incluir os models no /admin

```
from django.contrib import admin

# Register your models here.
from proj.models import Dimdata, Dimlocal, FRegistro, Publication

admin.site.register(Dimdata)
admin.site.register(Dimlocal)
admin.site.register(FRegistro)
admin.site.register(Publication)
```

```
python manage.py createsuperuser
```

No Django, os modelos são descritos no arquivo models.py.
Ele já foi criado no Capítulo anterior e está presente na pasta helloworld/models.py.

Portanto, toda vez que você alterar o seu modelo, não se esqueça de executar:

python manage.py makemigrations projeto

Para que o Django as aplique, são necessárias três coisas, basicamente:
- **1.** Que a configuração da interface com o banco de dados esteja descrita no settings.py
- **2.** Que os modelos e *migrations* estejam definidos para esse projeto.
- **3.** Execução do comando migrate

Agora só falta **executar o comando** migrate, propriamente dito!
Para isso, vamos para a raíz do projeto e executamos:

python manage.py migrate

Com nossa classe Funcionário modelada, vamos agora ver a API de acesso à dados provida pelo Django para facilitar muito a nossa vida! Vamos testar a adição de um novo funcionário utilizando o shell do Django. Para isso, digite o comando: 1

python manage.py shell

no shell:

```
from helloworld.models import Funcionario

funcionario = Funcionario(
nome='Marcos',
sobrenome='da Silva',
cpf='015.458.895-50',
tempo_de_servico=5,
remuneracao=10500.00
)

funcionario.save()
```

A CAMADA VIEW

É nela que descreveremos a lógica de negócios da nossa aplicação, ou seja: é nela que vamos descrever os métodos que irão processar as requisições, formular respostas e enviá-las de volta ao usuário.

A partir da URL que o usuário quer acessar (/funcionarios, por exemplo), o Django irá rotear a requisicão para quem irá tratá-la.
Mas primeiro, o Django precisa ser informado para **onde** mandar a requisição.
Fazemos isso no chamado **URLconf** e damos o nome a esse arquivo, por convenção, de urls.py!

Geralmente, temos um arquivo de rotas por *app* do Django. Portanto, crie um arquivo urls.py dentro da pasta /helloworld e outro na pasta /website.

Como o *app helloworld* é o núcleo da nossa aplicação, ele faz o papel de centralizador de rotas, isto é:
• Primeiro, a requisição cai no arquivo /helloworld/urls.py e é roteada para o *app* correspondente.
• Em seguida, o URLConf do *app* (/website/urls.py, no nosso caso) vai rotear a requisição para a *view* que irá processar a requisição.

```
1   from django.urls.conf import include
2   from django.contrib import admin
3   from django.urls import path
4   urlpatterns = [
5   # Inclui as URLs do app ‘website’
6   path('', include('website.urls', namespace='website')),
7   # Interface administrativa
8   path('admin/', admin.site.urls),
9   ]
10
11
```

Como o *app helloworld* é o núcleo da nossa aplicação, ele faz o papel de centralizador de rotas, isto é:
• Primeiro, a requisição cai no arquivo /helloworld/urls.py e é roteada para o *app* correspondente.
• Em seguida, o URLConf do *app* (/website/urls.py, no nosso caso) vai rotear a requisição para a *view* que irá processar a requisição.

Dessa forma, o arquivo helloworld/urls.py deve conter:

```
1
2
3
4
5
6
7
```

```
from django.urls.conf import include
from django.contrib import admin
from django.urls import path
urlpatterns = [
# Inclui as URLs do app ‘website’
path('', include('website.urls',
namespace='website')),
# Interface administrativa
path('admin/', admin.site.urls),
```

```
8                                                   ]
9
10
11
```

Exemplo de views.py do app

```
from django.shortcuts import render
from django.views.generic import TemplateView, CreateView, ListView, DetailView
from django.urls import reverse_lazy

from proj.models import Dimdata,Dimlocal,FRegistro
from appp.forms import AddFregistroForm

# Create your views here.

class IndexTemplateView(TemplateView):
    template_name = "appp/index.html"

class LocaisList(ListView):
    model = Dimlocal
    template_name = "appp/locais.html"
    paginate_by = 25

class LocalDetailView(DetailView):
    template_name = "appp/local_detail.html"
    model = Dimlocal

class RegCreateView(CreateView):
    template_name = "appp/add_reg.html"
    model = FRegistro
    form_class = AddFregistroForm
    success_url = reverse_lazy("appp:add_reg")
```

DEPLOY

Gerando arquivos com os requerimentos a instalar:
pip freeze > requirements.txt

Vamos transformar nossa pasta em um repositório local:
git init

Criar o repositório em github.com e copiar o link

Agora precisamos linkar esse nosso repositório local ao nosso repositório remoto.

git remote add origin https://github.com/fperazzo/covid19.git

Agora, adicione tudo que você tem nessa pasta para "commitar" (não vou detalhar o que é um commit agora, foge do escopo do post) seus arquivos para seu repositório remoto (o do GitHub).
git add * todos arquivos
git add . ---- qdo tiver arquivo gitignore

Agora faça o commit dessa sua adição, que é indicar para o git as suas alterações feitas até o momento
git commit -m "adicionando os arquivos do app de relaxamento"

“git pull upstream master”: com esse comando vocês baixarão para as suas pastas do projeto qualquer atualização do repositório upstream que vocês não tenham ainda

Agora você precisa, antes de enviar para o GitHub, criar uma branch chamada gh-pages, a fim de que seu app esteja visível online em formato de site para quem quiser ver.

Um branch em git é uma divisão do seu projeto. Por exemplo, se você está desenvolvendo um app e vai criar uma nova funcionalidade, para preservar a segurança da sua primeira versão funcional, você não cria as alterações em cima dessa primeira versão, mas cria uma branch que irá receber as alterações e após terminar tudo, você junta as alterações da sua branch secundária à sua branch principal. Isso evita erros de controle de versão.

Para seu site ficar visível na web usando o GitHub, seu projeto deve ter uma branch chamada gh-pages. Vamos criá-la:
git checkout -b gh-pages

Para atualizar os arquivos na rede:
git push origin gh-pages

Agora você pode enviar tudo para seu repositório remoto (o do GitHub):
git push -u origin gh-git
Agora precisamos criar um app no Heroku.
heroku create covid20192019

linkar nosso código local com o repositório remoto do Heroku,
heroku git:remote -a covid20192019

Agora basta mandar o código para o Heroku:
git push heroku master

Para apagar o branch localmente:

git branch -D <nome do branch>

Para apagar o branch remotamente:

git push <nome do origin> <nome do branch> --delete

https://medium.com/reprogramabr/como-trabalhar-em-equipe-usando-github-e-gitbash-cde37cab6526

DEBUG

Incluir no código no ponto onde quer ver as variáveis.

from ipdb import set_trace; set_trace()

ipdb> form
<Eq1_Form bound=True, valid=Unknown, fields=(idmetcol)>

ipdb> form.fields
{'idmetcol': <django.forms.models.ModelMultipleChoiceField object at 0x7fa6f566ba50>}

ipdb> form.fields['idmetcol']
<django.forms.models.ModelMultipleChoiceField object at 0x7fa6f566ba50>

ipdb> dir(form.fields['idmetcol'])
['__class__', '__deepcopy__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slotnames__', '__str__', '__subclasshook__', '__weakref__', '_check_values', '_get_choices', '_get_queryset', '_queryset', '_set_choices', '_set_queryset', 'bound_data', 'choices', 'clean', 'default_error_messages', 'default_validators', 'disabled', 'empty_label', 'empty_values', 'error_messages', 'get_bound_field', 'get_limit_choices_to', 'has_changed', 'help_text', 'hidden_widget', 'initial', 'iterator', 'label', 'label_from_instance', 'label_suffix', 'limit_choices_to', 'localize', 'prepare_value', 'queryset', 'required', 'run_validators',

'show_hidden_initial', 'to_field_name', 'to_python', 'valid_value', 'validate', 'validators', 'widget', 'widget_attrs']

ipdb> form.cleaned_data
{'idmetcol': <QuerySet [<Orcamento_Situacao: Orc ID 1 chave_proj B0000 Final Produtor BM 2 7/8"EU>]>}

Quando QuerySets são avaliados¶

https://docs.djangoproject.com/en/3.0/ref/models/querysets/

Internamente, um QuerySet pode ser construído, filtrado, cortado e geralmente distribuído sem atingir o banco de dados. Na verdade, nenhuma atividade do banco de dados ocorre até você fazer algo para avaliar o conjunto de consultas.

Você pode avaliar um QuerySet das seguintes maneiras:

Iteração. Um QuerySet é iterável e executa sua consulta ao banco de dados na primeira vez que você itera sobre ele. Por exemplo, isso imprimirá o título de todas as entradas no banco de dados:

```
for e in Entry.objects.all():
    print(e.headline)
```

Nota: não use isso se tudo que você deseja fazer é determinar se existe pelo menos um resultado. É mais eficiente usar `exists()`

Fatiamento. Conforme explicado em Limitando o QuerySets, um QuerySet pode ser fatiado, usando a sintaxe de fatiamento de array do Python. Fatiar um QuerySet não avaliado geralmente retorna outro QuerySet não avaliado, mas o Django executará a consulta ao banco de dados se você usar o parâmetro "step" da sintaxe da fatia e retornará uma lista. Fatiar um QuerySet que foi avaliado também retorna uma lista.

Observe também que, embora fatiar um QuerySet não avaliado retorne outro QuerySet não avaliado, não é permitido modificá-lo ainda mais (por exemplo, adicionar mais filtros ou modificar pedidos), pois isso não se traduz bem no SQL e também não teria um significado claro.

Pickling=Decapagem / armazenamento em cache(caching). Consulte a seção a seguir para obter detalhes sobre o que está envolvido ao selecionar QuerySets. O importante para os propósitos desta seção é que os resultados sejam lidos no banco de dados.

repr (). Um QuerySet é avaliado quando você chama repr () nele. Isso é por conveniência no intérprete interativo Python, para que você possa ver imediatamente seus resultados ao usar a API interativamente.

len (). Um QuerySet é avaliado quando você chama len () nele. Isso, como você pode esperar, retorna o tamanho da lista de resultados.

Nota: Se você precisar determinar apenas o número de registros no conjunto (e não precisar dos objetos reais), será muito mais eficiente gerenciar uma contagem no nível do banco de dados usando o SELECT COUNT do SQL (*). O Django fornece um método count () exatamente por esse motivo.

List(). Força a avaliação de um QuerySet chamando list () nele. Por exemplo:

```
entry_list = list(Entry.objects.all())
```

bool(). bool (). Testar um QuerySet em um contexto booleano, como usar bool (), ou, e / ou uma instrução if, fará com que a consulta seja executada. Se houver pelo menos um resultado, o QuerySet é True, caso contrário, False. Por exemplo:

```
if Entry.objects.filter(headline="Test"):

   print("There is at least one Entry with the headline Test")
```

Observação: se você deseja determinar se existe pelo menos um resultado (e não precisa dos objetos reais), é mais eficiente usar exits( ).

COMANDOS DOCKER

Spin down the development containers:

```
$ docker-compose down -v
```

Test:

```
$ docker-compose -f docker-compose.prod.yml up -d --build
$ docker-compose -f docker-compose.prod.yml exec web python manage.py
migrate --noinput
$ docker-compose -f docker-compose.prod.yml exec web python manage.py
collectstatic --no-input --clear
```

Again, requests to `http://localhost:1337/staticfiles/*` will be served from the "staticfiles"

You can also verify in the logs -- via

`docker-compose -f docker-compose.prod.yml logs -f`

```
$ docker-compose up -d --build
$ docker-compose exec web python manage.py startapp upload
```