

**XGH - Desenvolvimento de Software Ltda**

# **Documento de Tutoriais**

## **SNAPture**

**Equipe 02**

**06/2023**

# Tutorial 01 - Instalação e Configuração das Ferramentas e Ambiente de Desenvolvimento

Autores: Ingrid Miranda dos Santos, Enzo Eduardo Cassiano Ibiapina e Ciro Olímpio de Melo.

Será ensinado como instalar a IDE escolhida VS Code, como instalar a linguagem Python, como ativar um ambiente virtual, e como instalar e criar um projeto no Framework Django. Já que algumas instruções diferem de acordo com o sistema operacional, elas serão separadas entre Windows e Linux:

## Windows

Para instalar o VSCode na sua máquina windows, faça download do instalador localizado sob o símbolo do SO em <https://code.visualstudio.com/download>. Após isso, é só seguir o setup manager da instalação marcando as opções adicionais que preferir.

Para instalar o python na máquina, baixe a última versão da linguagem em <https://www.python.org/downloads/>. Após isso, basta executar o arquivo .exe gerado. Para verificar se a instalação ocorreu com sucesso execute *"python --version"* no prompt de comando.

Uma vez que isso estiver feito, instale as extensões de python para o seu VSCode e cheque o path para o seu python interpreter usando o comando "Ctrl+Shift+P".

Navegue até a pasta do projeto. Para criar um ambiente virtual, execute o comando *"python -m venv [nome\_pasta\_virtual]"*. Para ativá-lo, navegue até a pasta Scripts e execute o aplicativo 'activate'. Após isso, o nome do ambiente virtual aparecerá antes do caminho atual do prompt de comando.

## Linux (Ubuntu)

Para instalar o VSCode, abra o terminal e digite o comando *"sudo snap install code --classic"*, utilizando o sistema snap (caso a distro não possua o sistema snap, instale-o com o comando *"sudo apt install snapd"*). Para abrir a IDE, abra o programa pelo menu iniciar, ou pelo terminal com o comando *"code"*.

Para instalar o Python, digite o comando *"sudo apt install python3"* no prompt de comando. Verifique a instalação executando o comando *"python3 --version"*, o resultado esperado deve ser a versão instalada do Python.

Para criar um ambiente virtual, navegue até a pasta do projeto e digite o comando *"python3 -m venv [nomeDaPasta]"* Para ativá-lo, digite o comando *"source [nomeDaPasta]/bin/activate"*. Para desativá-lo, execute o comando *"deactivate"*.

## Instruções comuns

Nota: altere os comandos *"python [...]"* para *"python3 [...]"* se estiver usando o Linux.

Com o ambiente virtual ativado, agora podemos instalar o framework Django. Execute o comando `"pip install django"`. Para verificar se a instalação ocorreu com sucesso execute `"python -m django --version"`.

No Django, um projeto é composto por várias aplicações(apps). Esses apps representam funcionalidades que podem ser desacopladas e reutilizadas em outros projetos. Para inicializar um projeto django, execute o comando: `"django-admin startproject [nome]"`. Para criar uma aplicação dentro do projeto, utilize o comando: `"django-admin startapp [nome]"`.

Para executar o projeto, navegue até a sua raiz e execute o comando `"python manage.py runserver"`. Entre na URL que ele retorna e veja tudo funcionando.

## Tutorial 02 - Aplicação Web Básica com Formulário

Autor: João Pedro Monteiro da Silva Barros.

Será ensinado como criar uma aplicação web básica com formulário HTML. A aplicação em questão realiza o controle de visitantes de um condomínio, projeto paralelo de um dos membros.

Dentro do ambiente virtual, instale a biblioteca django-widget-tweaks: *"pip install django-widget-tweaks"*.

Com um projeto inicial do Django pronto, crie um aplicativo para a entidade visitantes: *"django-admin startapp visitantes"*.

Dentro do aplicativo de visitantes, no arquivo models.py, crie os campos necessários para visitantes no banco de dados.

```
nome_completo = models.CharField(
    verbose_name="Nome completo",
    max_length=194,
)

cpf = models.CharField(
    verbose_name="CPF",
    max_length=11,
)

data_nascimento = models.DateField(
    verbose_name="Data de nascimento",
    auto_now_add = False,
    auto_now = False,
)

numero_casa = models.PositiveSmallIntegerField(
    verbose_name="Número da casa a ser visitada",
    blank=True,
    null=True,
)

placa_veiculo = models.CharField(
    verbose_name="Placa do veículo",
    max_length=7,
    blank=True,
    null=True,
)

placa_veiculo_foto = models.ImageField(
    verbose_name="Foto da placa do veículo",
    upload_to="uploads/",
    max_length=500,
    null=True,
    blank=True,
)

horario_chegada = models.DateTimeField(
    verbose_name="Horário de chegada",
    auto_now_add=True,
)
```

Crie um arquivo forms.py da maneira especificada abaixo para tratar dos campos que serão necessários no formulário. Na linha 6, é informada qual tabela do banco de dados o formulário irá referenciar. Nos fields são colocados os nomes dos campos da tabela que aparecerão no formulário.

```
1 from django import forms
2 from visitantes.models import Visitante
3
4 class VisitanteForm(forms.ModelForm):
5     class Meta:
6         model = Visitante
7         fields = [
8             "nome_completo", "cpf", "data_nascimento", "numero_casa", "placa_veiculo", "placa_veiculo_foto"
9         ]
10        error_messages = {
11            "nome_completo": {
12                "required": "O nome completo do visitante é obrigatório para o registro"
13            },
14            "cpf": {
15                "required": "O cpf do visitante é obrigatório para o registro"
16            },
17            "data_nascimento": {
18                "required": "A data de nascimento do visitante é obrigatória para o registro",
19                "invalid": "Por favor, informe um formato válido para a data de nascimento (DD/MM/AAAA)"
20            },
21            "numero_casa": {
22                "required": "O número da casa a visitar é obrigatória para o registro"
23            }
24        }
25
```

Dentro do arquivo `views.py`, crie uma `view` referente à página do formulário chamada `registrar_visitante`. Verifique se a função recebe um `request` com o método `POST`. Este sendo o caso, o formulário deve ser instanciado e salvo na sua tabela do banco de dados. Depois, o usuário é redirecionado à página `/index/` junto com o formulário que é passado dentro de um contexto que é renderizado juntamente ao template daquela página.

```
10 def registrar_visitante(request):
11
12     form = VisitanteForm()
13
14     if request.method == "POST":
15         form = VisitanteForm(request.POST, request.FILES)
16
17         if form.is_valid():
18             # Salva os dados do formulário no banco de dados, mas não faz o commit
19             visitante = form.save(commit=False)
20
21             # Define o campo "registrado_por" do visitante como o porteiro atualmente autenticado
22             visitante.registrado_por = request.user.porteiro
23
24             # Salva o visitante no banco de dados
25             visitante.save()
26
27             # Mensagem de sucesso gerada quando há um registro de visitante
28             messages.success(
29                 request,
30                 "Visitante registrado com sucesso"
31             )
32
33             return redirect("index")
34
35     # Cria um contexto contendo o nome da página e o formulário
36     # Com o contexto se consegue utilizar o valor de cada chave diretamente no HTML
37     context = {
38         "nome_pagina": "Registrar Visitante",
39         "form": form,
40     }
41
42     return render(request, "registrar_visitante.html", context)
43
```

Após renderizar, podemos utilizar o `django-widget-tweaks` para iterar pelo formulário já formatado dentro do `html`, fazendo tratamento de erros, obrigatoriedades, e controle.

```
<form method="post" enctype='multipart/form-data'>
  <div class="form-row">
    {% csrf_token %}

    {% for field in form %}
      <div class="form-group col-md-12">
        <label>{{ field.label }} {% if field.field.required %} * {% endif %}</label>
        {% render_field field class="form-control" %}
      </div>
    {% endfor %}
  </div>

  <div class="text-right">
    <a href="{% url 'index' %}" class="btn btn-secondary text-white" type="button">
      <span class="text">Cancelar</span>
    </a>

    <button class="btn btn-primary" type="submit">
      <span class="text">Registrar Visitante</span>
    </button>
  </div>
</form>
```

Formulário de registro de visitante:

Registrar Visitante

Formulário para registro de novo visitante

O asterisco (\*) indica que o campo é obrigatório

Nome completo \*

CPF \*

Data de nascimento \*

Número da casa a ser visitada

Placa do veículo

Foto da placa do veículo

Nenhum arquivo escolhido

## Tutorial 3: Instalação e Configuração do Banco de Dados

Autor: Ciro Olímpio de Melo

Será mostrado como está configurado o banco de dados sqlite3.

Ao executar o par de comandos *makemigrations* e *migrate* presentes no fim do T1, é gerado um arquivo `db.sqlite3` na pasta base do nosso projeto django. Esse é o banco de dados padrão gerado pelo framework.

No *settings.py* da pasta do nosso projeto, vemos o diretório para onde o nosso banco de dados está atualmente configurado:

```
80 DATABASES = {
81     'default': {
82         'ENGINE': 'django.db.backends.sqlite3',
83         'NAME': BASE_DIR / 'db.sqlite3',
84     }
85 }
```

Sempre que formos adicionar uma nova tabela no nosso banco de dados, precisamos colocar nos `models.py` os atributos que estarão presentes na mesma. Por exemplo, a nossa tabela de imagens tem um espaço para imagem e outra para descrição da imagem, conforme vemos no exemplo existente no nosso T5:

```
4 class Imagem(models.Model):
5     imagem = models.ImageField(upload_to="imgs/", null=True, blank=True)
6     descricao = models.TextField(blank=True)
```

Uma vez criada a tabela, precisamos definir como ela será preenchida. Geralmente isso será realizado por meio de diferentes formulários existentes nos aplicativos do nosso projeto. Por exemplo, nossa tabela de imagens é preenchida por um formulário presente no nosso aplicativo de imagens:

```
4 class ImagemForm(forms.ModelForm):
5     class Meta:
6         model = Imagem
7         fields = [
8             'imagem',
9             'descricao'
10        ]
```

Assim que essas estruturas estejam definidas e prontas, temos que realizar os dois comandos a seguir:

“*python manage.py makemigrations*”, que irá verificar a integridade das mudanças realizadas no projeto, nos avisando o que irá ser adicionado no banco de dados (ou se houve algum problema/não existem mudanças para serem realizadas).

“*python manage.py migrate*” que confirmará as mudanças realizadas.

Podemos então verificar as nossas tabelas abrindo o nosso banco de dados para visualização da maneira que achamos melhor.

imagem\_imagem (3 rows) ▼

SELECT \* FROM 'imagem\_imagem' LIMIT 0,30

Execute

id	imagem	descricao
1	imgs/AmeFingerBang.jpg	Ame abelha
2	imgs/AmePain.jpg	como eu estou agora
3	imgs/AmeDisappointed.jpg	brabo



## Tutorial 4: Persistência e Recuperação de Dados

Autor: Gabriel Alves da Silva

Será mostrado como persistir e recuperar dados através de um formulário de cadastro e uma página de listagem de usuários.

Crie o app para o tutorial 4:

```
C:\Users\Gabriel\Documents\teste>django-admin startapp tutorial4
```

Adicione o app do tutorial no INSTALLED\_APPS para que o Django saiba onde procurar seus models, comandos de controle, etc:

```
33 INSTALLED_APPS = [  
34     'django.contrib.admin',  
35     'django.contrib.auth',  
36     'django.contrib.contenttypes',  
37     'django.contrib.sessions',  
38     'django.contrib.messages',  
39     'django.contrib.staticfiles',  
40     'tutorial4.apps.Tutorial4Config'  
41 ]
```

Crie um Model para mapear os atributos de um usuário a uma tabela do banco de dados. Para criar um model no Django, você deve criar uma classe que herde de `models.Model` (não se esqueça de importar `models`!). Atributos “tipoField” indicam atributos que serão construídos no banco de dados. No nosso caso, todos os atributos são `CharField` e portanto armazenam caracteres:

```
1 from django.db import models  
2  
3 class Usuario(models.Model):  
4     nome_de_usuario = models.CharField(max_length=20)  
5     nome_completo = models.CharField(max_length=50)  
6     email = models.CharField(max_length = 30)  
7     senha = models.CharField(max_length= 20)
```

Crie um formulário para o cadastro de usuário. O Django fornece uma maneira conveniente de condensar os atributos de um formulário e essa maneira é mostrada abaixo. Deve-se criar uma classe que herde de `ModelForms`. Após isso, deve-se definir uma “classe meta”, ou seja uma classe dentro da classe formulário para setar o model e os campos do formulário:

```
1 from django import forms
2 from .models import Usuario
3
4 class FormulárioDeCadastro(forms.ModelForm):
5     class Meta:
6         model = Usuario
7         fields = ["nome_de_usuario", "nome_completo", "email", "senha",]
8         labels = {'nome_de_usuario': "Nome de Usuário", "nome_completo": "Nome Completo", "email": "Email", "senha": "Senha",}
```

Crie as views que vão ser responsáveis por receber as requisições e retornar respostas a elas. No nosso caso, teremos duas: uma view para cadastro e outra para listagem. A view de listagem simplesmente recupera todos os objetos salvos no banco de dados e envia eles para a página “listagem\_de\_usuarios.html” quando solicitada. Já a view de cadastro, salva os dados no formulário se o método da requisição for do tipo “POST”(o usuário tiver preenchido os dados e clicado em “submeter”). Caso contrário, simplesmente abre a página “cadastro\_de\_usuario.html”. Os templates(páginas HTML citadas) serão descritas mais a seguir :

```
1 from django.shortcuts import render
2 from .models import Usuario
3 from .formularios import FormulárioDeCadastro
4
5 # Create your views here.
6 def listagem_de_usuarios(request):
7     usuarios = Usuario.objects.all()
8     contexto = {"usuarios": usuarios}
9     return render(request, "listagem_de_usuarios.html", contexto)
10
11 def cadastro_de_usuario(request):
12     if request.method == "POST":
13         formulario_de_cadastro = FormulárioDeCadastro(request.POST)
14
15         if formulario_de_cadastro.is_valid():
16             formulario_de_cadastro.save()
17
18     else:
19         formulario_de_cadastro = FormulárioDeCadastro()
20
21     contexto = {"formulario_de_cadastro": formulario_de_cadastro}
22
23     return render(request, "cadastro_de_usuario.html", contexto)
```

Adicione paths para mapear as urls com as views. Para fins de organização, crie um arquivo urls no app tutorial4 e realize o mapeamento da maneira mostrada abaixo:

```
1 from django.urls import path
2 from . import views
3
4 urlpatterns = [
5     path("listagem_de_usuarios/", views.listagem_de_usuarios),
6     path("cadastro_de_usuario/", views.cadastro_de_usuario)
7 ]
```

Depois adicione uma inclusão do urls de tutorial4 na lista de urls root do projeto da maneira mostrada abaixo:

```
1  from django.contrib import admin
2  from django.urls import path, include
3
4  urlpatterns = [
5      path("tutorial4/", include("tutorial4.urls")),
6      path('admin/', admin.site.urls)
7  ]
```

Dessa maneira, quando o usuário acessar uma url que contenha “tutorial4/”, o django vai automaticamente buscar o resto dessa url no arquivo urls dentro do app tutorial4. Se após “tutorial4/” houver, por exemplo, “cadastro\_de\_usuario/”, o Django vai invocar a view cadastro\_de\_usuario.

Crie os templates que vão exibir as telas de cadastro e de listagem. No Django, injetamos código Python no HTML através de pares de chaves. O par “{% %}” define tags Django que são comandos específicos do framework. O par “{{ }}” permite acessar atributos.

Na tela de cadastro, chamamos a tag csrf\_token para proteção contra ataques Cross-site Request Forgery. Logo em seguida, acessamos o objeto formulario\_de\_cadastro, que é recebido pela view cadastro\_de\_usuario que vimos lá atrás, e esse objeto vai ser renderizado como uma série de divs.

```
1  <!DOCTYPE html>
2  <html>
3      <body>
4          <div class="cadastro">
5              <form method="POST">
6                  <legend>Cadastro</legend>
7                  {% csrf_token %}
8                  {{ formulario_de_cadastro.as_div }}
9                  <button type="submit" class="btn btn-primary">Submit</button>
10             </form>
11         </div>
12     </body>
13 </html>
```

Na tela de listagem, percorremos com um for a lista de usuários, recebida pela view `listagem_de_usuarios`, e recuperamos cada um de seus atributos dentro de parágrafos.

```
1  <!DOCTYPE html>
2  <html>
3      <body>
4          <div id="listagem_de_usuarios">
5              <h1>Listagem</h1>
6              <br>
7              {% for usuario in usuarios %}
8                  <p>{{usuario.nome_de_usuario}}</p>
9                  <p>{{usuario.nome_completo}}</p>
10                 <p>{{usuario.email}}</p>
11                 <p>{{usuario.senha}}</p>
12                 <br>
13             {% endfor %}
14         </div>
15     </body>
16 </html>
```

Crie uma migration para rastrear as mudanças no banco de dados(no nosso caso, a criação da tabela `Usuario`) e depois dê o comando de migrate para salvar as mudanças no banco de dados atual.

```
C:\Users\Gabriel\Documents\teste>python manage.py makemigrations
No changes detected

C:\Users\Gabriel\Documents\teste>python manage.py migrate
```

Inicie o servidor em uma máquina local. A porta utilizada no exemplo é a padrão(8000):

```
C:\Users\Gabriel\Documents\teste>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
June 14, 2023 - 22:03:34
Django version 4.2.1, using settings 'teste.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Acesse o formulário ([http://127.0.0.1:8000/tutorial4/cadastro\\_de\\_usuario.html](http://127.0.0.1:8000/tutorial4/cadastro_de_usuario.html)) e cadastre usuários.

Cadastrando o usuário galves:

Cadastro

Nome de Usuário:

Nome Completo:

Email:

Senha:

Cadastrando o usuário joao:

Cadastro

Nome de Usuário:

Nome Completo:

Email:

Senha:

Acesse a página de listagem, localizada no seguinte link: ([http://127.0.0.1:800/tutorial4/cadastro\\_de\\_usuario.html](http://127.0.0.1:800/tutorial4/cadastro_de_usuario.html)). Ela irá listar o nome de usuário, nome completo, email e senha dos usuários cadastrados no banco de dados.

## Listagem

galves

Gabriel Alves

[gabriel@gmail.com](mailto:gabriel@gmail.com)

12345678

joao

joao da silva

[joao@gmail.com](mailto:joao@gmail.com)

12345678

## Tutorial 5: Upload de Imagens

Autores: Ciro Olímpio de Melo e Enzo Eduardo Cassiano Ibiapina

Será mostrado como realizar e visualizar o upload de imagens por meio de formulários.

Primeiro vamos tratar da aplicação. Com um projeto inicial do Django pronto, crie um aplicativo para o serviço de imagem: `"django-admin startapp imagem"`.

Na pasta do app, o primeiro passo é editar o arquivo `model.py`, criando o modelo de "Imagem". O campo "imagem", que recebe a tipagem `ImageField`, é o campo que representa a imagem a ser salva no modelo, que nesse exemplo, será salva na pasta "imgs". (`ImageField` é parte da biblioteca Pillow, sendo necessária sua instalação pelo comando `pip install pillow`):

```
1 from django.db import models
2
3 # Create your models here.
4 class Imagem(models.Model):
5     imagem = models.ImageField(upload_to="imgs/", null=True, blank=True)
6     descricao = models.TextField(blank=True)
```

Depois de criado o modelo, o próximo passo é criar o seu formulário no arquivo `forms.py`:

```
1 from django import forms
2
3 from .models import Imagem
4
5 class ImagemForm(forms.ModelForm):
6     class Meta:
7         model = Imagem
8         fields = [
9             'imagem',
10            'descricao'
11        ]
```

Com o modelo e o seu respectivo formulário prontos, o próximo passo é criar as views relacionadas ao upload de imagem e a listagem de imagens.

Para o upload, é definido a função `upload_view`. Se a função receber um request com o método POST, ele salva na base de dados. Nesse exemplo, o formulário criado anteriormente é renderizado dentro de um contexto no template `upload.html`

Para a listagem, é definida a função `listagem_view`. A variável entradas recebe todos os objetos `Imagem` salvos na base de dados, e dentro de um contexto, esses objetos são renderizados no template `listagem.html`

Para exibir as duas páginas citadas anteriormente, é definido o `homepage_view`.

```

1 from django.shortcuts import render
2 from .models import Imagem
3 from .forms import ImagemForm
4
5 # Create your views here.
6 def upload_view(request, *args, **kwargs):
7     form = ImagemForm(request.POST, request.FILES)
8
9     if (request.method == "POST"):
10         if form.is_valid():
11             form.save()
12             form = ImagemForm()
13
14     context = {
15         'form': form
16     }
17
18     return render(request, "upload.html", context)
19
20 def listagem_view(request, *args, **kwargs):
21     entradas = Imagem.objects.all()
22     context = {
23         'object_list': entradas
24     }
25     return render(request, "listagem.html", context)
26
27 def homepage_view(request, *args, **kwargs):
28     return render(request, "home.html", {})

```

Após essas edições, edita-se o *settings.py* do projeto, adicionando o aplicativo a lista de apps instalados no projeto, bem como carregar variáveis relacionadas a como as imagens serão salvas dentro da máquina em que o projeto está. Nesse exemplo, o path completo das imagens é "media/imgs/". (O projeto também deve ter uma pasta "static").

```

32 # Application definition
33
34 INSTALLED_APPS = [
35     'django.contrib.admin',
36     'django.contrib.auth',
37     'django.contrib.contenttypes',
38     'django.contrib.sessions',
39     'django.contrib.messages',
40     'django.contrib.staticfiles',
41
42     # Aplicativos
43     'imagem.apps.ImagemConfig',
44 ]
45
46
119 # Static files (CSS, JavaScript, Images)
120 # https://docs.djangoproject.com/en/4.2/howto/static-files/
121
122 STATIC_URL = 'static/'
123 STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static/'),)
124 MEDIA_URL = 'media/'
125 MEDIA_ROOT = os.path.join(BASE_DIR, 'media/')
126

```

Após a edição do *settings.py*, deve-se editar também o *urls.py*, para que as imagens sejam salvas com o url correto por meio do static.

```

17 from django.contrib import admin
18 from django.urls import path
19 from django.conf import settings
20 from django.conf.urls.static import static
21
22 from imagem import views
23
24 urlpatterns = [
25     path('', views.homepage_view, name='home'),
26     path('upload/', views.upload_view),
27     path('listagem/', views.listagem_view),
28     path('admin/', admin.site.urls),
29 ]+static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)

```

Uma vez que tiver feito todos esses ajustes, execute o par de comandos para atualizar o banco de dados do projeto com o novo aplicativo de imagens: *“python manage.py makemigrations”*, seguido por *“python manage.py migrate”*. Com isso, a parte de upload de imagens está pronta.

Agora vamos tratar dos templates. Além de alguns ajustes já mostrados anteriormente que também abrangem os templates, também é necessário editar o arquivo *settings.py* com a pasta do projeto onde eles se localizam.

```

500 TEMPLATES = [
501     {
502         'BACKEND': 'django.template.backends.django.DjangoTemplates',
503         'DIRS': [BASE_DIR / "templates"],
504         'APP_DIRS': True,
505         'OPTIONS': {
506             'context_processors': [
507                 'django.template.context_processors.debug',
508                 'django.template.context_processors.request',
509                 'django.contrib.auth.context_processors.auth',
510                 'django.contrib.messages.context_processors.messages',
511             ],
512         },
513     },
514 ]

```

Feito isso, aqui é mostrado os templates feitos para esse exemplo.

Em *upload.html*, *form.as\_p* renderiza o formulário de upload criado anteriormente em tags HTML. O formulário é então enviado por meio de um request POST, que é tratado pela função *upload\_view* em *views.py*, e então salvo na base de dados.

```

1 <!doctype HTML>
2 <html>
3 <body>
4     <div id="UploadPage">
5         <h1>Upload</h1>
6         <form method="POST" enctype="multipart/form-data"> {% csrf_token %}
7             {{ form.as_p }}
8         </form>
9         <a href="/">Voltar</a>
10    </div>
11</body>
12</html>

```

Em *listagem.html*, é feita uma iteração sobre a lista de objetos Imagem salvos na base de dados, que foram passados como contexto pela função *listagem\_view* em *views.py*. O resultado mostra cada imagem e sua descrição na página.



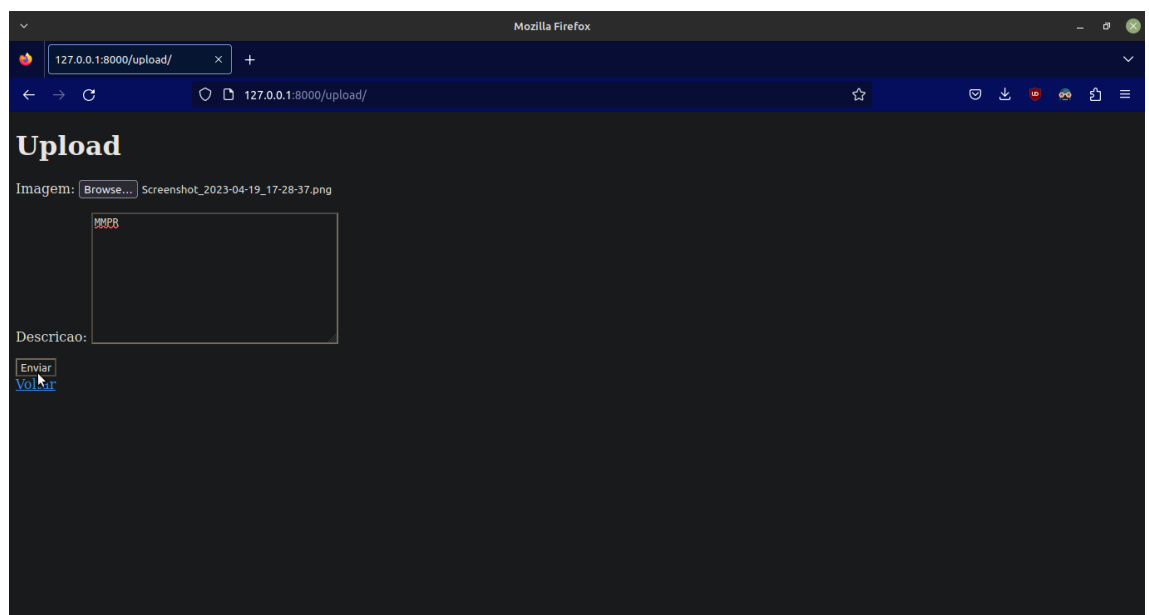
```
1 <!doctype HTML>
2 <html>
3 <body>
4     <div id="ListagemPage">
5         <h1>Listagem</h1>
6
7         {% for entrada in object_list %}
8
9             <img src='{ {{ entrada.imagem.url }} }'></img>
10            <p>{{ entrada.descricao }}</p>
11
12        {% endfor %}
13
14        <a href="/">Voltar</a>
15    </div>
16 </body>
17 </html>
```

Em *home.html*, teremos a página inicial que será capaz de acessar as páginas de upload e de listagem de imagens.

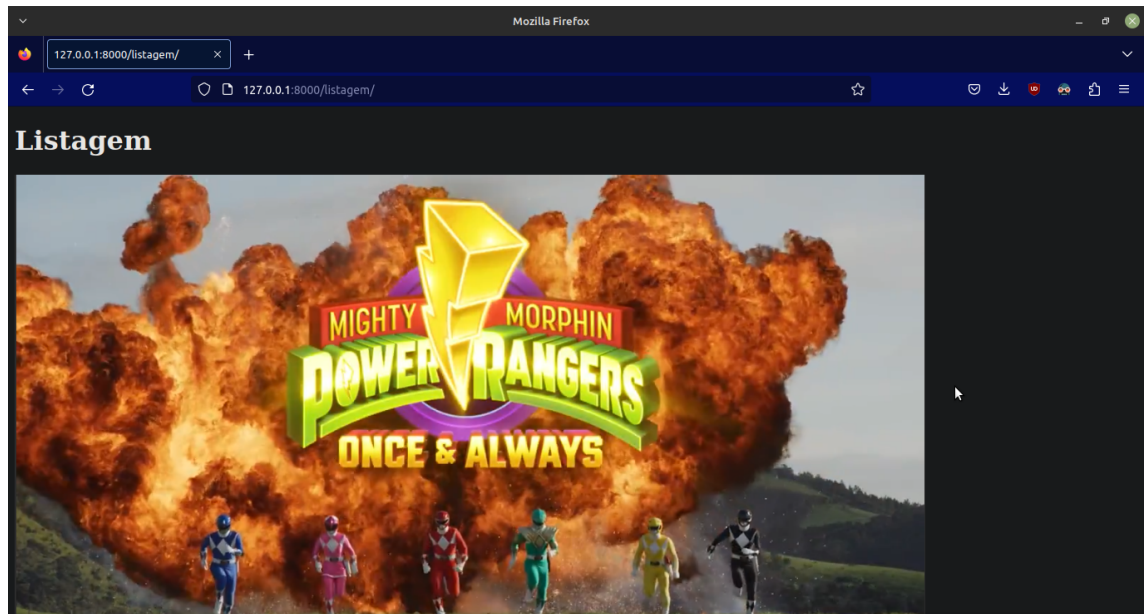
```
1 <!doctype HTML>
2 <html>
3 <body>
4     <div id="Homepage">
5         <h1>Home</h1>
6         <a href="/upload">Upload</a>
7         <a href="/listagem">Listagem</a>
8     </div>
9 </body>
10 </html>
```

Com tudo isso feito, finalmente podemos ver o projeto em ação:

Realizando um upload:



Listando imagens:



## Tutorial 6: Instalação e uso da ferramenta de testes Selenium

Autores: Ciro Olímpio de Melo e Enzo Eduardo Cassiano Ibiapina

### No Windows:

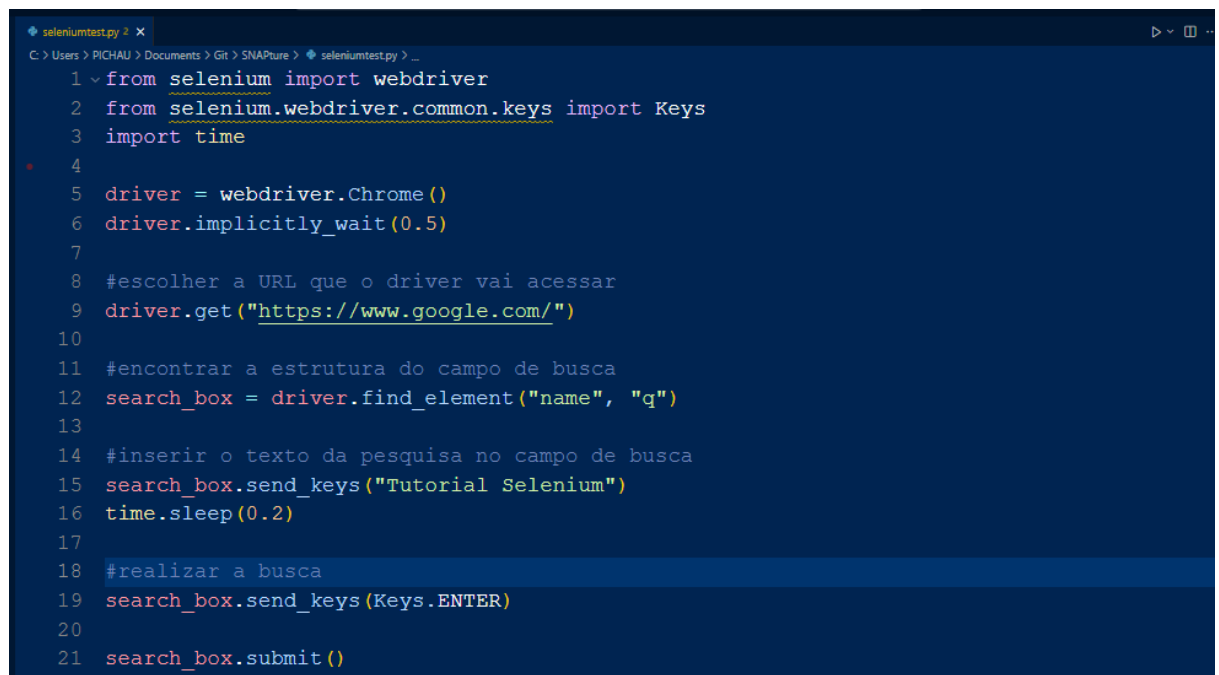
(Windows 10 / Google Chrome)

Para instalar o selenium, precisamos ativar o nosso ambiente virtual (já criado em tutoriais anteriores). Executamos o comando *"maquinavirtual\Scripts\activate"* no cmd (partindo do diretório onde está localizada a máquina virtual), substituindo maquinavirtual pelo nome da máquina criada anteriormente.

Uma vez com a máquina virtual ativada, devemos executar o comando *"pip install selenium"* realizarmos a instalação do serviço.

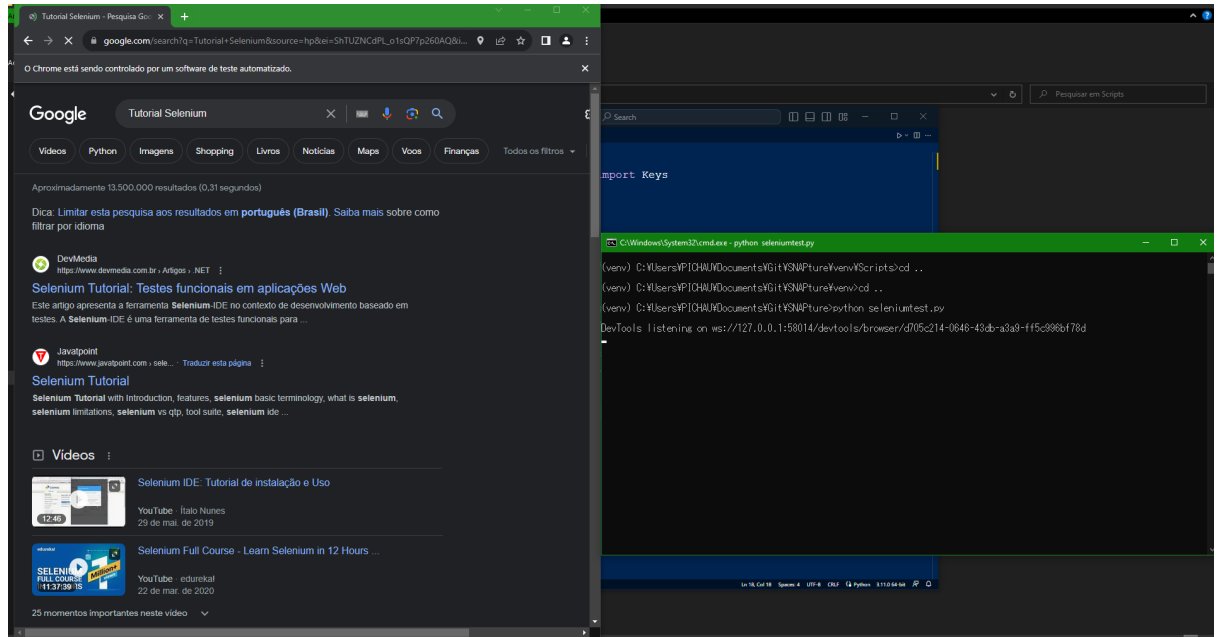
Para possibilitar os testes, o computador deve ter também um driver específico para o navegador por onde eles serão realizados. Nesse caso, utilizaremos o chromedriver para conseguir realizar os testes pelo google chrome. É necessário visitar o endereço <https://chromedriver.chromium.org/downloads> e escolher a versão compatível com o nosso browser. Uma vez baixado, devemos colocar o nosso chromedriver na pasta onde estará o arquivo do teste.

Para nos assegurarmos de que o selenium foi instalado e consegue efetivamente interagir com nosso browser, montamos um código teste na nossa IDE padrão (VSCode) para ver se ele consegue abrir o browser, acessar uma página, inserir um input e realizar uma busca. O código é algo bem simples(temporização adicionada por razões de confiabilidade).



```
seleniumtest.py x
C:\Users\PICHAU>Documents>Git>SNApture>seleniumtest.py>_
1 from selenium import webdriver
2 from selenium.webdriver.common.keys import Keys
3 import time
4
5 driver = webdriver.Chrome()
6 driver.implicitly_wait(0.5)
7
8 #escolher a URL que o driver vai acessar
9 driver.get("https://www.google.com/")
10
11 #encontrar a estrutura do campo de busca
12 search_box = driver.find_element("name", "q")
13
14 #inserir o texto da pesquisa no campo de busca
15 search_box.send_keys("Tutorial Selenium")
16 time.sleep(0.2)
17
18 #realizar a busca
19 search_box.send_keys(Keys.ENTER)
20
21 search_box.submit()
```

Utilizando o cmd do nosso venv inicializamos a execução. Com a execução do código, vemos que o Selenium consegue abrir o nosso browser na página especificada e realizar os comandos especificados no código.

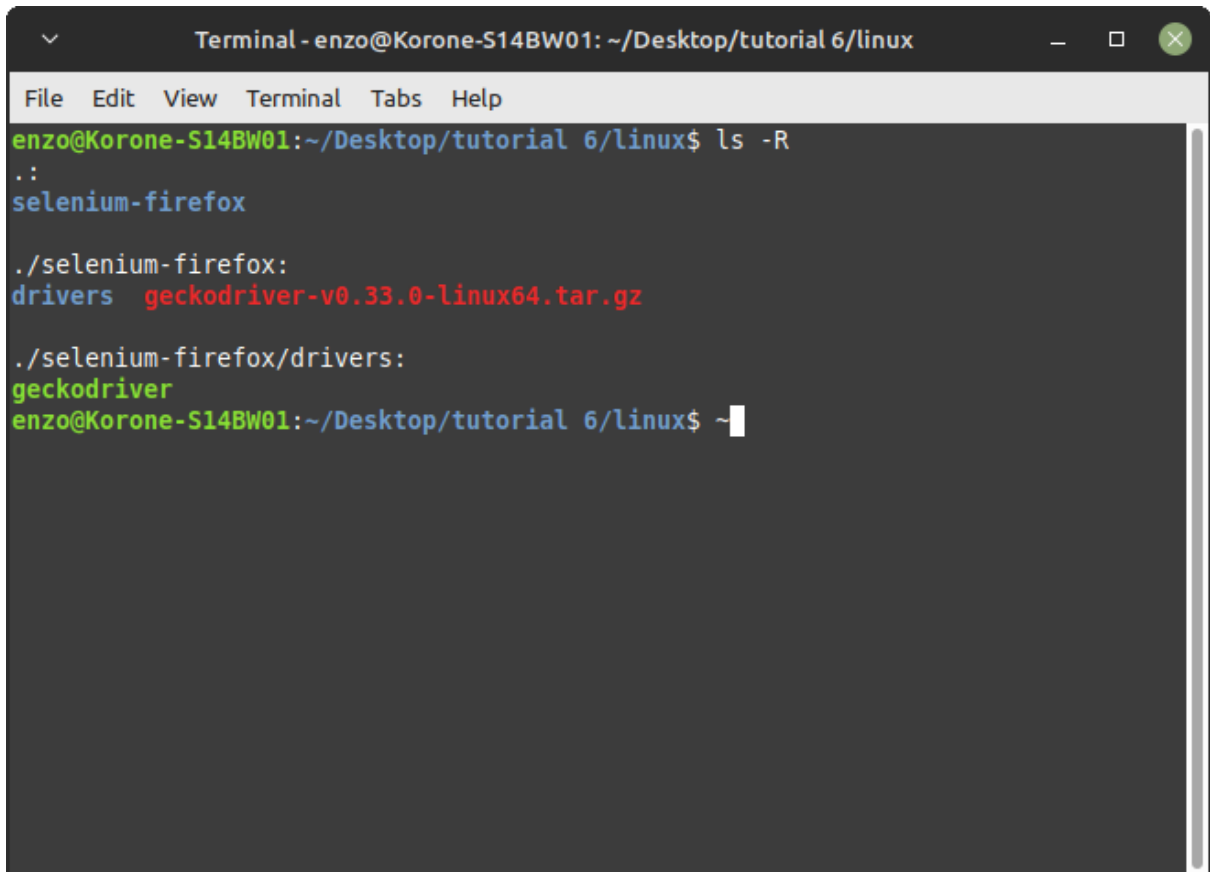


## No Linux: (Distribuições Ubuntu-based / Firefox)

Para o teste, foi criado uma pasta *selenium-firefox*, que será tratada como a pasta do projeto. Dentro da pasta do projeto, crie uma pasta *drivers* (o conteúdo da pasta será tratado posteriormente). Somado a isso, crie um ambiente virtual com “*python3 -m venv nomeDaPasta*”, e entre no ambiente virtual com o comando “*source nomeDaPasta/bin/activate*”.

Para instalar o Selenium de facto, dentro do ambiente virtual, execute o comando “*pip3 install selenium*”.

Para que o Selenium realize os testes com o navegador, a ferramenta precisa ter o controle sob o mesmo, e isso se dá a partir de um driver. O driver sugerido para o Firefox é o Gecko Web Driver (<https://github.com/mozilla/geckodriver>). Baixe a release mais recente do driver direto do repositório oficial no github, ou utilize o comando “*wget https://github.com/mozilla/geckodriver/releases/download/VersaoMaisRecente/geckodriver-VersaoMaisRecente-linux64.tar.gz*” (versão 64 bits) para baixar o arquivo pelo terminal. Após baixar o arquivo compactado, extraia-o e passe o resultado para a pasta *drivers* com o comando “*tar -xvz geckodriver-VersaoMaisRecente-linux64.tar.gz -C drivers*”.



```
Terminal - enzo@Korone-S14BW01: ~/Desktop/tutorial 6/linux
File Edit View Terminal Tabs Help
enzo@Korone-S14BW01:~/Desktop/tutorial 6/linux$ ls -R
.:
selenium-firefox

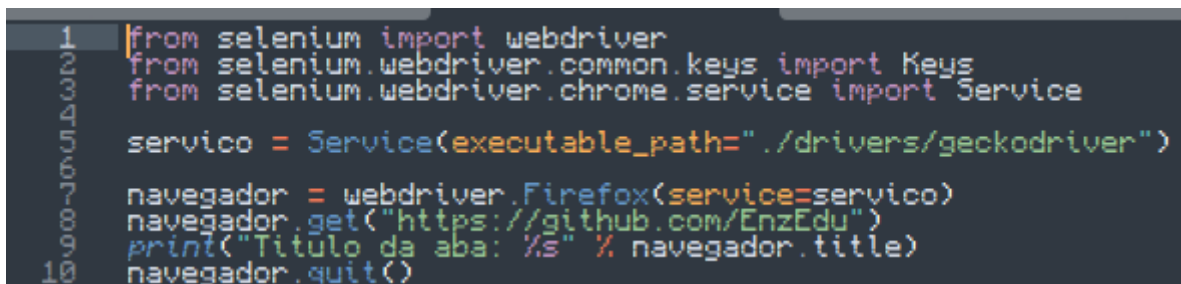
./selenium-firefox:
drivers  geckodriver-v0.33.0-linux64.tar.gz

./selenium-firefox/drivers:
geckodriver
enzo@Korone-S14BW01:~/Desktop/tutorial 6/linux$ ~
```

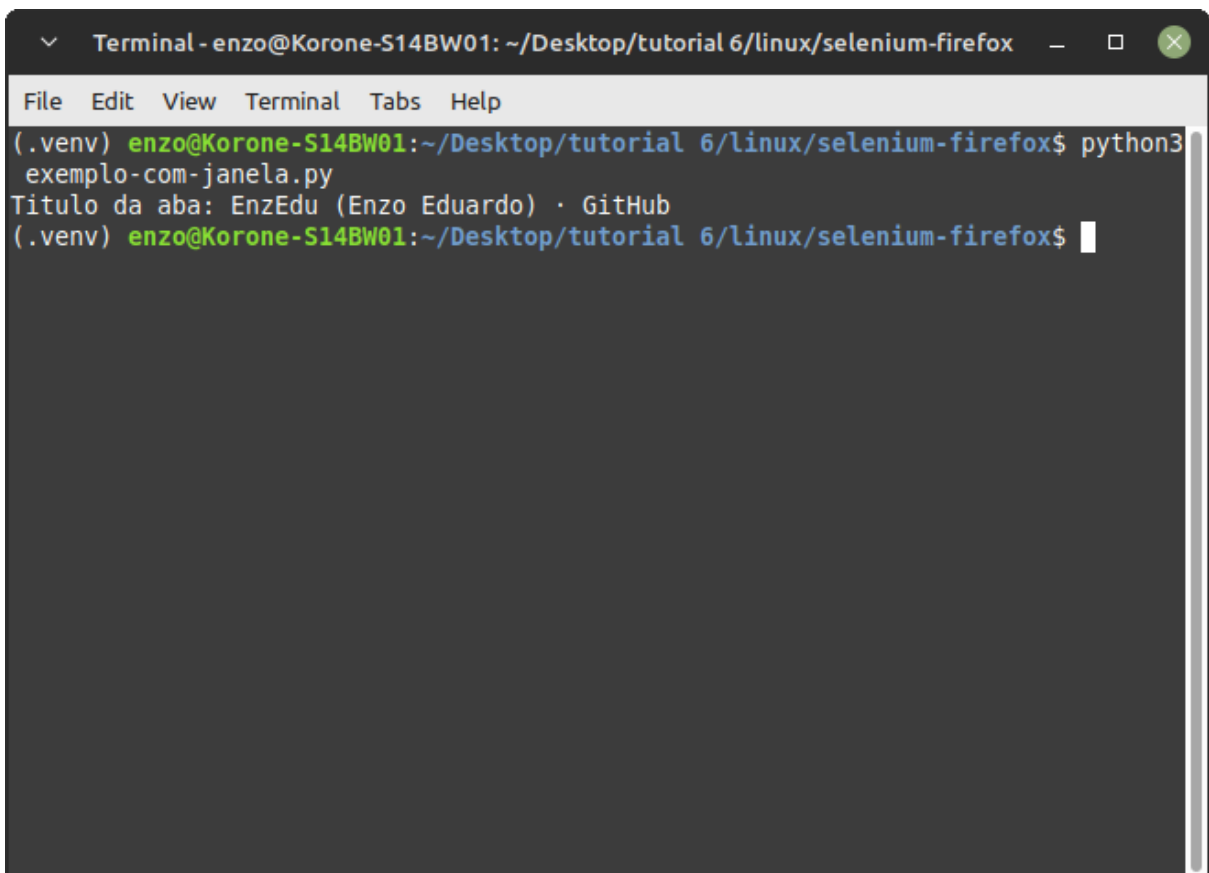
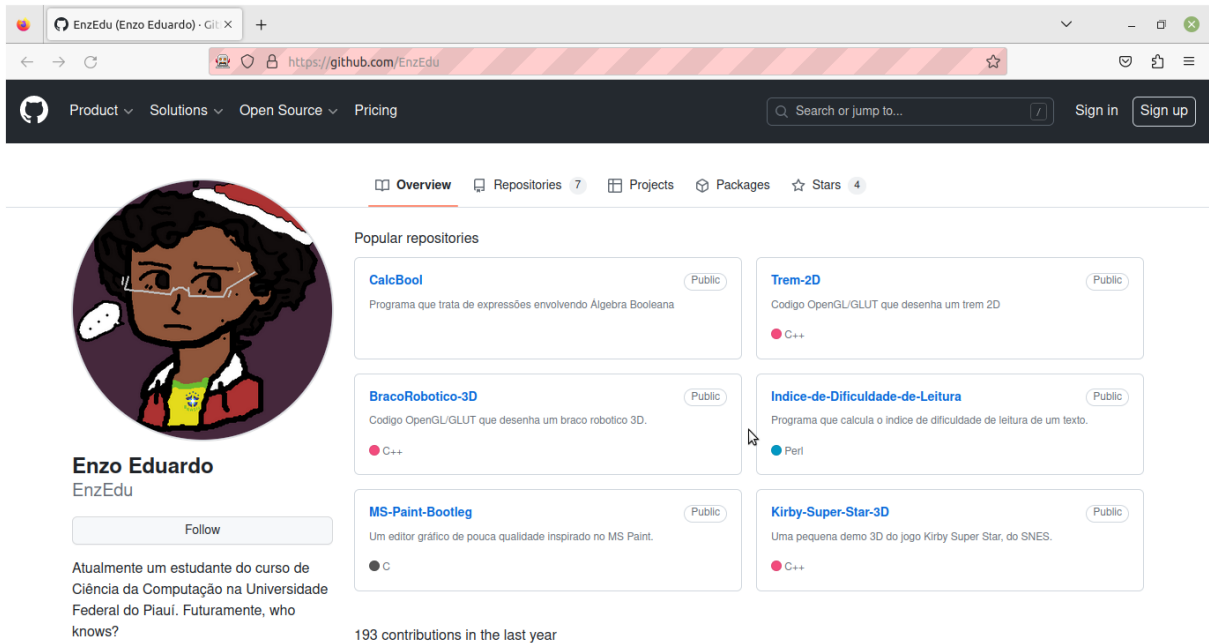
Com isso, o ambiente para o Selenium está pronto.

Foram criados dois arquivos python de teste: um em que o Selenium abre uma nova janela para o navegador, e outro em que o Selenium não abre uma janela. Para a execução de qualquer um dos testes, basta executar o comando `python3 nomeDoArquivoTeste.py`.

Para o primeiro arquivo teste, criou-se um objeto de classe Service, passando como argumento o arquivo binário do driver do respectivo navegador. Após isso, é instanciado o Navegador, por meio do módulo webdriver, tipo do navegador e o objeto Service passado como argumento. Utilize o método `.get()` da instância do Navegador para acessar uma página web por meio da url passada como parâmetro no método. Ao final do teste, feche o objeto Navegador.

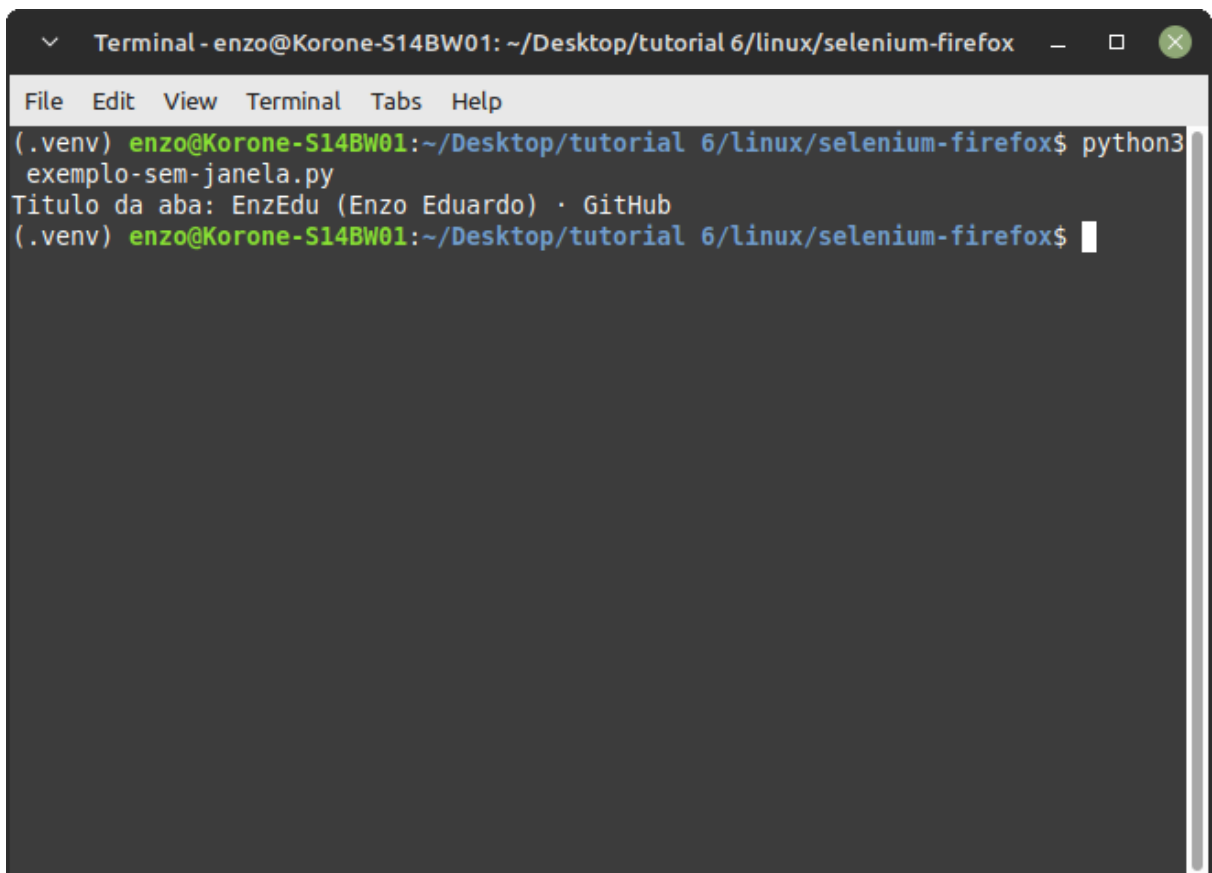


```
1 from selenium import webdriver
2 from selenium.webdriver.common.keys import Keys
3 from selenium.webdriver.chrome.service import Service
4
5 servico = Service(executable_path="./drivers/geckodriver")
6
7 navegador = webdriver.Firefox(service=servico)
8 navegador.get("https://github.com/EnzEdu")
9 print("Titulo da aba: %s" % navegador.title)
10 navegador.quit()
```



Para o segundo arquivo teste, para conseguir a execução do Selenium sem a abertura de uma janela do browser, a mudança realizada é a instanciação de uma classe Options, com o argumento `-headless`. Esse objeto é então utilizado na instanciação do Navegador, junto do Service com o path do driver do navegador.

```
1 from selenium import webdriver
2 from selenium.webdriver.common.keys import Keys
3 from selenium.webdriver.chrome.service import Service
4 from selenium.webdriver.firefox.options import Options
5
6 service = Service(executable_path="./drivers/geckodriver")
7
8 opcoes = Options()
9 opcoes.add_argument("-headless")
10
11 navegador = webdriver.Firefox(service=service, options=opcoes)
12 navegador.get("https://github.com/EnzEdu")
13 print("Titulo da aba: %s" % navegador.title)
14 navegador.quit()
15
```



The screenshot shows a terminal window titled "Terminal - enzo@Korone-S14BW01: ~/Desktop/tutorial 6/linux/selenium-firefox". The terminal displays the output of running a Python script. The script imports Selenium modules, sets up a headless Firefox service, navigates to "https://github.com/EnzEdu", prints the page title "Titulo da aba: EnzEdu (Enzo Eduardo) · GitHub", and then quits the browser. The terminal output matches the script's execution.

```
Terminal - enzo@Korone-S14BW01: ~/Desktop/tutorial 6/linux/selenium-firefox
File Edit View Terminal Tabs Help
(.venv) enzo@Korone-S14BW01:~/Desktop/tutorial 6/linux/selenium-firefox$ python3
exemplo-sem-janela.py
Titulo da aba: EnzEdu (Enzo Eduardo) · GitHub
(.venv) enzo@Korone-S14BW01:~/Desktop/tutorial 6/linux/selenium-firefox$
```