Selenium - O que você deveria saber - Parte 1

// Tags selenium python selenium-serie

Esse é o primeiro post da série sobre Selenium, pretendo cobrir desde o básico até algumas coisas mais legais :)

Introdução

Selenium é um ótimo framework para realizar diversos tipos de tarefas com o browser.

Nesta série, vou tentar compartilhar com vocês o que acredito ser necessário para o bom uso da ferramenta.

Veja como está organizada a série:

Parte I

- Instalação
- Abrindo uma página
- Manipulando elementos

Parte 2

- Brincando com formulários
- Trabalhando com múltiplas janelas
- Trabalhando com frames
- E se eu quiser esperar?!

Parte 3

- Executando código javascript
- Como utilizar diferentes navegadores

Parte 4

- Expected conditions
- ActionsChains - Operações avançadas
- EventListener - Ouvindo seu código

Instalação

Para instalar o Selenium não existe nenhum segredo, basta executar:

```
pip install selenium
```

Abrindo uma página

```
from selenium import webdriver
```

```
firefox = webdriver.Firefox()
firefox.get('http://google.com.br')
```

Vamos entender o que está acontecendo aqui.

Primeiro eu importo o `webdriver` que é o módulo que provê implementações para diferentes browsers.

```
from selenium import webdriver
```

Nesse caso utilizaremos o "Mozila Firefox", pois não precisa de nenhuma configuração adicional, basta que ele esteja instalado.

Então criamos uma instância chamada `firefox` e depois invocamos o metódo `get` passando como parâmetro a URL da página que desejamos abrir.

```
firefox = webdriver.Firefox()
firefox.get('http://google.com.br/')
```

Outros exemplos:

```
# Abrir o site da Python Brasil
firefox.get('http://python.org.br/')

# Abrir o site da Python MG
firefox.get('http://pythonmg.com.br/')
```

Manipulando elementos

Sempre existe a necessidade de manipularmos algum elemento da página, para isso você precisa saber como encontrá-lo.

Conhecimento em HTML é necessário para facilitar a manipulação da página

Se precisarmos encontrar um elemento pelo id, invocamos o metódo `find_element_by_id`:

```
# Se o elemento não for encontrado uma exception é gerada
find_element_by_id('<id>')
```

Se precisarmos encontrar todos os elementos que possuem uma classe específica, invocamos o metódo `find_elements_by_class_name`.

```
# Retornam vários elementos ou uma lista vazia
find_elements_by_class_name('<class_name>')
```

Existem diversos metódos disponíveis, abaixo estão os que mais utilizo:

```
# Encontrar elemento pelo ID
find_element_by_id('<id>')

# Encontrar elemento pelo atributo name
find_element_by_name('<name>')

# Encontrar elemento pelo texto do link
find_element_by_link_text('<text>')

# Encontrar elemento pelo seu seletor css
```

```
find_element_by_css_selector('<css_selector>')

# Encontrar elementos pelo nome da tag
find_elements_by_tag_name('<tag_name>')

# Encontrar elementos pela classe
find_elements_by_class_name('<class_name>')
```

Para visualizar todos os metódos, veja a documentação.

Exemplo para estudo

Let's code

Premissas

No Python Club os `posts` estão localizados dentro de uma `div`.

```
<div class="posts">
  <section class="post">[...]</section>
  <section class="post">[...]</section>
  <section class="post">[...]</section>
</div>
```

E cada `post` está dentro de uma `section` que possui a `class="post"` .

```
<section class="post">
  <header class="post-header">
    [...]
    <h3>
      <a class="post-title" href="<post_url>"><post_title></a>
    </h3>
    [...]
  </header>
</section>
```

Objetivo

Queremos que seja mostrado o título de cada `post` e seu `link`.

Execute o código abaixo e veja o resultado.

```
from selenium import webdriver

# Criar instância do navegador
firefox = webdriver.Firefox()

# Abrir a página do Python Club
firefox.get('http://pythonclub.com.br/')

# Seleciono todos os elementos que possuem a class post
posts = firefox.find_elements_by_class_name('post')

# Para cada post printar as informações
for post in posts:

    # O elemento `a` com a class `post-title`
    # contém todas as informações que queremos mostrar
    post_title = post.find_element_by_class_name('post-title')
```

```
    # `get_attribute` serve para extrair qualquer atributo do elemento
    post_link = post_title.get_attribute('href')

    # printar informações
    print u"Títutlo: {titulo}, \nLink: {link}".format(
      titulo=post_title.text,
      link=post_link
  )

# Fechar navegador
firefox.quit()
```

Código-fonte do exemplo: pythonclub.py.

Desafios

Desafio 1

Modificar o exemplo para mostrar o nome do autor do `post`.

Desafio 2

Modificar o exemplo 01 para salvar os dados(titulo, link, autor) em um arquivo `json`.

Gostou? Leia a segunda parte.

Qualquer dúvida pode enviar um e-mail contato@lucasmagnum.com.br ficarei feliz em ajudar =)

Selenium - O que você deveria saber - Parte 2

// Tags selenium python selenium-serie

Esse é o segundo post da série sobre Selenium, hoje vamos aprender a manipular formulários, frames e múltiplas janelas. Vamos também vai descobrir que é possível esperar para tentar encontrar um elemento na página.

Se você não leu a primeira parte, clique aqui.

Parte 2

Brincando com formulários

Quantas vezes você já não preencheu um formulário na web?

Hoje vamos aprender como fazer isso, vamos visualizar o exemplo abaixo que procura por um termo no google.

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

firefox = webdriver.Firefox()
firefox.get('http://google.com.br/')

# pegar o campo de busca onde podemos digitar algum termo
campo_busca = firefox.find_element_by_name('q')

# Digitar "Python Club" no campo de busca
campo_busca.send_keys('Python Club')

# Simular que o enter seja precisonado
campo_busca.send_keys(Keys.ENTER)
```

Foi bem simples, encontramos o `campo_busca` e invocamos o metódo `send_keys` com o texto que desejamos digitar e depois simulamos o pressionamento do botão "Enter".

Nota: Sempre que houver a necessidade de utilizar uma tecla especial podemos encontrá-la na classe `Keys`.

E se for um campo `select`, como eu faço?

Digamos que exista o seguinte código:

```
<select name="estados">
  <option value="mg">MG</option>
  <option value="rj">RJ</option>
  <option value="sp">SP</option>
</select>
```

Existe uma classe só para facilitar o trabalho com `selects`, ela possui metódos para selecionar uma opção pelo seu texto ou valor, entre outras.

```
# Importar a classe Select
>> from selenium.webdriver.support.ui import Select

# É preciso passar o elemento para a classe
>> estados = Select(firefox.find_element_by_name('estados'))

# Selecionar a opção MG
>> estados.select_by_visible_text('MG')

# Selecionar a opção SP
>> estados.select_by_value('sp')
```

Agora você já sabe o básico para manipular um formulário, você pode digitar nos campos, selecionar valores, clicar em botões.

Trabalhando com múltiplas janelas

E quando se abre uma nova janela? ou um alerta? O que fazemos?

```
>> firefox.window_handles
[u'{7fd12d82-4fb3-48a4-a8b9-e1e460c00236}']
```

A instância `firefox` possui um atributo chamado `window_handles`, que é uma lista contendo um ID para cada janela aberta.

Nota: O ID é criado para nova janela aberta e não para uma ABA ou um alerta javascript.

Quando você abrir uma nova janela, poderá perceber que é criado automaticamente um novo ID.

```
>> firefox.window_handles
[u'{7fd12d82-4fb3-48a4-a8b9-e1e460c00236}', u'{2ce4de19-0902-48e3-a1cc-
50f6378afd79}']
```

Para alternar entre janelas, basta chamar o metódo `switch_to_window` passando o ID da janela como parâmetro.

Imagine que temos uma janela aberta na página do Google e a outra na página da Python Club.

```
>> firefox.window_handles
[u'{7fd12d82-4fb3-48a4-a8b9-e1e460c00236}', u'{2ce4de19-0902-48e3-a1cc-
50f6378afd79}']

# Título da janela atual
>> firefox.title
u'Google'

# Trocar para a última janela da lista
>> firefox.switch_to_window_handles(firefox.window_handles[-1])
>> firefox.title
u'PythonClub //'

# Fechar a janela atual
>> firefox.close()

# Voltar para a janela do Google
>> firefox.switch_to_window_handles(firefox.window_handles[-1])
>> firefox.title
u'Google'
```

E como saber o ID da janela atual? Simples!

```
>> firefox.current_window_handle
u'{7fd12d82-4fb3-48a4-a8b9-e1e460c00236}'
```

Fácil não?!

E se abrir um alerta `javascript`, como fazemos???

Existe uma função para tratar alertas `javascript`, a função `switch_to_alert` irá permitir que manipule eles sem problemas.

```
>> alerta = firefox.switch_to_alert()
>> alerta.  # Tab para autocomplete
alerta.accept     alerta.dismiss    alerta.driver     alerta.send_keys
alerta.text

# Se for um `confirm`, você pode aceitar ou cancelar.
>> alerta.accept()  # Aceitar, ou clicar em "OK"
>> alerta.dismiss() # Cancelar, ou clicar em "Cancel"/"Cancelar"

# Se for uma caixa de texto e você quiser digitar algo
>> alerta.send_keys('Digitar esse texto')

# Se você quiser visualizar o texto que está presente no alerta
>> alerta.text
u'Texto do alerta'
```

Trabalhando com frames

Não existe muita diferença entre manipulação de frames e janelas, o princípio é o mesmo.

- Encontrar o elemento (metódo `find_element`)
- Mudar para ele (metódo `switch_to_frame`)
- Realizar as ações

Conceitos

Por padrão o frame principal ou aquele que está disponível quando você abre uma página é denominado `default_content`.

Se algum elemento estiver dentro de um frame você não irá localizá-lo sem "entrar" neste frame.

Se você estiver dentro de um frame e o elemento estiver no `default_content` você não conseguirá localizá-lo sem voltar para o frame principal.

Exemplo:

Precisamos clicar em um botão que está dentro de um frame.

Premissas: O iframe onde estão os botões que precisamos manipular possui o ID `buttons`

```
# Vamos tentar localizar o botão que está dentro do frame e será gerado uma
Exception
>> firefox.find_element_by_id('<botao_id>')
NoSuchElementException: Message: u'Unable to locate element:
{"method":"id","selector":"<botao_id>"}'

# Precisamos antes encontrar o frame
>> frame = firefox.find_element_by_id('buttons')

# Vamos alterar para ele
>> firefox.switch_to_frame(frame)

# Agora podemos encontrar o elemento
>> botao = firefox.find_element_by_id('<botao_id>')

# E podemos manipulá-lo :)
>> botao.click()
```

E se eu quiser esperar?!

Quando você tenta localizar um elemento, o Selenium irá consultar a página e se não encontrar será gerado uma `exception` de imediato.

Mas e se o elemento demorar um pouco para aparecer, pode ser que ele faça parte de uma animação, um consulta `ajax` ou qualquer coisa do tipo.

Então precisamos dizer ao Selenium para **esperar**.

Existe uma classe chamada `WebDriverWait` que pode ser utilizada para para esperar por determinadas ações.

Hoje será apresentado o básico sobre ela e voltaremos a falar sobre na Parte 4 deste tutorial.

Veja o exemplo abaixo:

```
>> elemento = firefox.find_element_by_id('<elemento_id>')
```

Nesse caso se o elemento não existe, será gerado uma `exception`.

Mas e se soubermos que ele pode demorar um tempo para aparecer?

```
# Importar a classe WebDriverWait
>> from selenium.webdriver.support.ui import WebDriverWait

def esperar_pelo_elemento(firefox):
  return firefox.find_element_by_id('<elemento_id>')

>> elemento = WebDriverWait(firefox, 5).until(esperar_pelo_elemento)
```

O que fizemos? Nós pedimos para o `firefox` esperar por `5` segundos até que o resultado da função `esperar_pelo_elemento` seja `True`. Caso passe esse tempo e ele não encontre o elemento, então será gerada uma `exception`.

Essa foi nossa introdução ao `WebDriverWait`, basicamente você precisa passar uma função que aceite como parâmetro a instância do navegador e lá executar o código para encontrar o elemento.

Por hoje é só! Nos vemos na próxima, espero que tenha aprendido algo hoje :)

Selenium - O que você deveria saber - Parte 3

// Tags selenium python selenium-serie

Esse é o terceiro post da série sobre Selenium, hoje vamos aprender a executar código javascript e usar diferentes navegadores.

- Veja a primeira parte.
- Veja a segunda parte.

Parte 3

Executando código javascript

Algumas vezes é necessário executar algum código `javascript`, seja para adiantar a execução de uma função ou até mesmo para manipular um elemento.

Vamos ao nosso exemplo:

```
from selenium import webdriver

firefox = webdriver.Firefox()
firefox.get('http://google.com.br/')
```

```
firefox.execute_script('alert("código javascript sendo executado")')
```

```
firefox.execute_async_script('alert("código javascript sendo executado")')
```

O Selenium permite que você faça isso através de uma instância do navegador, chamando os metódos `execute_script` e `execute_async_script`.

A diferença entre os dois é que o primeiro (`execute_script`) irá esperar até ter a resposta do navegador e o outro não.

Você pode executar qualquer código javascript e isso pode ser muito útil!

Como utilizar diferentes navegadores

Para utilizar navegadores diferentes é bem simples, vamos ver como configurar 2 navegadores diferentes.

Firefox

O *Firefox* é o mais simples de ser configurado, você não precisa passar nenhum parâmetro adicional.

```
from selenium import webdriver
```

```
firefox = webdriver.Firefox()
```

Se a instalação do firefox tiver sido alterada e feito em alguma pasta diferente da padrão, você pode informar o caminho para o executável.

```
from selenium import webdriver
```

```
firefox = webdriver.Firefox(firefox_binary='/bin/firefox')
```

```
# se estiver usando o windows, basta informar o caminho completo
firefox = webdriver.Firefox(firefox_binary='C:/firefox/firefox.exe')
```

Chrome

Para utilizar *Chrome* você precisa ter instalado o chrome no seu computador, você pode fazer isso pelo terminal:

```
apt-get install chromium-browser
```

Após instalar o navegador você precisa realizar o download do `chromedriver` que é um intermediário entre o Selenium o e Chrome.

Por default o Selenium procura pelo `chromedriver` na mesma pasta de onde está sendo executado.

Faça o download da ultima versão do chromedriver, coloque em um local de sua preferência e passe o caminho completo na hora de iniciar o navegador.

```
from selenium import webdriver
```

```
chrome = webdriver.Chrome(executable_path='<caminho para chromedriver>')

# exemplo
chrome =
webdriver.Chrome(executable_path='/home/lucasmagnum/downloads/chromedriver')
```

Você pode visualizar todos os navegadores suportados pelo Selenium

Por hoje é só! Nos vemos na próxima, espero que tenha aprendido algo hoje :)

Selenium - O que você deveria saber - Parte 4

// Tags selenium python selenium-serie

Esse é o quarto post da série sobre Selenium, hoje você irá aprender a fazer algumas coisas mais interessantes!

- Veja a primeira parte.
- Veja a segunda parte.
- Veja a terceira parte.

Esse post será o mais longo, então prepare-se!

Parte 4

- Expected conditions
- ActionsChains - Operações avançadas
- EventListener - Ouvindo seu código

Expected conditions

Em alguns casos você precisa esperar para manipular o elemento até que uma condição se satisfaça, para isso o Selenium possui um conjunto de **funções** para facilitar na maioria das situações.

Essas condições são chamadas `expected conditions`, abaixo a lista das principais condições:

`title_is`
: Checa se o título (title) da página corresponde ao informado (comparação exata). Retorna True ou False.

`title_contains`
: Checa se o título contém a string informada (case-sensitive). Retorna True ou False.

`presence_of_element_located`
: Checa se o elemento está presente no DOM (ele não precisa estar visível). Retorna o elemento se ele estiver presente ou False.

`visibility_of_element_located`
: Checa se o elemento está presente no DOM e visível (ele precisa ter width e height maior que 0). Retorna o elemento se ele estiver visível ou False.

`text_to_be_present_in_element`
: Checa se o texto está presente no elemento. Retorna True ou False.

`text_to_be_present_in_element_value`
: Checa se o texto está presente no atributo "value" do elemento. Retorna True ou False.

`element_to_be_clickable`
: Checa se o elemento está visível e disponível para ser clicado. Retorna o elemento ou False.

`alert_is_present`
: Checa se existe algum alerta na página. Retorna o alerta ou False.

Todas as funções estão no arquivo: selenium/webdriver/support/expected_conditions.py

Você pode utilizar essas funções em conjunto com o que foi aprendido na Parte 2

Exemplos de uso:

```
# Importar classe para inicializar o browser
from selenium import webdriver
# Importar a classe WebDriverWait
from selenium.webdriver.support.ui import WebDriverWait
# Importar a classe que contém as funções e aplicar um alias
from selenium.webdriver.support import expected_conditions as EC
# Importar classe para ajudar a localizar os elementos
from selenium.webdriver.common.by import By

firefox = webdriver.Firefox()
# Instanciar a classe que irá esperar até 5 segundos
wait = WebDriverWait(firefox, 5)

# Aguardar até que a página tenha o título "PythonClub"
wait.until(EC.title_is("PythonClub"))

# Aguardar página contenha o título "PythonClub"
wait.until(EC.title_contains("PythonClub"))

# Aguardar até que o elemento "id_elemento" esteja presente no DOM
elemento = wait.until(EC.presence_of_element_located((By.ID, 'id_elemento')))

# Aguardar até que o elemento "id_elemento" esteja visível
elemento = wait.until(EC.visibility_of_element_located((By.ID, 'id_elemento'))))

# Aguardar até que o elemento "id_elemento" contenha o texto "Python"
wait.until(EC.text_to_be_present_in_element((By.ID, 'id_elemento'), 'Python'))

# Aguardar até que o elemento "id_elemento" contenha o valor "Python"
wait.until(EC.text_to_be_present_in_element_value((By.ID, 'id_elemento'),
'Python')))

# Aguardar até que o elemento "id_elemento" possa ser clicado
elemento = wait.until(EC.element_to_be_clickable((By.ID, 'id_elemento')))

# Aguardar até que um alerta esteja presente na página
wait.until(EC.alert_is_present)
```

Após os 5 segundos, se a condição retornar `False` será gerado uma exception `TimeoutException`.

Use a abuse das `conditions`!

ActionsChains - Operações avançadas

`click_and_hold(on_element=None)`
: Clica no elemento e mantém o botão esquerdo do mouse pressionado. Se o parâmetro `on_element` não for informado, será realizado na posição atual do mouse.

`release(on_element=None)`
: Despressiona o botão do mouse. Se o parâmetro `on_element` for informado, será realizado no elemento.

`context_click(on_element=None)`
: Clica no elemento com o botão direito do mouse. Se o parâmetro `on_element` não for informado, será realizado na posição atual do mouse.

`double_click(on_element=None)`
: Realiza o clique duplo. Se o parâmetro `on_element` não for informado, será realizado na posição atual do mouse.

`key_down(value, element=None)`
: Simula uma tecla sendo pressionada e mantida assim. (Deve ser utilizada somente com as teclas de modificação **Control**, **Alt** e **Shift**). Se o parâmetro `element` não for informado, será realizado no elemento atual.

`key_up(value, element=None)`
: Simula uma tecla sendo despressionada. (Deve ser utilizada somente com as teclas de modificação **Control**, **Alt** e **Shift**). Se o parâmetro `element` não for informado, será realizado no elemento atual.

As ações são armazenadas na ordem em que foram invocadas e para que sejam realizadas é preciso chamar o metódo `perform` da instância.

Exemplos de uso:

```
# Importar classe para inicializar o browser
from selenium import webdriver
# Importar a classe ActionChains responsável pelas manipulações
from selenium.webdriver.common.action_chains import ActionChains
# Importar a classe Keys que podem ser utilizadas no key_up e key_down
from selenium.webdriver.common.keys import Keys

firefox = webdriver.Firefox()
actions = ActionChains(firefox)


# Clicar e manter na posição atual do mouse
actions.click_and_hold()
# Voltar o mouse para o estado inicial
actions.release()
# Disparar metódo para que as ações sejam executadas
actions.perform()


botao = firefox.find_element_by_id("#botaoqualquer")
# Clicar e manter em um botão
actions.click_and_hold(on_element=botao)
actions.perform()


# Clicar com o botão direito na posição atual do mouse
actions.context_click()
actions.perform()

# Clique duplo
actions.double_click(on_element=botao)
actions.perform()

# Exemplo com key_down e key_up. Simular um CTRL + C
actions.key_down(Keys.CONTROL) # Pressionar o CTRL
```

```
actions.send_keys('c') # Pressionar o C
actions.key_up(Keys.CONTROL) # Liberar o CTRL
actions.perform()
```

Tem algumas outras ações bem interessantes, vale dar uma olhada no arquivo.

Você pode visualizar as outras funções em: selenium/webdriver/common/action_chains.py

EventListener - Ouvindo seu código

Introdução

O Selenium provê uma forma bem simples para que possa monitorar tudo que está sendo feito pelo navegador. Para cada ação pode ser disparado um evento e esse evento pode ser "ouvido" por seu `EventListener`.

É necessário que a instância do navegador seja passada para uma `classe` que irá disparar todos os eventos e então construir seu `EventListener` para ouvi-lá.

Listener

A classe `AbstractEventListener` foi construída para ser herdada e alterada. Cada metódo da classe será executado quando determinado evento for disparado.

```
class AbstractEventListener(object):

  # Será invocado antes de uma url ser acessada pelo metódo "get" do webdriver
  def before_navigate_to(self, url, driver):  pass

  # Será invocado após uma url ser acessada pelo metódo "get" do webdriver
  def after_navigate_to(self, url, driver):   pass

  # Será invocado antes que o metódo "back" do webdriver seja executado
  def before_navigate_back(self, driver): pass

  # Será invocado depois que o metódo "back" do webdriver seja executado
  def after_navigate_back(self, driver):  pass

  # Será invocado antes que o metódo "forward" do webdriver seja executado
  def before_navigate_forward(self, driver):  pass

  # Será invocado depois que o metódo "forward" do webdriver seja executado
  def after_navigate_forward(self, driver):   pass

  # Será invocado antes que o metódo "find(s)_element(s)_by_*" seja executado
  def before_find(self, by, value, driver):   pass

  # Será invocado após que o metódo "find(s)_element(s)_by_*" seja executado
  def after_find(self, by, value, driver):    pass

  # Será invocado antes que o metódo "click" seja executado
  def before_click(self, element, driver):    pass

  # Será invocado após que o metódo "click" seja executado
  def after_click(self, element, driver): pass

  # Será invocado antes que o metódo "clear" ou "send_keys" seja executado
  def before_change_value_of(self, element, driver):  pass

  # Será invocado após que o metódo "clear" ou "send_keys" seja executado
```

```
  def after_change_value_of(self, element, driver):   pass

  # Será invocado antes que o metódo "execute_script" ou "execute_async_script"
seja executado
  def before_execute_script(self, script, driver):    pass

  # Será invocado após que o metódo "execute_script" ou "execute_async_script"
seja executado
  def after_execute_script(self, script, driver): pass

  # Será invocado antes que uma página seja fechada com o metódo "close"
  def before_close(self, driver): pass

  # Será invocado após que uma página seja fechada com o metódo "close"
  def after_close(self, driver):  pass

  # Será invocado antes que o metódo "quit" seja executado
  def before_quit(self, driver):  pass

  # Será invocado depois que o metódo "quit" seja executado
  def after_quit(self, driver):   pass

  # Será executado sempre que houver uma exeception
  def on_exception(self, exception, driver):  pass
```

Arquivo: selenium/webdriver/support/abstract_event_listener.py

Dispatch

A classe `EventFiringWebDriver` é responsável por disparar os eventos do navegador. Para que ela funcione é necessário passar uma instância do navegador e um `EventListener`.

Arquivo: selenium/webdriver/support/event_firing_webdriver.py

Seu próprio Listener

Vamos criar um `EventListener` simples para utilizarmos no nosso exemplo.

```
from selenium.webdriver.support.events import AbstractEventListener

class MyListener(AbstractEventListener):
    def before_navigate_to(self, url, driver):
        print "Antes de abrir a url %s" % url

    def after_navigate_to(self, url, driver):
        print "Depois de abrir a url %s" % url

    def before_click(self, element, driver):
        print "Antes de clicar no elemento"

    def after_click(self, element, driver):
        print "Depois de clicar no elemento"

    def before_close(self, driver):
        print "Antes de fechar a pagina"

    def after_close(self, driver):
        print "Depois de fechar a pagina"

    def on_exception(self, exeception, driver):
        print "Ocorreu um erro"
```

Para criar um `EventListener` é preciso criar uma classe que herde da `AbstractEventListener` e implementar os metódos que serão invocados a cada evento ou criar uma classe com os mesmos metódos.

Salve o código em um arquivo chamado "mylistener.py" para testes.

Juntando tudo

Para que tudo funcione corretamente, você precisa primeiro importar as bibliotecas necessárias.

```
# para instanciar o navegador
from selenium import webdriver

# EventFiringWebdriver para disparar os eventos
from selenium.webdriver.support.events import EventFiringWebDriver

# Importe o seu Listener
from mylistener import MyListener
```

Depois criar as instâncias:

```
firefox = webdriver.Firefox()
listener = MyListener()

# Firefox com os eventos sendo disparados
ef_firefox = EventFiringWebDriver(firefox, listener)
```

Agora é só executar algumas ações e você verá as informações sendo "printadas" na tela.

```
# abrir página da python club
ef_firefox.get('http://pythonclub.com.br/')

try:
    """
        Propositalmente irá gerar uma exception pois a classe não existe.
        O evento "on_exception" deve ser chamado
    """
    post = ef_firefox.find_element_by_class_name('post-tile')
except:
    pass

# localizamos o primeiro post
post = ef_firefox.find_element_by_class_name('post-title')

# clicar no elemento
post.click()

# fechar a página
ef_firefox.close()
```

Resultado:

```
>> python mylistener.py

Antes de abrir a url http://pythonclub.com.br/
Depois de abrir a url http://pythonclub.com.br/
Ocorreu um erro
Antes de clicar no elemento
Depois de clicar no elemento
Antes de fechar a pagina
Depois de fechar a pagina
```

Código completo: mylistener.py.

Agora você pode explorar o `AbstractEventListener` e o `EventFiringWebdriver` e construir seu próprio `EventListener` de acordo com suas necessidades.

The End

Acredito que esse seja o último post sobre Selenium, espero que tenham gostado!