Selenium extensions Documentation

Release 0.1.2

Vladyslav Ovchynnykov

Sep 04, 2017

Contents

Contents:

CHAPTER 1

Selenium extensions

Tools that will make writing tests, bots and scrapers using Selenium much easier

- Free software: MIT license
- Documentation: https://selenium-extensions.readthedocs.io.

Installation

```
$ pip install selenium_extensions
```

Example

Creating a headless Selenium bot and filling in a form is as easy as

```
from selenium.webdriver.common.by import By
from selenium_extensions.core import SeleniumDriver


class MyBot(SeleniumDriver):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def goto_google(self):
        self.driver.get('https://google.com')
        searchbox_locator = (By.ID, 'lst-ib')
        self.wait_for_element_to_be_present(searchbox_locator)
        self.populate_text_field(searchbox_locator, 'query')


bot = MyBot(browser='chrome', executable_path='/usr/bin/chromedriver', headless=True)
```

```
bot.goto_google()
bot.shut_down()
```

Or do you want to wait until you will be redirected from login page? `selenium_extensions` makes it easy

```
from selenium_extensions.helpers import wait_for_function_truth
from selenium_extensions.helpers import element_has_gone_stale


...
login_btn = self.driver.find_element_by_css_selector(
    "button.submit.EdgeButton.EdgeButton--primary")
login_btn.click()

# Wait to be redirected
wait_for_function_truth(element_has_gone_stale, login_btn)
```

Features

- `selenium_extensions.drivers.chrome_driver` - extended Chrome webdriver class with built-in support for headless mode and rendering webpages without media.
- `selenium_extensions.drivers.firefox_driver` - extended Firefox webdriver class with built-in support for headless mode and rendering webpages without media.
- `selenium_extensions.core.scroll` - scrolls the current page or the Selenium WebElement if one is provided.
- `selenium_extensions.core.element_is_present` - shortcut to check if the element is present on the current page.
- `selenium_extensions.core.wait_for_element_to_be_clickable` - waits for element described by *element_locator* to be clickable.
- `selenium_extensions.helpers.element_has_gone_stale` - checks if element has gone stale.
- `selenium_extensions.core.SeleniumDriver` - class with all necessary tools in one place. User's classes should inherit from this class and initialize it using `super()`. After this their class will have `driver` attribute and all the methods ready to go.

and more.

CHAPTER 2

Installation

Stable release

To install Selenium extensions, run this command in your terminal:

```
$ pip install selenium_extensions
```

This is the preferred method to install Selenium extensions, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

From sources

The sources for Selenium extensions can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/pythad/selenium_extensions
```

Or download the tarball:

```
$ curl  -OL https://github.com/pythad/selenium_extensions/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

There are three submodules you will look into for different utils:

Drivers

Provides shortcuts for drivers creation.

Available tools are:

- *selenium_extensions.drivers.chrome_driver()* - function to initialize `selenium.webdriver.Chrome` with extended options.
- *selenium_extensions.drivers.firefox_driver()* - function to initialize `selenium.webdriver.Firefox` with extended options.

Core

Provides core functionality of the package. All of the function from this module directy access the webdriver and its state.

Available tools are:

- *selenium_extensions.core.shut_down()* - shuts down the driver and its virtual display.
- *selenium_extensions.core.scroll()* - scrolls the current page or the Selenium WebElement if one is provided.
- *selenium_extensions.core.click_on_element()* - clicks on a Selenium element represented by `element_locator`.
- *selenium_extensions.core.element_is_present()* - shortcut to check if the element is present on the current page.

- *selenium_extensions.core.wait_for_element_to_be_present()* - shortcut to wait until the element is present on the current page.
- *selenium_extensions.core.wait_for_element_to_be_clickable()* - waits for element described by *element_locator* to be clickable.
- *selenium_extensions.core.populate_text_field()* - populates text field with provided text.
- *selenium_extensions.core.SeleniumDriver* - base class for selenium-based drivers. User's classes should inherit from this class and initialize it using `super()`. After this their class will have `driver` attribute and all the methods ready to go.

Helpers

Provides helpers for writing things using Selenium.

Available tools are:

- *selenium_extensions.helpers.kill_virtual_display()* - kills virtual display created by `pyvirtualdisplay.Display()`.
- *selenium_extensions.helpers.element_has_gone_stale()* - checks if element has gone stale.
- *selenium_extensions.helpers.wait_for_function_truth()* - waits for function represented by `condition_function` to return any non-False value.
- *selenium_extensions.helpers.join_css_classes()* - joins css classes into a single string.

About `core.SeleniumDriver`

selenium_extensions.core.SeleniumDriver provides all of the tools available in `selenium_extensions.core` in a single class. It also can create driver by calling `super()` from child class and then use it for all the `selenium_extensions.core` functionality, **so you don't need to provide driver as the first argument to SeleniumDriver's methods**. Let's look at some code:

```
from selenium.webdriver.common.by import By
from selenium_extensions.core import SeleniumDriver


class MyBot(SeleniumDriver):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def goto_google(self):
        self.driver.get('https://google.com')
        searchbox_locator = (By.ID, 'lst-ib')
        # core.wait_for_element_to_be_present is now available as self.wait_for_
↪element_to_be_present
        self.wait_for_element_to_be_present(searchbox_locator)
        # core.populate_text_field is now available as self.populate_text_field
        self.populate_text_field(searchbox_locator, 'query')


bot = MyBot(browser='chrome', executable_path='/usr/bin/chromedriver', run_
↪headless=True, load_images=False)
```

```
bot.goto_google()
bot.shut_down()  # core.shut_down() is now available as self.shut_down()
```

Another option, if you don't enjoy OOP style, would be we just to initialize `SeleniumDriver` and use its `driver` attribute to do whatever you want. So the code above could look like this:

```
from selenium.webdriver.common.by import By
from selenium_extensions.core import SeleniumDriver


bot = SeleniumDriver(browser='chrome', executable_path='/usr/bin/chromedriver',
                     run_headless=False, load_images=False)
bot.driver.get('https://google.com')
searchbox_locator = (By.ID, 'lst-ib')
bot.wait_for_element_to_be_present(searchbox_locator)
bot.populate_text_field(searchbox_locator, 'query')
bot.shut_down()
```

CHAPTER 4

selenium_extensions package

selenium_extensions.drivers module

selenium_extensions.drivers.**chrome_driver**(*executable_path=None*, *run_headless=False*, *load_images=True*)

Function to initialize `selenium.webdriver.Chrome` with extended options

Parameters

- **executable_path** (*str*) – path to the chromedriver binary. If set to `None` selenium will serach for `chromedriver` in `$PATH`.
- **run_headless** (*bool*) – boolean flag that indicates if chromedriver has to be headless (without GUI).
- **load_images** (*bool*) – boolean flag that indicates if Chrome has to render images.

Returns created driver.

Return type selenium.webdriver.Chrome

Note: In order to create Chrome driver Selenium requires Chrome to be installed and chromedriver to be downloaded.

Warning: Headless Chrome is shipping in Chrome 59 and in Chrome 60 for Windows. Update your Chrome browser if you want to use `headless` option.

selenium_extensions.drivers.**firefox_driver**(*executable_path=None*, *run_headless=False*, *load_images=True*)

Function to initialize `selenium.webdriver.Firefox` with extended options

Parameters

- **executable_path** (*str*) – path to the `geckdriver` binary. If set to `None` selenium will search for `geckdriver` in `$PATH`.
- **run_headless** (*bool*) – boolean flag that indicates if `geckodriver` has to be headless (without GUI). `geckodriver` doesn't support native headless mode, that's why `pyvirtualdisplay` is used.
- **load_images** (*bool*) – boolean flag that indicates if Firefox has to render images.

Returns created driver.

Return type selenium.webdriver.Firefox

Note: In order to create Firefox driver Selenium requires Firefox to be installed and geckodriver to be downloaded.

Note: Firefox doesn't support native headless mode. We use `pyvirtualdisplay` to simulate it. In order `pyvirtualdisplay` to work you need to install `Xvfb` package: `sudo apt install xvfb`.

---

selenium_extensions.core module

class `selenium_extensions.core.`**SeleniumDriver**(*browser=None*, *executable_path=None*, *run_headless=False*, *load_images=True*)

Base class for selenium-based drivers

User's classes should inherit from this class and initialize it using `super()`. After this their class will have `driver` attribute and all the methods ready to go.

Parameters

- **browser** (*'chrome' or 'firefox'*) – webdriver to use.
- **executable_path** (*str*) – path to the browser's webdriver binary. If set to `None` selenium will serach for browser's webdriver in `$PATH`.
- **run_headless** (*bool*) – boolean flag that indicates if webdriver has to be headless (without GUI).
- **load_images** (*bool*) – boolean flag that indicates if webdriver has to render images.

Raises `selenium_extensions.exceptions.SeleniumExtensionsException` – `browser` is not supported by `selenium_extensions`.

Example

```
from selenium_extensions.core import SeleniumDriver


class MyBot(SeleniumDriver):

    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def goto_google(self):
```

```
        self.driver.get('https://google.com')


bot = MyBot(browser='chrome', executable_path='/usr/bin/chromedriver', run_
↪headless=True, load_images=False)
bot.goto_google()
bot.shut_down()
```

Note: In order to create Chrome driver Selenium requires Chrome to be installed and chromedriver to be downloaded.

Note: In order to create Firefox driver Selenium requires Firefox to be installed and geckodriver to be downloaded.

Note: Firefox doesn't support native headless mode. We use `pyvirtualdisplay` to simulate it. In order `pyvirtualdisplay` to work you need to install `Xvfb` package: `sudo apt install xvfb`.

`selenium_extensions.core.`**`scroll`**(*driver*, *scroll_element=None*)

Scrolls the current page or the Selenium WebElement if one is provided

Parameters

- **`driver`** (*`selenium.webdriver.`*) – Selenium webdriver to use.
- **`scroll_element`** (*`selenium.webdriver.remote.webelement.WebElement`*) – Selenium webelement to scroll.

Examples

```
from selenium import webdriver
from selenium_extensions.core import scroll


driver = webdriver.Chrome()
scroll(driver)
```

```
from selenium import webdriver
from selenium_extensions.core import scroll


driver = webdriver.Chrome()
...
pop_up = driver.find_element_by_class_name('ff_pop_up')
scroll(driver, pop_up)
```

`selenium_extensions.core.`**`click_on_element`**(*driver*, *element_locator*)

Clicks on a Selenium element represented by `element_locator`

Parameters **`element_locator`** (*(`selenium.webdriver.common.by.By.`, `str`)*) – element locator described using *By*. Take a look at Locate elements By for more info.

Example

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium_extensions.core import click_on_element


driver = webdriver.Chrome()
...
click_on_element(driver, (By.ID, 'form-submit-button'))
```

selenium_extensions.core.**element_is_present**(*driver*, *element_locator*, *waiting_time=2*)

Shortcut to check if the element is present on the current page

Parameters

- **driver** (*selenium.webdriver.*) – Selenium webdriver to use.
- **element_locator** ((*selenium.webdriver.common.by.By., str*)) – element locator described using *By*. Take a look at Locate elements By for more info.
- **waiting_time** (*int*) – time in seconds - describes how much to wait.

Returns True if the element is present on the current page, False otherwise.

Return type bool

Example

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium_extensions.core import element_is_present


driver = webdriver.Chrome()
...
if not element_is_present(driver, (By.CLASS_NAME, 'search_photos_block')):
    pass # Do your things here
```

selenium_extensions.core.**wait_for_element_to_be_present**(*driver*, *element_locator*, *waiting_time=2*)

Shortcut to wait until the element is present on the current page

Parameters

- **driver** (*selenium.webdriver.*) – Selenium webdriver to use.
- **element_locator** ((*selenium.webdriver.common.by.By., str*)) – element locator described using *By*. Take a look at Locate elements By for more info.
- **waiting_time** (*int*) – time in seconds - describes how much to wait.

Raises selenium.common.exceptions.TimeoutException – timeout waiting for element described by element_locator.

Example

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium_extensions.core import wait_for_element_to_be_present


driver = webdriver.Chrome()
...
wait_for_element_to_be_present(driver, (By.CLASS_NAME, 'search_load_btn'))
```

selenium_extensions.core.**wait_for_element_to_be_clickable**(*driver*, *element_locator*, *waiting_time=2*)

Waits for element described by *element_locator* to be clickable

Parameters

- **element_locator** (*(selenium.webdriver.common.by.By., str)*) – element locator described using *By*. Take a look at Locate elements By for more info.
- **waiting_time** (*int*) – time in seconds - describes how much to wait.

Raises selenium.common.exceptions.TimeoutException – timeout waiting for element described by element_locator.

Example

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium_extensions.core import wait_for_element_to_be_clickable


driver = webdriver.Chrome()
...
wait_for_element_to_be_clickable(driver, (By.CLASS_NAME, 'form-submit-button'))
```

selenium_extensions.core.**populate_text_field**(*driver*, *element_locator*, *text*)

Populates text field with provided text

Parameters

- **element_locator** (*(selenium.webdriver.common.by.By., str)*) – element locator described using *By*. Take a look at Locate elements By for more info.
- **text** (*str*) – text to populate text field with.

Example

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium_extensions.core import populate_text_field


driver = webdriver.Chrome()
...
populate_text_field(driver, (By.CLASS_NAME, 'textbox'), 'some text')
```

selenium_extensions.core.**shut_down**(*driver*)
Shuts down the driver and its virtual display

Parameters **driver** (*selenium.webdriver.*) – Selenium webdriver to stop.

Example

```
from selenium import webdriver
from selenium_extensions.core import shut_down


driver = webdriver.Chrome()
...
shut_down(driver)
```

selenium_extensions.helpers module

selenium_extensions.helpers.**element_has_gone_stale**(*element*)
Checks if element has gone stale

Parameters **element** (*selenium.webdriver.remote.webelement.WebElement*) – Selenium webelement to check for.

Returns True if element has gone stale, False otherwise.

Return type bool

Examples

```
from selenium_extensions.helpers import element_has_gone_stale


if element_has_gone_stale(your_element):
    pass  # Do something
```

```
from selenium_extensions.helpers import wait_for_function_truth
from selenium_extensions.helpers import element_has_gone_stale


login_btn = driver.find_element_by_class_name('login_btn')
wait_for_function_truth(element_has_gone_stale, element)
```

selenium_extensions.helpers.**wait_for_function_truth**(*condition_function*, **args*, *time_to_wait=10*, *time_step=0.1*)
Waits for function represented by `condition_function` to return any non-False value

Parameters

- **condition_function** (*function*) – function to wait for.
- ***args** – arguments that should be applied to the function.
- **time_to_wait** (*int*) – time in seconds to wait.

- **time_step** (*float*) – step in seconds between checks.

Returns True if `wait_for_function_truth` succeeded and didn't reach `time_to_wait` limit

Return type bool

Raises `selenium.common.exceptions.TimeoutException` – timeout waiting for function described by `condition_function` to return any non-False value.

Example

```
from selenium_extensions.helpers import wait_for_function_truth
from selenium_extensions.helpers import element_has_gone_stale


login_btn = driver.find_element_by_class_name('login_btn')
wait_for_function_truth(element_has_gone_stale, element)
```

`selenium_extensions.helpers.`**kill_virtual_display**(*self*, *display*)

Kills virtual display created by `pyvirtualdisplay.Display()`

Parameters **display** (*pyvirtualdisplay.Display*) – display to kill.

Example

```
from selenium_extensions.helpers import kill_virtual_display


display = Display(visible=0, size=(1024, 768))
display.start()
...
kill_virtual_display(display)
```

`selenium_extensions.helpers.`**join_css_classes**(**args*)

Joins css classes into a single string

Parameters ***args** – arguments that represent classes that should be joined.

Returns a single string representing all of the classes.

Return type str

Examples

```
from selenium_extensions.helpers import join_css_classes


classes = join_css_classes('class1', 'class2')
print(classes)  # '.class2 .class2'
```

CHAPTER 5

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/pythad/selenium_extensions/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

Write Documentation

Selenium extensions could always use more documentation, whether as part of the official Selenium extensions docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/pythad/selenium_extensions/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *selenium_extensions* for local development.

1. Fork the *selenium_extensions* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/selenium_extensions.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv selenium_extensions
$ cd selenium_extensions/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
2. The pull request should work for Python 3.4, 3.5, 3.6.

CHAPTER 6

Credits

Development Lead

- Vladyslav Ovchynnykov <ovd4mail@gmail.com>

Contributors

None yet. Why not be the first?

CHAPTER 7

History

0.1.0 (2017-08-28)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index