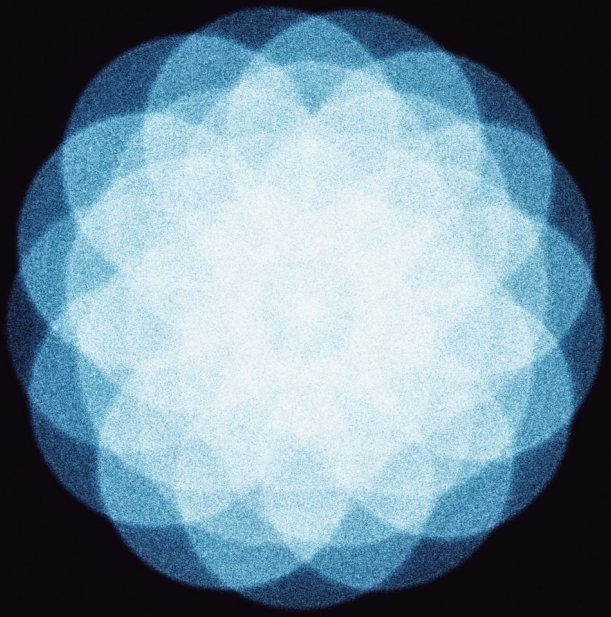
Искусство Автоматизации Тестирования с Selene



Яков Крамаренко

Содержание

Вступление

Исскуство Автоматизации

Исскуство Автоматизации Тестирования с Selene

Яков Крамаренко

Эта книга - практическое руководство по искусству автоматизации тестирования для начинающих ее изучение с самого нуля а также опытных автоматизаторов. Большое внимание уделено теме рефакторинга, написанию чистых, простых, читабельных и легких в поддержке тестов. Она раскрывает тему автоматизации пользовательского интерфейса веб-приложений с использованием следующих инструментов: Python, Selene, Selenium WebDriver, Pytest, Allure Reporting. Книга построена в виде сборника практических заданий и упражнений по Методу Кейсов.

Книгу можно купить по ссылке http://leanpub.com/selene-automation-ru.

Автор фото на обложке - Dan Hodgkins.

Эта версия была опубликована 2020-01-16.

Предисловие

В 2015 году я начал обучать на платных "офлайн" и онлайн ИТ-курсах (по программированию, автоматизации тестирования, и т.д.). Сначала я относился к этому как к временному заработку, совсем маленькому, но заработку за счет занятия делом, которое мне всегда приносило удовольствие - делиться опытом с другими, при этом структурируя знания и развиваясь самому. Со временем это занятие обросло четкой концепцией создания учебной программы и материалов, позволяющих начинающим ИТ-специалистам в быстрые строки обучаться практическим навыкам, и главное - обучать обучаться. Помня и ценя свой собственный опыт само-обучения, я строил программу на основе практических заданий с минимумом теории. Я старался не приподнести все секреты на блюдечке, наоборот, - предоставить возможность набить шишки студентам самим, но сделать это в ускоренном режиме, пройдя по специально разработанному маршруту, где будут встречаться реальные рабочие проблемы, которые я собирал годами своего опыта в IT.

Сфера обучения также меня интересовала, и продолжает интересовать - как источник специалистов для моих проектов. Я заметил, что надежней находить начинающих способных инженеров и обучать их нужным навыкам, чем переучивать "старичков", уровень знаний которых часто несоизмеримо мал по сравнению с их эгом:)

Со временем я заметил, что обычный формат курса с преподавателями и менторами - довольно тяжел в поддержке, и сложнее масштабируется. Так и появилась идея перевести его в формат книги.

В этой книге учащемуся предлагается пройти полный путь построения автоматизации небольшого веб приложения с помощью решения серии заданий, которые так или иначе ждут его и в реальном проекте. При этом перед началом работы над каждым заданием дается минимум теоретических знаний, которые могут быть доступны либо в самой книге, либо по ссылкам на другие публичные ресурсы. Если знаний уже должно быть достаточно - урок с теорией может быть упущен совсем.

Далее, в процессе работы над заданием, встречаясь с затруднениями и проблемами, если "гугл не помог", студент может подсматривать в следующий за заданием раздел с решением. Раздел может содержать детальное описание процесса решения задачи, или список с частыми вопросами и ответами, или список с частыми ошибками и их решениями. После самостоятельного выполнения задания, учащийся сможет окончательно себя проверить по этому же разделу.

На данный момент книга все еще находится в разработке. Поддержать которую можно купив книгу по рекомендуемой цене на этом сайте. Новые главы будут выходить со временем, без четких пределов по срокам. Но есть желание закончить основную часть до конца 2020 года. Главное и самое полезное содержание книги - это списки частых ошибок и их решений. Именно струкутурирование этой информации занимает самое большое время. Пока книга наполняется этой информацией, эта же програма доступна в виде онлайн-курса, где решения проверяются в стиле код-ревью мной и менторами. Записаться на курс можно обратившись по почте automician@gmail.com.

Программа и статус готовых разделов:

- Введение в Selene
- [TODO] 01 Начало. Проверка Концепции
- [TODO] 02 Пересмотр. Рефакторинг Теста - Часть 1
- [TODO] 03 Закалка. Расширение покрытия. Атомарные тесты
- [TODO] 04 Структура и переиспользование. Рефакторинг Тестов - Часть 2 (PageObjects)
- [TODO] 05 Отчетность
- [TODO] 06 Тестовые наборы
- [TODO] 07 Параметризация
- [TODO] 08 Масштабирование. Грид
- [TODO] 09 Интеграция. CI
- [TODO] 10 Оптимизация. Параллелизация
- [TODO] A1 Обзор Selenium Webdriver в сравнении с Selene
- Общий FAQ

В процессе написания книги, эта программа может изменятся.

К книге прилагаются рабочие образцы кода из теоретических разделов а также решения соответствующих заданий. Некоторые главы могут сопровождаться видео, выложенном в публичный доступ или доступном среди дополнительных материалов прилагаемых к книге.

Пока книга находится в разработке, будь готов к присутствию "багов", опечаток и неточностей. Список известных:

- не работают ссылки в содержании для некоторых разделов

Как работать с этой книгой

Эта книга-курс рассчитана, в первую очередь, на самостоятельное обучение. Для ее "прохождения" нужны определенные базовые навыки. Полный список таких навыков указан в разделе Необходимые знания перед стартом. В нем же можно найти публичные материалы, рекомендуемые к освоению.

Книга состоит из разделов, каждый из которых может состоять из следующего типа глав:

- урок
- задание
    - решение

При наличии урока, ты можешь ознакомиться с самыми базовыми знаниями темы. Желательно повторить самому примеры показанные в нем.

Далее ты приступаешь к заданию, которое следует попробовать решить полностью самомстоятельно, обращаясь за помощью, в первую очередь - к google.com. Будь аккуратен, ты можешь нагуглить и уже готовые решения других "студентов", выложенные где то на github.com - в них не подглядывай;). Если быстрый поиск не помог, тогда можно заглянуть в следующую главу с решением.

Глава с решением позволит либо полностью разобрать процесс работы над заданием, либо подсказать как решить очередную проблему. Либо после самостоятельно законченного задания - проверить себя по частым ошибкам и ответам. Обычно ошибки в начале - будут содержать подсказки а не ответы. Рекомендуется попробовать сначала сообразить решение по подсказкам, и только в самом конце - проверить правильный результат. Для удобтва, решение всегда будет идти на следующей за подсказками странице.

Удачи!

Необходимые знания перед стартом

Перед тем, как начать работу над упражнениями из книги, ты должен иметь следующую базу:

- HTML, CSS и JavaScript (самые основы). В интернете полно бесплатных интерактивных туториалов, просто загугли;). Вот всего лишь несколько примеров:
  - Intro to Software Development [HTML, CSS, JavaScript]
    - Как более обширная альтернатива, знакомящая с множеством языков и технологий: "CS50. Основы программирования" от javarush.ru
    - Как более сфокусированная на языке JavaScript альтернатива: Hexlet.io: Введение в программирование (JavaScript)
  - Hexlet.io: HTML & CSS
- Python. Гугл снова поможет;) Но вот несколько примеров ресурсов:
  - если уже знаком с другим языком программирования:
    - LearnXinYminutes: Python
      - на русском
    - python koans
  - code-basics: python
    - альтернатива: learnpython.org/
  - Hexlet.io: Python: Основы
  - Hexlet.io: Python: Настройка окружения
  - Introductions to Python Testing Frameworks

В целом, в некоторых случаях, даже прохождение половины материалов, упомянутых выше, - может быть достаточно, если ты начинаешь обучение автоматизации тестирования с самого нуля. Для автоматизации тестирования не нужно знать все особенности языка программирования. Вначале, автоматизация вообще может показаться довольно простой. И ты заметишь это по тому, как еще недоучив язык, сможешь пройти пол этой книги;) Обычно, для написания автоматизированных тестов, не придется использовать даже простые алгоритмы. Все сводится просто к вызову команд и структурированию кода. Но рано или поздно сложные задачи все

равно появятся. Поэтому, хотя в этой книге сложных алгоритмических упражнений будет не много, желательно сразу паралельно "набивать руку" в программировании более сложных алгоритмических задач, чтобы не обмануться простотой "первых проб в автоматизации". Следующие ресурсы могут в этом помочь, можешь выбрать тот который больше по душе:

- exercism.io: Python Track
- CheckIO
- repl.it: Python Auto-Graded Course with Solutions
- CodinGame
- codewars

В какой то момент обучения, обязательно стоит ознакомиться со следующими "класическими" материалами:

- Пиши код, как настоящий Питонист: идиоматика Python (перевод)
- Python Tips, Tricks, and Hacks
- Loop like a native
- Transforming Code into Beautiful, Idiomatic Python
- Python's Class Development Toolkit

Также, тебе будет более удобно работать над заданиями книги, если ты будешь сохранять версии своих решений с помощью систем контроля версий. Я рекомендую использовать git и один из самых популярных серверов - github.com. Вот и бесплатных гайд по теме: Github - Hello World.

Введение в Selene

Быстрый Старт. Исходные условия, зависимости, первые шаги и документация

Selene - это инструмент для автоматизации действий пользователя в браузере, ориентированный на удобство и легкость реализации бизнес-логики в автотестах, на языке пользователя, не отвлекаясь на технические детали работы с "драйвером браузера".

Например, к техническим деталям - можно отнести работу с ожиданиями элементов при автоматизации тестирования динамических веб-приложений, реализацию высокоуровневых действий над элементами, и так далее.

Давай с ним очень быстро познакомимся.

Итак, имея установленными следующие инструменты (можешь загуглить как - сам):

- pyenv + python
- poetry
- Chrome Browser
- ChromeDriver (не обязательно, Selene сможет установить его сам)
- PyCharm (достаточно "Community Edition")

В unix-терминале (под Windows доступен через Windows Subsystem for Linux либо через git bash) выполним следующее...

Создадим новый проект:

```
$ poetry new selene-intro
Created package selene-intro in selene-intro
$ cd selene-intro
$ ls
README.rst    pyproject.toml    selene_intro    tests
```

Выберем нужную версию питона:

```
$ pyenv local 3.7.3
$ python -V
Python 3.7.3
```

Открыв проект в PyCharm, увидим довольно простую структуру проекта:

```
selene-intro
├── pyproject.toml
├── README.rst
├── selene_intro
│   └── __init__.py
└── tests
    ├── __init__.py
    └── test_poetry_demo.py
```

Тут...

- `selene-intro` - папка с проектом
- `pyproject.toml` - файл конфигурации проекта в формате TOML
- `README.rst` - базовая документация в формате ReST (заменив расширение с `.rst` на `.md` можно поменять формат на Markdown если он больше по душе)
- `selene_intro` - главный корневой модуль нашего проекта, в котором в init-файле хранится версия проекта:

```
__version__ = '0.1.0'
```

- `tests` - модуль с тестами, с примером простого теста проверяющего текущую версию:

```
from selene_intro import __version__


def test_version():
    assert __version__ == '0.1.0'
```

Нас инересует в первую очередь `pyproject.toml` . Следующие конфигурации проекта были сгенерированы автоматически при создании проекта, и значения их очевидны и говорят сами за себя:

```
[tool.poetry]
name = "selene-intro"
version = "0.1.0"
description = ""
authors = ["yashaka <yashaka@gmail.com>"]

[tool.poetry.dependencies]
python = "^3.7"

[tool.poetry.dev-dependencies]
pytest = "^3.0"

[build-system]
requires = ["poetry>=1.0.0"]
build-backend = "poetry.masonry.api"
```

В зависимостях...

```
# ...
[tool.poetry.dependencies]
python = "^3.7"

[tool.poetry.dev-dependencies]
pytest = "^3.0"
# ...
```

уже подключена библиотека pytest, которую мы и будем использовать для организации тестов и их запуска.

Теперь, все что нужно сделать, чтобы начать работать с Selene - это добавить и его в зависимости проекта:

```
# ...
[tool.poetry.dependencies]
python = "^3.7"
```

```
selene = {version = "^2.*", allow-prereleases = true}

[tool.poetry.dev-dependencies]
pytest = "^3.0"
# ...
```

И все это доустанавливать с терминала:

```
$ poetry install
```

Или сделать последние два шага одной командой:

```
$ poetry add selene --allow-prereleases
```

В процессе установки poetry создаст виртуальное окружение и установит туда все нужные зависимости, сохранив их актуальные версии в файле "poetry.lock". Если используешь `git` , не забудь добавить этот файл под систему контроля версий ;)

Теперь в PyCharm стоит указать в `Preferences>Project: selene-intro>Project Interpreter` путь к созданному виртуальному окружению, что-бы IDE понимала где искать используемые зависимости в проекте. Путь можно взять из лога запуска команды `poetry install` :

```
$ poetry install
Creating virtualenv selene-intro-py3.7 in
/Users/yashaka/Library/Caches/pypoetry/virtualenvs
Updating dependencies
Resolving dependencies... (6.5s)

Writing lock file
...
```

Теперь, чтобы использовать всю прелесть Selene, в коде достаточно просто сделать пару импортов в стиле:

```
from selene import by, be, have
from selene.support.shared import browser
#...
```

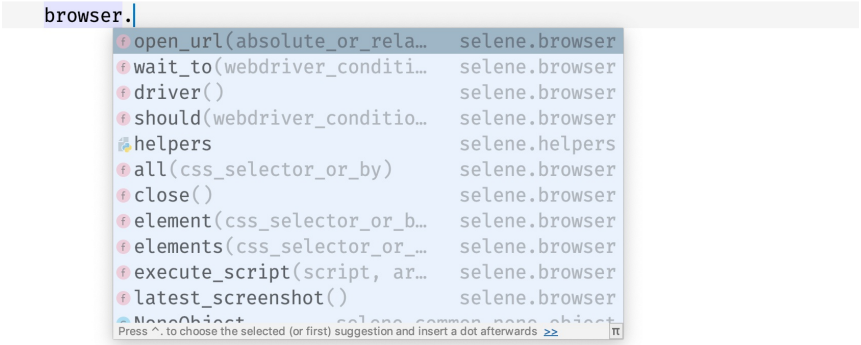
Самые-самые базовые "команды" Selene (методы обьекта `browser.*`) - это:

- `browser.open(url)` - загружает страницу по URL (и открывает браузер автоматически, если он еще не открыт)
- `browser.element(selector)` - ищет элемент по селектору (css, xpath или специальному селектору из модуля `by.*`)
- `browser.all(selector)` - ищет коллекцию элементов по селектору (css, xpath, или специальному селектору из модуля `by.*`)

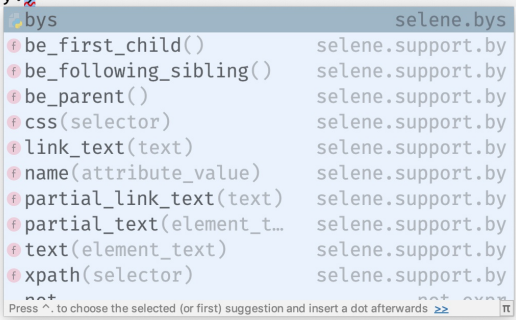
Специализированные селекторы, такие как поиск по аттрибутам (например `by.name`) или по тексту (`by.text`) живут в модуле `by.*`.

В модулях `be.*` и `have.*` живут условия для проверки элементов с помощью метода `should`, например `have.exact_text` - для проверки точного текста у элемента.

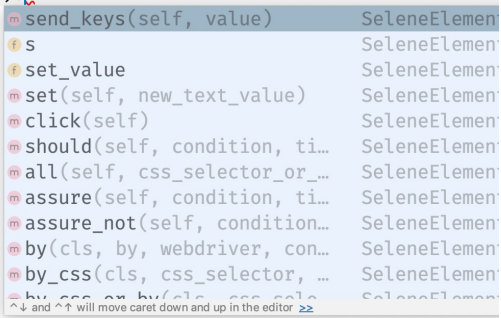
В принципе, этого достаточно, чтобы начать работу более опытному специалисту. Все остальные команды будут подсказаны любимым IDE. После ввода точки в коде, IDE подскажет все допустимые команды над объектом.



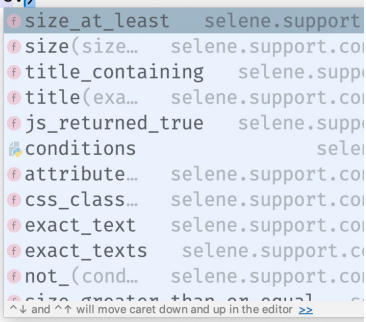
```
browser.element(by.)
```




```
browser.element(by.name('q')).
```



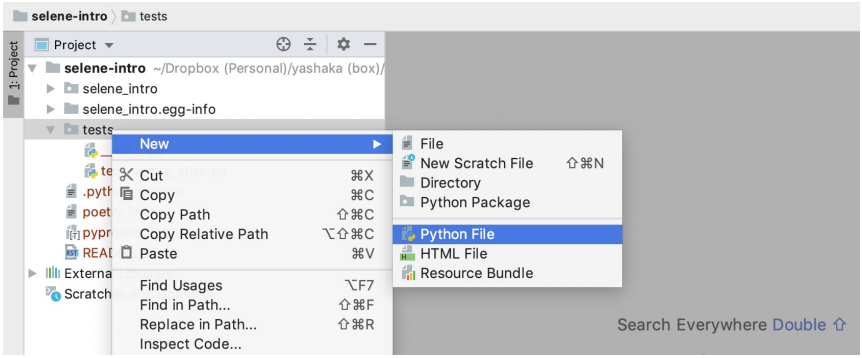

```
browser.element(by.name('q')).should(have.)
```

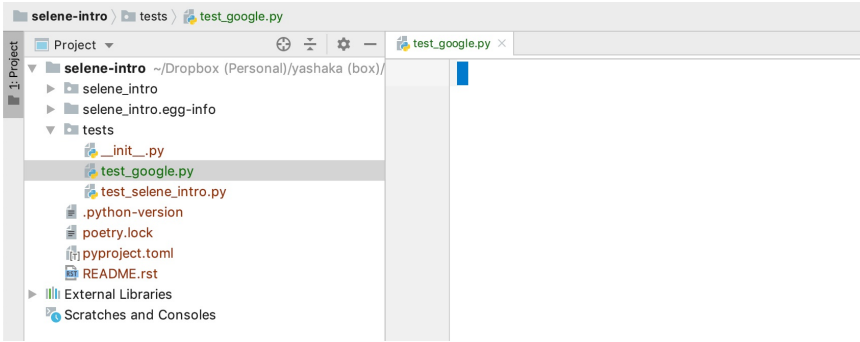


Таким образом, Selene предоставляет набор инструментов для автоматизации сценариев пользователя его же языком, используя привычную для него терминологию, что довольно важно для приемочного уровня тестирования:

```
browser.element(by.name('q')).should(be.blank).type('selenium').press_enter()
```

В таком вот иследовательском режиме можно и гугл успеть протестировать ;)





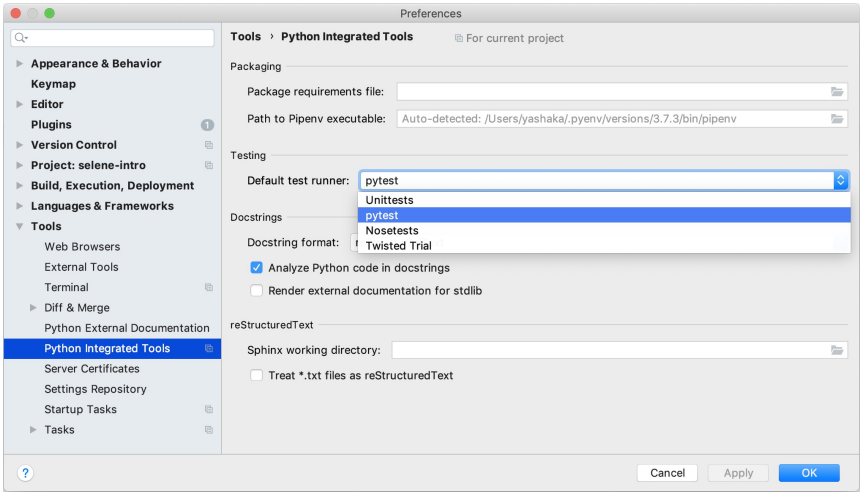
...

```
from selene import by, be, have
from selene.support.shared import browser


def test_search():
    browser.open('https://google.com/ncr')

    browser.element(by.name('q')).should(be.blank)\
        .type('python selene').press_enter()

    browser.all('#search .g').should(have.size_greater_than_or_equal(6))\
        .first.should(have.text('Concise API for Selenium'))\
        .element('.r>a').click()

    browser.should(have.title_containing('yashaka/selene'))
```

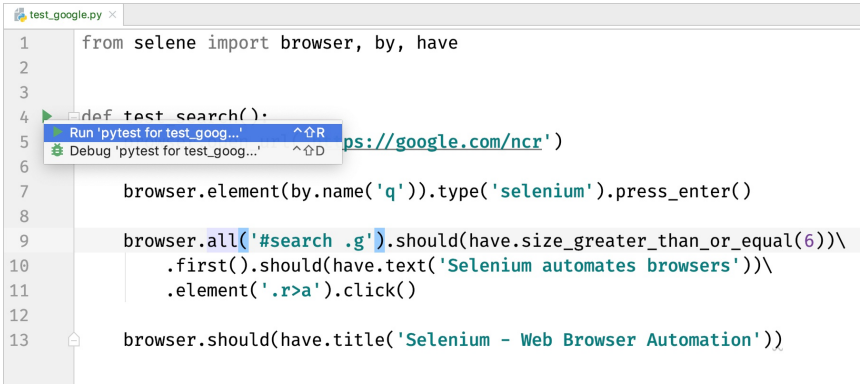
Или чуть более многословно, но возможно более понятно:

```
from selene import by, be, have
from selene.support.shared import browser


def test_search():
    browser.open('https://google.com/ncr')

    browser.element(by.name('q')).should(be.blank)\
        .type('python selene').press_enter()

    results = browser.all('#search .g')
    results.should(have.size_greater_than_or_equal(6))
    results.first.should(have.text('Concise API for Selenium'))
    results.first.element('.r>a').click()

    browser.should(have.title_containing('yashaka/selene'))
```

Если было время настроить тест-раннер в PyCharm:



То запустить тест можно прямо с IDE:



Другой способ - старый верный запуск с терминала:

```
$ poetry run py.test tests/test_google.py
```

Каждый из способов должен нам показать красивое кино в браузере Chrome;)

Если не хватает информации о том, что да как работает, всегда можно "провалиться" (`Cmd+Click` на Mac OS, `Ctrl+Click` на Windows) в код реализации нужных методов и разобраться с кодом.

Например, узнать что там еще входит в API можно провалившись в `from selene` в строке с импортами. Тебя там ждет даже небольшая документация с примерами их использования;)

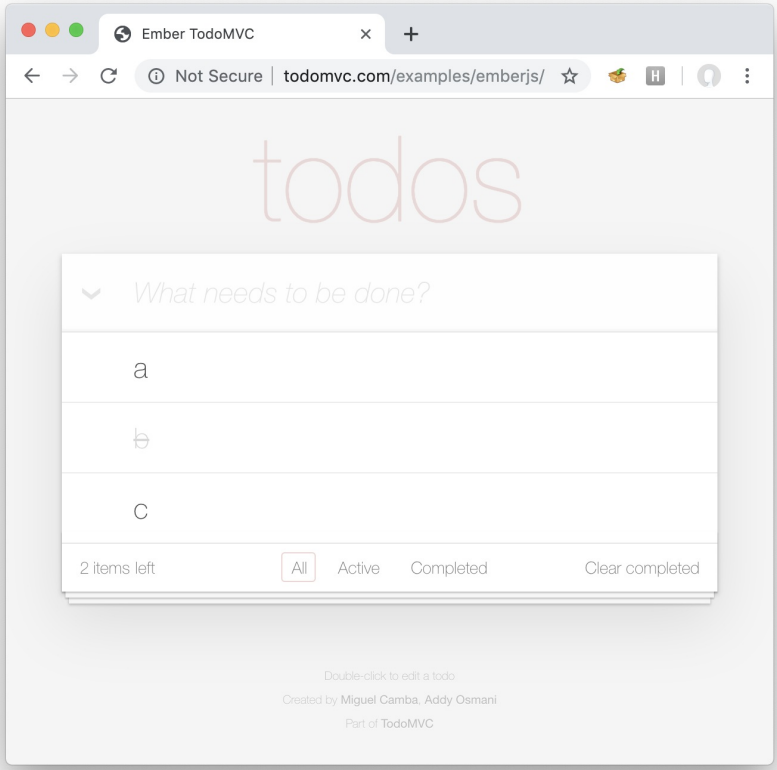
Ссылки на дополнительную документацию а также готовые примеры использования Selene можно найти в официальном README проекта.

Selene в действии

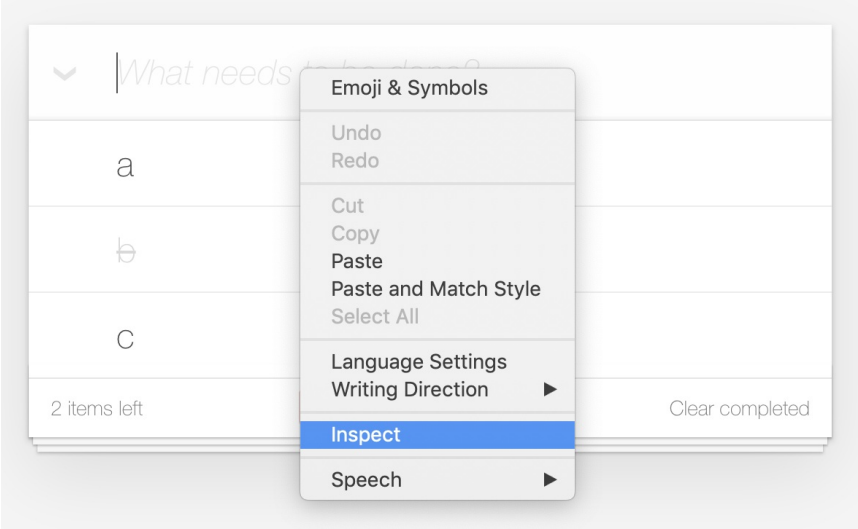
Давай теперь чуть более детально познакомимся с Selene на примере реализации тест-сценария для приложения - менеджера задач - TodoMVC:

```
# selene-intro/tests/test_todomvc.py

def test_complete_task():
    # open TodoMVC page

    # add tasks: 'a', 'b', 'c'
    # tasks should be 'a', 'b', 'c'

    # toggle 'b'
    # completed tasks should be 'b'
    # active tasks should be 'a', 'c'
    pass
```

Давай сразу поиграемся с этим приложением, попробуем вручную повторить этот сценарий...


Ember TodoMVC
Not Secure | todomvc.com/examples/emberjs/
todos
What needs to be done?
a
b
c
2 items left
All
Active
Completed
Clear completed
Double-click to edit a todo
Created by Miguel Camba, Addy Osmani
Part of TodoMVC

И сразу зароемся в структуру html для такого набора задач...



... обращая внимания на элементы с которыми мы взаимодействуем и игнорируя те элементы, которые нам пока не интересны:

```
<section id="todoapp">
  <header id="header">
    <!-- ... -->
    <input type="text" id="new-todo" placeholder="What needs to be done?" auto
focus="">
  </header>
  <section id="main" class="ember-view">
    <!-- ... -->
    <ul id="todo-list" class="todo-list">
      <li id="ember267" class="ember-view">
        <div class="view">
          <input type="checkbox" class="toggle">
          <label>a</label>
          <!-- ... -->
        </div>
        <!-- ... -->
      </li>
      <li id="ember317" class="completed ember-view">
```

```
        <div class="view">
          <input type="checkbox" class="toggle">
          <label>b</label>
          <!-- ... -->
        </div>
        <!-- ... -->
      </li>
      <li id="ember319" class="ember-view">
        <div class="view">
          <input type="checkbox" class="toggle">
          <label>c</label>
          <!-- ... -->
        </div>
        <!-- ... -->
      </li>
    </ul>
  </section>
  <footer id="footer">
    <!-- ... -->
    <ul id="filters">
      <li><a href="#/" id="ember275" class="selected ember-view">All</a></li>
      <li><a href="#/active" id="ember282" class="ember-view">Active</a></li>
      <li><a href="#/completed" id="ember298" class="ember-view">Completed</a>
</li>
    </ul>
    <!-- ... -->
  </footer>
</section>
```

Вот они, эти элементы, нужные для соответствующих действий нашего сценария:

- add 'a', 'b', 'c'

```
      <input type="text" id="new-todo" placeholder="What needs to be done?" a
    utofocus="">
```

- tasks should be 'a', 'b', 'c'

```
      <ul id="todo-list" class="todo-list">
        <li id="ember267" class="ember-view">
```

```
      <div class="view">
        <input type="checkbox" class="toggle">
        <label>a</label>
        <!-- ... -->
      </div>
      <!-- ... -->
    </li>
    <li id="ember317" class="completed ember-view">
      <div class="view">
        <input type="checkbox" class="toggle">
        <label>b</label>
        <!-- ... -->
      </div>
      <!-- ... -->
    </li>
    <li id="ember319" class="ember-view">
      <div class="view">
        <input type="checkbox" class="toggle">
        <label>c</label>
        <!-- ... -->
      </div>
      <!-- ... -->
    </li>
  </ul>
```

- toggle 'b'

```
    <li id="ember317" class="completed ember-view">
      <div class="view">
        <input type="checkbox" class="toggle">    <!-- <<< -->
        <label>b</label>
```

- completed tasks should be b

```
    <li id="ember317" class="completed ember-view">
      <div class="view">
        <input type="checkbox" class="toggle">
        <label>b</label>
        <!-- ... -->
      </div>
```

```
        <!-- ... -->
```

- active tasks should be a, c

```
<ul id="todo-list" class="todo-list">
  <li id="ember267" class="ember-view">
    <div class="view">
      <input type="checkbox" class="toggle">
      <label>a</label>
      <!-- ... -->
    </div>
    <!-- ... -->
  </li>
  <li id="ember319" class="ember-view">
    <div class="view">
      <input type="checkbox" class="toggle">
      <label>c</label>
      <!-- ... -->
    </div>
    <!-- ... -->
  </li>
</ul>
```

Теперь, зная характеристики-атрибуты наших элементов, мы сможем их найти в коде и совершить нужные действия, знать бы только, чем их искать и как потом с ними взаимодействовать.

Ну вперед, начнем же, реализовывать наш сценарий:

```
def test_complete_task():
    # open TodoMVC page

    # add tasks: 'a', 'b', 'c'
    # tasks should be 'a', 'b', 'c'

    # toggle 'b'
    # completed tasks should be 'b'
    # active tasks should be 'a', 'c'
    pass
```

Для первого шага команда очевидна, нужно просто передать нужный URL как параметр в `browser.open(...)`:

```
from selene import browser

def test_complete_task():
    # open TodoMVC page
    browser.open('http://todomvc.com/examples/emberjs/')

    # add tasks: 'a', 'b', 'c'
    # tasks should be 'a', 'b', 'c'

    # toggle 'b'
    # completed tasks should be 'b'
    # active tasks should be 'a', 'c'
```

Для следующего шага, нам нужно вспомнить последовательность действий для добавления одной задачи:

```
#...

# add task 'a':
#   1) find "new task" edit field 2) set value to 'a' 3) press Enter
```

Теперь перевести это на "язык Selene" не составит труда, используя подсказки IDE для поиска нужных нам команд-действий над элементом:

```
#...

# add task 'a':
#   1) find "new task" edit field 2) set value to 'a' 3) press Enter
browser.element(by.id('new-todo')).type('a').press_enter()
```

Команду `element` мы используем, чтобы получить доступ к элементу на странице ...

```
<input type="text" id="new-todo" placeholder="What needs to be done?"
       autofocus="">
```

... по локатору, находящему элемент по уникальному идентификатору: `by.id('new-todo')`

Можно использовать CSS селектор, напрямую передавая его команде `element`, и получить чуть более лаконичный код:

```
#...

# add task 'a':
#   1) find "new task" edit field 2) set value to 'a' 3) press Enter
browser.element('#new-todo').type('a').press_enter()
```

Теперь, можно добавить и другие задачи, используя очень полезный подход - "Copy & Paste Driven Development" ;)

```
#...

# add tasks 'a', 'b', 'c'
browser.element('#new-todo').type('a').press_enter()
browser.element('#new-todo').type('b').press_enter()
browser.element('#new-todo').type('c').press_enter()

# tasks should be 'a', 'b', 'c'
# ...
```

Теперь найдем все задачи в списке с помощью команды `browser.all` и проверим, что они имеют соответствующие тексты:

```
#...

# add tasks 'a', 'b', 'c'
browser.element('#new-todo').type('a').press_enter()
browser.element('#new-todo').type('b').press_enter()
browser.element('#new-todo').type('c').press_enter()

# tasks should be 'a', 'b', 'c'
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))
```

Как видим, "язык Selene" не очень отличается от "английского" ;) Проверки над элементами или коллекциями элементов, как в последнем случае, совершаются при помощи метода `should`, которому передается условие для проверки. Наборы условий доступны через синтаксис `have.*` и `be.*`, что и позволяет составлять код в виде читабельных английских фраз.

```
from selene import have, be
#...

# add tasks 'a', 'b', 'c'
browser.element('#new-todo').type('a').press_enter()
browser.element('#new-todo').type('b').press_enter()
browser.element('#new-todo').type('c').press_enter()

# tasks should be 'a', 'b', 'c'
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))
```

Давай, теперь придумаем пример для использования `be.*` ;)

Обрати внимание, что если посмотреть "source" веб-страницы TodoMVC в браузере:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<title>Todomvc</title>
<meta name="description" content="">
<meta name="viewport" content="width=device-width, initial-scale=1">
<meta name="todomvc/config/environment" content="%7B%22modulePrefix%22%3A%22to
domvc%22%2C%22environment%22%3A%22production%22%2C%22baseURL%22%3Anull%2C%22lo
cationType%22%3A%22hash%22%2C%22EmberENV%22%3A%7B%22FEATURES%22%3A%7B%7D%2C%22
EXTEND_PROTOTYPES%22%3A%7B%22Date%22%3Afalse%7D%7D%2C%22APP%22%3A%7B%22name%22
%3A%22todomvc%22%2C%22version%22%3A%220.0.0+47754603%22%7D%2C%22exportApplicat
ionGlobal%22%3Afalse%7D" />

<link rel="stylesheet" href="assets/vendor7b5c98520910afa58d74e05ec86cd
873.css" integrity="sha256bsagGHduhay9QPLUFpddcZFq7Kmr2ScM3VKnWhdX8oM=sha512eN
sGN2aLecWPvoqNVH8oXK8o/IJ7rO5ti0zgS8lF8LiwmKUHdEIuFduwcDL1VLAt2r+3YjgDzoSNYK6c
57pJzw==" >
```

```
<link rel="stylesheet" href="assets/todomvcd41d8cd98f00b204e9800998ecf842
7e.css" integrity="sha25647DEQpj8HBSa+/TImW+5JCeuQeRkm5NMpJWZG3hSuFU= sha51
2z4PhNX7vuL3xVChQ1m2AB9Yg5AULVxXcg/SpIdNs6c5H0NE8XYXysP+DGNKHfuwvY7kxvUdBeoGlO
DJ6+SfaPg==" >

</head>

<body>

<script src="assets/vendor-22a6a947beb9d4b28a782879e18b0f65.js" integrity="sha
256-M1pD1q8B9PyrHkKX/mlfsOLLHMrh/x7vvCGLRC63OyI= sha512-I4vC6+4Z29iHB4nJBCzcIj
rgMtDerq7sxYLE2lM/AhjkNLp6gf2+Zpne8OtGdTLRkyvTqzPm1V7gq9w4HQxOXg==" >
</script>
<script src="assets/todomvc-d191d5c1c9280b108d69413f052d3bb4.js" integrity="sh
a256-1wUxToLQP6yjsvm0/8e3xQnv7SbSYcj3P/OPEdThZOk= sha512-X5K7gsPRYsUSRvJcnj80S
LlclDd4X/g1Qgle1L1P4Zb63eUM0mYEfQECBcjcks7iZFItSGr8EVPOVELXO35HUA==" >
</script>

</body>
</html>
```

То окажется, что никаких знакомых нам элементов типа `#new-todo` - там нет :) Это значит, как минимум, то, что эти элементы добавляются динамически после загрузки html страницы с помощью JavaScript. То есть, они реально "появляются на странице после загрузки", а не сразу. И, конечно же, это занимает какое-то время. Возможно, нам просто повезло, ничего не тормозило, и JavaScript динамически добавил наш элемент `#new-todo` достаточно быстро, чтобы мы уже смогли продолжить наш сценарий. Ну, наверное не всегда ж нам будет везти, и стоит, наверное, этот момент предвидеть и дождаться видимости элемента перед тем, как совершать нужные действия:

```
#...

# add tasks 'a', 'b', 'c'
browser.element('#new-todo').should(be.visible).type('a').press_enter()
browser.element('#new-todo').type('b').press_enter()
browser.element('#new-todo').type('c').press_enter()

# tasks should be 'a', 'b', 'c'
```

```
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))
```

Метод `should` - играет как раз роль "ожидания момента, когда элемент будет удовлетворять условию", а не только совершает проверку по условию...

А теперь хорошие новости - на самом деле, писать `.should(be.visible)` перед каждым действием не нужно. Selene сам подождет пока элемент будет доступен для выполнения действия над ним:

```
#...
browser.element('#new-todo').type('a').press_enter()
#...
```

Это одно из главных отличий Selene от "чистого" Selenium Webdriver. В Selenium неявные ожидания отключены по умолчанию и если включены - ждут только до появления элемента в DOM страницы, но при этом элемент все еще может быть невидимым и с ним не получится взаимодействовать, тест упадет. Или до окончания загрузки элемент может быть перекрыт другим элементом, и поэтому все еще быть недоступным для выполнения действия над ним. В Selene же ожидания по умолчанию включены, с таймутом ожидания по умолчанию, равному `4` , и доступным к конфигурированию через:

```
from selene import have
from selene.support.shared import browser, config
#...
config.timeout = 6;
#...

# add tasks 'a', 'b', 'c'
browser.element('#new-todo').type('a').press_enter()
browser.element('#new-todo').type('b').press_enter()
browser.element('#new-todo').type('c').press_enter()

# tasks should be 'a', 'b', 'c'
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))
```

Можно обойтись и без явного импорта:

```
from selene import have
from selene.support.shared import browser
#...
browser.config.timeout = 6;
#...
```

Этот `config` - целый склад всевозможных настроек, которыми можно тюнить поведение Selene и его команд, обязательно поисследуй все возможные опции;)

Например, через соответствующее поле этого класса, можно поменять тип используемого браузера, и вместо запуска тестов в Chrome, запустить их в Firefox:

```
#...
config.browser_name = "firefox"

# add tasks 'a', 'b', 'c'
element('#new-todo').type('a').press_enter()
element('#new-todo').type('b').press_enter()
element('#new-todo').type('c').press_enter()

# tasks should be 'a', 'b', 'c'
elements('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))
```

Давай теперь вернемся к нашему сценарию...

```
# open TodoMVC page
browser.open('http://todomvc.com/examples/emberjs/')

# add tasks: 'a', 'b', 'c'
browser.element('#new-todo').type('a').press_enter()
browser.element('#new-todo').type('b').press_enter()
browser.element('#new-todo').type('c').press_enter()
# tasks should be 'a', 'b', 'c'
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))

# toggle 'b'
# completed tasks should be 'b'
# active tasks should be 'a', 'c'
```

и закончим его реализацию;)

Следующий шаг - обозначить задачу как 'completed'. Для этого нужно:

```
# ...

# toggle 'b'
    # 1. among all tasks 2. find the one with 'b' text
    # 3. find its 'toggle' checkbox 4. click it
```

В Selene это сделать очень просто:

```
# ...

# toggle b

      # among all tasks
                           # find the one with 'b' text
browser.all('#todo-list>li').element_by(have.exact_text('b'))\
    .element('.toggle').click()
                       # click it
   # find its 'toggle' checkbox
}
```

Как видно, для поиска нужного элемента среди других элементов коллекции, мы используем такого же вида условие как и для "ожиданий-проверок":
`.element_by(have.exact_text('b'))`

Без таких возможностей Selene, в обычном Selenium Webdriver пришлось бы использовать намного более громоздкий и менее читабельный XPath-селектор.

Финальные шаги отвечают за проверку результата предыдущего действия:

```
# ...

# tasks should be 'a', 'b', 'c'
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))

# toggle b
browser.all('#todo-list>li').element_by(have.exact_text('b'))\
```

```
    .element('.toggle').click()

# completed tasks should be 'b':
    # among all tasks - filter only completed ones - check their texts
# active tasks should be 'a', 'c':
    # among all tasks - filter only not completed ones - check their texts
}
```

Перевод на "язык Selene" все такой же простой:

```
# ...

# completed tasks should be 'b'

      # among all tasks
                          # filter only completed ones
browser.all('#todo-list>li').filtered_by(have.css_class('completed'))\
    .should(have.exact_texts('b'))
   # check their texts

# active tasks should be 'a', 'c'

      # among all tasks
                          # filter only not completed ones
browser.all('#todo-list>li').filtered_by(have.no.css_class('completed'))\
    .should(have.exact_texts('a', 'c'))
   # check their texts
```

Как видно, в целом код действительно получился настолько читабельным, что наши комментарии...

```
# open TodoMVC page
browser.open('http://todomvc.com/examples/emberjs/')

# add tasks: 'a', 'b', 'c'
browser.element('#new-todo').type('a').press_enter()
browser.element('#new-todo').type('b').press_enter()
browser.element('#new-todo').type('c').press_enter()

# tasks should be 'a', 'b', 'c'
```

```
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))

# toggle 'b'
browser.all('#todo-list>li').element_by(have.exact_text('b'))\
    .element('.toggle').click()

# completed tasks should be 'b'
browser.all('#todo-list>li').filtered_by(have.css_class('completed'))\
    .should(have.exact_texts('b'))

# active tasks should be 'a', 'c'
browser.all('#todo-list>li').filtered_by(have.no.css_class('completed'))\
    .should(have.exact_texts('a', 'c'))
```

... совсем не нужны:

```
browser.open('http://todomvc.com/examples/emberjs/')

browser.element('#new-todo').type('a').press_enter()
browser.element('#new-todo').type('b').press_enter()
browser.element('#new-todo').type('c').press_enter()
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))

browser.all('#todo-list>li').element_by(have.exact_text('b'))\
    .element('.toggle').click()
browser.all('#todo-list>li').filtered_by(have.css_class('completed'))\
    .should(have.exact_texts('b'))
browser.all('#todo-list>li').filtered_by(have.no.css_class('completed'))\
    .should(have.exact_texts('a', 'c'))
```

Этих основных команд, поддержки функций Autocomplete и Quick Fix от IDE, мужества покопаться в коде, и, в конце концов, официальной документации должно быть вполне достаточно, чтобы начать писать первые автотесты с Selene + Python.

Удачи;)

Задание: Selene и CSS селекторы

Особенность Selene в том, что при его использовании, обычно, не нужны сложные CSS или XPath селекторы. Достаточно самых простых селекторов, которые ищут элементы по атрибутам, например:

```
browser.element(by.name('q'))  # ...

browser.element(by.id('new-todo'))  # ...
# или еще проще для последнего примера:
browser.element('#new-todo')  # ...
```

Более же сложные сложные варианты поиска элементов достигаются использованием дополнительных "команд" Selene, таких как `Collection#element_by(condition)`, `Element#element(selector)`, которые позволяют построить более читабельный и потому более легкий в поддержке код:

```
browser.all('#todo-list>li').element_by(have.exact_text('b'))\
        .element('.toggle').click()
```

Но чтобы прочувствовать все это на собственном опыте, да еще и закрепить знания в CSS селекторах, твоим заданием будет реализовать сценарий из раздела "Selene в действии":

```
def test_complete_task():
    # open TodoMVC page

    # add tasks: "a", "b", "c"
    # tasks should be "a", "b", "c"

    # toggle b
    # completed tasks should be b
    # active tasks should be a, c
```

для приложения TodoMVC (emberjs version), не используя при этом следующих методов Selene:

- `SeleneElement#element` (то есть `browser.element(".parent").element(".child")`)
- `SeleneElement#all` (то есть `browser.element(".parent").all(".child")`)
- `SeleneCollection#[index]` (то есть `browser.all(".item")[1]`)
- `SeleneCollection#first`
- `SeleneCollection#element_by(Condition condition)`
- `SeleneCollection#filtered_by(Condition condition)`
- и других им подобных ;)

По сути, все, чем можно пользоваться - это:

- `browser.element(...).*`, где `*` - это действия над элементом или ассерты (`should`)
- `browser.all(...).should(...)`
- и CSS селекторами для составления локаторов, которые можно передавать просто как строки, например: `browser.element('#new-todo')`

При этом важно стараться написать код как можно более аналогичный показанному в "Selenide в действии".

CSS селекторы - очень простые, удобные и читабельные (проще чем XPath селекторы), но при этом - довольно ограниченные в возможностях (по сравнению с теми же XPath селекторами). Поэтому, в любом случае - написать код, аналогичный тому, что был показан в "Selenide в действии" - не получится. Ну что ж, придется подумать и найти другой путь в поиске элементов и проверок над ними;)

Также, не спеши создавать переменные или функции для хранения/переиспользования локаторов либо тестовых данных. Мы к этому еще вернемся в следующих заданиях. Сейчас важно двигаться шаг за шагом, детально прорабатывая темы на которых мы фокусируемся в каждом из уроков.

Дополнительные материалы:

- Краткая шпаргалка по CSS селекторам от w3school
- Рецепты CSS и XPATH
- Видео от automated-testing.info о том, как писать локаторы (CSS, XPath)

В особо трудную минуту, можно подглядывать за подсказками в следующую главу с FAQ. В нем же можно будет найти список частых ошибок по которому проверить свое финальное решение;)

Удачи!

Решение: Список частых вопросов

"Строгие, но хрупкие" или "слабые, но стабильные" локаторы?

Строгими называются локаторы (или "селекторы", что, в принципе, примерно одно и то же) с более точной информацией о предмете поиска – элементе. Они более *хрупкие*, ведь если что-то поменяется в "пути селектора" – элемент уже не будет найден, и тест упадет, даже если тестируемая фича все еще будет работать. Все это будет увеличивать время поддержки существующих тестов. На некоторых проектах, где структура html часто меняется – это может привести к очень большим неоправданным затратам.

В противоположность строгим, **слабые** локаторы используют только минимум информации о элементе, его атрибутах и месте в структуре html. С такими локаторами, обычно, тесты – более *стабильные*. Обычно, это упрощает их поддержку, особенно учитывая то, что автоматизация пользовательского интерфейса веб-приложений сама по себе не очень "легковесна" (такие тесты сложнее и медленнее чем те же юнит-тесты или API-тесты).

При этом, на проектах, где автоматизация используется в первую очередь тестировщиками для полного контроля всего процесса тестирования, и где программисты, к сожалению, не так сильно заинтересованы в поддержке и использовании автотестов – "более строгие локаторы" могут быть полезны тем, что своей "нестабильностью" в тестах – всегда будут показывать зоны в приложении – "где что-то изменилось". Даже несмотря на то, что покрытая тестом фича работает, возможно, перестало работать что-то другое, с ней связанное. Ведь автотесты не могут заменить полностью ручное исследовательское тестирование, их покрытие намного слабее. Поэтому тестировщики могут использовать такую "хрупкость автотестов", как сигнал к более тщательному ручному тестированию в соответствующих "зонах приложения".

Если же автотесты используются разработчиками, с целью проверять, не поломали ли они те функциональные аспекты приложения, которые точно работали ранее – такие "строгие локаторы" – наоборот, будут приносить много проблем, и задержек. На самом деле, главная цель автотестов – как раз помогать разработчикам, давать им как можно более быстрый фидбек об их изменениях, а не уменьшать время ручной

регрессии для мануальных тестировщиков. При таком подходе автоматизация намного более эффективна, ведь влияет на обеспечение качества продукта на более ранних стадиях его разработки. Это то, к чему стоит стремиться. Поэтому "более строгие" локаторы – это не то, чего стоит "желать" на своем проекте. Но... Все зависит от контекста и от конкретного проекта;)

Примеры разного вида таких локаторов можно найти в соответствующих разделах списка частых ошибок и их решений (FAM) ниже.

Локаторы, автоматически сгенерированные в инспекторе, или подобранные вручную?

Тема хрупкости и стабильности локаторов также актуальна в контексте способа подбора локаторов. Их можно подбирать вручную, а можно получить уже готовый сгенерированный локатор в инспекторе браузера (Browser: Developer Tools/Inspector). Второй способ может быть быстрее, особенно в приложениях со сложной структурой элементов, но может выдавать и более "строгие локаторы", и от того – более хрупкие, и часто - намного менее читабельные. Это может привести к большим затратам в поддержке автотестов при условии, что структура html будет часто меняться на проекте. А если не будет? Тогда, возможно, действительно можно повысить свою эффективность, не заморачиваясь ручным подбором локаторов, и быть намного более быстрым при их подборе с помощью инспектора. А чтобы хоть как-то решить проблему читабельности – можно выносить такие сгенерированные локаторы в переменные, и уже их использовать в автотестах. Все, как всегда – зависит от условий на реальных проектах. Эта тема неплохо раскрыта в статье Сергея Пирогова "Мой взгляд на хорошие локаторы" и моих комментариях к ней ;)

В любом случае, стоит освоить оба способа – и иметь их в своем арсенале.

Когда выносить локаторы в отдельные переменные и/или классы?

Во-первых, давай не будем спешить;) Мы еще успеем навыносить и локаторы в переменные, и код действий над элементами в функции либо методы классов, и потом "скрыть" все эти "технические детали" в какой-то класс, и положить этот класс в укромное и удобное местечко в нашем проекте. Но сейчас не стоит этого делать осознанно, с целью "подождать момента, пока нам реально припечет", пока не увидим в этом еще большей нужды. Такой подход в контексте этого обучения - позволит нам

на практике осознать – когда действительно появляется эта нужда "что-то куда-то выносить" и чем это нам поможет. Поэтому пока давай "выжидать" и "собирать компромат" на эти "сырые" локаторы ;)

Но раз, вопрос появился, давай успокоим сердце некоторыми объяснениями, касающимися нашего случая.

Возможно самый яркий "компромат", который может бросаться в глаза – это уровень **читабельности** локаторов для "простых смертных". Если на проекте практикуется разделение между мунальными тестировщиками и автоматизаторами, то, например, мануальным тестировщикам, что-бы проверить, покрыли ли автоматизаторы то, что было нужно в автоматизированном сценарии – придется сначала разбираться, что такое CSS или XPath. Но если, автоматизаторы не использовали XPath, то разобрать синтаксис CSS не так уж сложно, и в любом случае полезно. И если автотест будет выглядеть в стиле оригинального решения на Selene из урока, и атрибуты использованные для нахождения элементов читабельны сами по себе, то после небольшого знакомства и консультаций с автоматизаторами такие сценарии должны быть вполне понятными:

```
browser.open("http://todomvc.com/examples/emberjs/");

browser.element('#new-todo').setValue('a').pressEnter();
browser.element('#new-todo').setValue('b').pressEnter();
browser.element('#new-todo').setValue('c').pressEnter();
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'));

browser.all('#todo-list>li').element_by(have.exact_text('b'))\
    .element(".toggle").click();
browser.all('#todo-list>li').filtered_by(have.css_class("completed"))\
    .should(have.exact_texts('b'));
browser.all('#todo-list>li').filtered_by(have.no.css_class("completed"))\
    .should(have.exact_texts('a', 'c'));
```

Второй компромат - это **повторение кода**. Обрати внимания на куски вида `element('#new-todo')` или `all('#todo-list>li')` - они довольно часто повторяются в сценарии. Можно было бы пожаловаться на то что такие куски каждый раз медленней набирать чем имена "заготовленных переменных". Но в реальной жизни, скорее всего ты будеш просто дублировать целые строки кода с помощью какой то

комбинации гарячих клавиш (например `Command + D` в PyCharm IDE под Mac OS), и редактировать соответствующие места. Можно было бы тогда пожаловаться на то, что если что то в таких повторяющихся кусках поменяется, собственно часть самого локатора - то прийдется менять во всех местах (вспомни принцип DRY). Но и в этом случае проблема может неплохо решаться обычной функцией редактора "найти и заменить";) Предпочтение таким вот "более быстрым и простым" подходам соответствует следованию принципу KISS ("Keep It Simple, Stupid"). Конечно, все принципы и подходы имеют свой контекст применения. Все зависит от условий, и в определенный момент мы можем понять, что следование тому же принципу DRY и соотвтествующее вынесение локаторов в переменные - теперь более предпочтительно. Главное, как и в обычной жизни, стараться не сильно спешить, "что-бы людей не насмешить";). Это особенно важно если мы начинаем проект, который может оказаться не "шаблонным". Последние пожелания, кстати, отображает принцип YAGNI ("You Aren't Gonna Neede It"), который обычно всегда идет рука об руку с принципом KISS;)

Все вышеупомянутые "упрощения" особенно актуальны именно для программирования в сфере тестирования, то есть в написании авто-тестов. Это объясняется тем, что

- тестов очень много
- их пишут разные люди
- тест, вероятно, будет прочитан человеком, который его не писал
- а тесты часто требуют повышенного внимания людей и их быстрой реакции, потому что могут часто падать

Поэтому так важно, чтобы тесты были реализованы как можно проще, с минимумом усилий, с минимумом использований сложных конструкций языка, в том числе и таких безобидных на первый взгляд, как переменные, функции, классы и их методы. Потому что разбираться со всем этим инженеру, который все это не писал, а теперь нуждается в быстром исправлении падающего теста – нет особого времени. И главное, что в силу небольшой сложности тестовых сценариев в сравнении с сложностью реализации внутренней логики продукта – тесты действительно реально сделать простыми.

Единственное, стоит учитывать, что "простота" должна распространяться не только на "реализацию" тестов, но и на их "использование" – то есть в частном случае "их чтение", а значит и "понятность". Причем, поскольку последняя избушка ("чтение") – как раз к нам передом, а "к лесу" задом – то она и является главным маркером, контролирующим "уровень простоты реализации". Если для наших конечных пользователей тестов – тесты и так понятны и читабельны – то зачем вводить новые "абстракции" – переменные, функции, классы – при этом усложняя реализацию? Если же наоборот – читать тесты сложно, сложно быстро распознать за кодом теста тестовую логику – тогда наоборот, стоит подумать об использовании соответствующих абстракций. Ко всему этому, и ко всем упомянутым выше принципам - мы еще вернемся в одном из следующих разделов. А сейчас поиграемся в продвинутых инженеров, кому и так все понятно, чтобы ощутить разницу и приобрести важный опыт;)

Как найти элемент из списка однотипных элементов по нужному тексту его внутреннего элемента (CSS)?

> Допустим, есть список `<li>` элементов, внутри каждого из которых есть "лейблы" (`<label>`) с разными значениями – я не понял как сделать поиск по значениям этих "лейблов"... Точнее, сами лейблы-то с такими значениями найти можно, а вот именно чтобы найти нужный элемент `li` , у которого есть "лейбл" с определенным значением – не вышло. Пришлось делать через нахождение элемента по индексу.

Важно понимать, что все, что находится `<label>между открытым и закрытым тегом элемента</label>` технически считается текстом элемента, а не "значением". Значениями корректней называть "значения атрибута value некоторых элементов, таких как `<input>` ".

Так вот, дело в том, что CSS селекторы, действительно, не умеют искать элементы по тексту, и тем более не умеют выбирать нужный элемент из списка по какому-либо признаку (будь-то "значение" или "текст"), присущему одному из его внутренних элементов. Вот с помощью xPath селекторов это сделать можно, ну и, конечно же, это можно сделать с помощью некоторых "команд" Selene, описанных в "Selene в действии", которые в этом задании использовать запрещено;)

Так что, если речь идет только об использовании CSS селекторов, все что остается – это искать элемент по индексу.

Список частых ошибок

Структурная информация в именах тест-сьюта и тестов, которая не несет пользы с точки зрения тестирования

Пример 1

```
# tests/test_todomvc_css_version.py
# ...

def test_completes_task():
```

Пример 2

```
# tests/test_todomvc.py
# ...

def test_completes_task_css_version():
```

Пример 3

```
# tests/test_todomvc_css_version.py
# ...

class TestTodoMvcCssVersion:

    def test_completes_task():
```

Пример 4

```
# tests/test_todomvc.py
# ...

class TestTodoMvc:

    def test_completes_task_css_version():
```

Имена тест-модуля (тест-класса) и тест-функций (тест-методов) играют, по сути, роль имен "тест-сьюта" и "тест-кейсов" соответственно, а последние – должны отображать то, что мы тестируем, а не, например, "нюансы реализации", как в примерах выше. Если хочется структурировать свои решения для заданий – лучше создать отдельные папки (пакеты python) и разложить в них, как по полочкам – разные версии решений.

Например так... (ищи решение на следующей странице)

... минималистично:

```
selene-intro/
  ...
  tests/
    css_task/
      test_todomvc.py
    xpath_task/
      test_todomvc.py
    test_google.py
    test_todomvc.py
  ...
```

или еще более организованно:

```
selene-intro/
  ...
  tests/
    lesson/
      test_google.py
      test_todomvc.py
    tasks/
      css/
        test_todomvc.py
      xpath/
        test_todomvc.py
  ...
```

или еще и с фокусом на том что мы тестируем (а не на том что "учимся'):

```
selene-intro/
  ...
  tests/
    google/
      ..
    todomvc/
      full_selene_version/
        test_todomvc.py
      css_version/
        test_todomvc.py
```

```
      xpath_version/
        test_todomvc.py
  ...
```

Или еще и избегая тафтологии для большей ясности ;)

```
selene-intro/
  ...
  tests/
    google/
      ..
    todomvc/
      full_selene_version/
        test_operations.py
          ...
          def test_complete_task()
      css_version/
        test_operations.py
      xpath_version/
        test_operations.py
  ...
```

Несоответствие имен тест-сьюта или тест-кейсов целям тестирования

Пример 1

```
# tests/test_todomvc.py
# ...
def test_todomvc:
```

Пример 2

```
# tests/test_todomvc_compete.py
# ...
def test_completes_task:
```

Пример 3

```
# tests/test_todomvc.py
# ...
class TestTodoMvc:
    # ...

    def test_todomvc:
```

Пример 4

```
# tests/test_todomvc.py
# ...
class TestTodoMvcCompletes:
    # ...

    def test_completes_task:
```

Имена тест-модуля (тест-класса) и тест-функций (тест-методов) играют, по сути, роль имен "тест-сьюта" и "тест-кейсов" соответственно, а последние – должны отображать то, что мы тестируем. В первом примере выше, имя тест-функции не несет совсем никакой полезной информации с точки зрения целей тестирования для нашего сценария. Во втором примере имя тест-модуля недостаточно абстрактно в контексте

нашего покрытия (мы как раз "завершение" задачи покрываем в тест-функции). Последние два примеры аналогичны первым дувм, только в них используется стиль тест-классов/тест-методов вместо тест-модулей/тест-функций.

Решение...

Пример 1-2

```
# tests/test_todomvc.py
# ...
def test_completes_task:
```

Пример 3-4

```
# tests/test_todomvc.py
# ...
class TestTodoMvc:
    # ...

    def test_completes_task:
```

Установка пути к chromedriver в коде

Пример

```
# ...
from selenium import webdriver

options = webdriver.ChromeOptions
options.binary_location = '/opt/bin/chromedriver'
browser.config.driver = webdriver.Chrome(chrome_options=options)
```

Обычно, намного проще и универсальней **прописать путь к chromedriver в системную переменную окружения PATH** (возможно, чтобы изменения вступили в силу, придется перегрузить OS). Такое решение лучше по той причине, что тесты могут быть запущены на разных машинах, и если мы пропишем точный путь в коде тестов к chromedriver – нам придется на всех машинах подгонять местоположение драйвера к этому пути. А что делать, если мы захотим запустить тесты под другой OS, где и пути-то могут быть совсем в другом формате? ;) Поэтому лучше переложить обязанность за "установку пути драйвера" с кода тестов на конфигурации машин для запуска тестов.

Как установить PATH? – можно загуглить решение конкретно под нужную OS;)

Использование "запрещенных в задании методов Selenide"

Пример

```
browser.element('#todo-list>li:nth-child(2)').element('.toggle').click()
```

В этом задании мы учимся находить элементы только силами CSS. В заданиях указано, какие типы методов нельзя использовать;)

В примере выше использован метод `Element#element` (метод `element`, вызванный для объекта класса `selene.core.Element`, который возвращает метод `element` объекта `browser`) – он позволяет найти внутренний элемент по селектору внутри другого, а наша задача научиться такому "внутреннему поиску" только с помощью CSS селекторов в этом задании;)

Решение...

```
browser.element('#todo-list>li:nth-child(2) .toggle').click()
```

Хрупкие строгие локаторы

Обычно есть несколько способов написать локатор что-бы найти один тот же элемент на странице. Можно использовать очень много информации о структуре html (вложенность элементов, порядок их следования один за другим) и содержании (тексты, значения атрибутов элементов) и тогда получится "строгий" локатор. Или использовать минимум информации - и тогда получится "слабый" или "гибкий" локатор. В следующей серии ошибок мы расмотрим типичные случаи когда строгие локаторы приводят к нестабильным "хрупким" тестам (flaky tests).

Хрупкие строгие локаторы. Избыточность в путях - привязка к тегу

Допустим у нас есть элементы:

```
<li>
  <div class="crd">
    <label>1</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
<li>
  <div class="crd">
    <label>2</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
...
```

Тогда следующий селектор вероятно будет "хрупким":

```
li:nth-child(2) input.destroy
```

Для точного определения элементов лучшей практикой в разработке внутреннего представления html страниц принято использовать атрибуты элементов, в большинстве случаев - `id` (для определения уникальных элементов) и `class` (для

определения группы однотипных элементов). Поэтому использовать теги элементов особенно когда элемент можно определить по css классу - менее предпочтительно. В нашем случае внутрення реализация может например поменятся от:

```
<input type="button" class="destroy">
```

до

```
<button class="destroy"></button>
```

В таком случае наш селектор перестанет работать и тест упадет. Хотя функционал по сути не поменялся - как была кнопка удаления задачи так и осталась. То есть функциональный тест упасть не должен был бы.

Решение:

```
li:nth-child(2) .destroy
```

Хрупкие строгие локаторы. Избыточность в путях - привязка к точному пути

Допустим у нас есть элементы:

```
<li>
  <div class="crd">
    <label>1</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
<li>
  <div class="crd">
    <label>2</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
...
```

Тогда следующий селектор вероятно будет "хрупким":

```
li:nth-child(2)>div>.destroy
```

С развитием проекта внутренняя структура html может усложнятся. И там где сейчас у нас есть два "родительских элемента" по отношении к тому элементу что мы ищем:

```
<li>
  <div class="crd">
    <label>2</label>
    <input type="button" class="destroy"/>
  </div>
  ...
</li>
```

со временем может появится 3:

```
<li>
```

```
    <div>
      <div class="crd">
        <label>2</label>
        <input type="button" class="destroy"/>
      </div>
    </div>
    ...
  </li>
```

В таком случае наш селектор перестанет работать и тест упадет. Хотя функционал по сути не поменялся - как была кнопка удаления задачи так и осталась. То есть функциональный тест упасть не должен был бы.

Решение...

```
li:nth-child(2) .destroy
```

Хрупкие строгие локаторы. Привязка к полному значению аттрибута class либо его начала или конца

Допустим у нас есть элементы:

```
<li class="end">1</li>
<li class="middle end">2</li>
<li class="end">3</li>
```

И нам нужно уметь отдельно находить второй элемент (с текстом 2) и отдельно оба - первый (1) и третий (3). Мы можем увидеть что их отличает значение атрибута `class` . У второго элемента это значение начинается с текста `"middle"` , а у первого и третьего - целое значение равно `"end"` .

Соответственно мы можемо построить и локаторы соответственно:

- найти элемент(ы) у которого значение аттрибута класс начинается с текста `"middle"`

```
[class^='middle"]
```

- найти элемент(ы) у которого целое значение аттрибута класс равно тексту `"end"`

```
[class='end"]
```

Аналогичные примеры будут и для XPATH селекторов. Иногда такой подход единственное что остается. Но в случае когда мы работаем с атрибутом `class` стоит быть осторожным. Дело в том что новые CSS классы склонны появляться у элементов время от времени. И если, как в примере выше, мы намертво привяжемся к полному значению атрибута `class` , либо, его началу, то если вдруг програмисты решать повесить на наши элементы еще один цсс класс, например `"start"` :

```
<li class="start end">1</li>
<li class="start middle end">2</li>
<li class="start end">3</li>
```

то нашы локаторы "сломаются" и перестанут находить нужные нам элементы. При этом фича, которую мы должны были покрывать в нашем тесте где мы использовали эти локаторы – все еще должна работать, и, соответственно, результат теста – будет не показательным с точки зрения его "цели тестирования". Такая хрупкость локаторов может быть нежелательна на проектах где могут иметь место частые изменения такого вида. В таких случаях стоит поискать более "гибкий" и "стабильный" локатор.

Обычно можно построить локатор через поиск по вхождению определенного текста в значение атрибута, или даже по вхождению "слова" в значение аттрибута. В CSS селекторах конкретно в случае поиске по CSS классу для этого даже есть специальное сокращение. Когда же нужно исключить из поиска элементы с значением аттрибутов имеющим определенные слова - и для этого и в CSS и XPATH селекторах есть соответствующие модификаторы поиска.

Решение...

- найти второй элемент

  ```
  [class~='middle"]
  ```

  или лучше:

  ```
  .middle
  ```

- найти все элементы кроме второго

  ```
  :not(.middle)
  ```

Хрупкие гибкие локаторы. Привязка к частичному значению атрибута class

Иногда с "гибкостью" локаторов можно перестараться, и в результате получится не стабильный а хрупкий локатор.

Допустим у нас есть html код:

```
<li class="foo bar">1</li>
<li class="foo">2</li>
<li class="foo bar">3</li>
```

Нам нужно уметь находить элементы с классом `bar`. И вот мы строим наш локатор по принципу:

- найти элемент(ы) у которых значений атрибута `class` содержит (contains) текст `"bar"`.

```
[class*="bar"]
```

И тут девелоперы в какой то момент осознали, что есть особенный вид элементов `foobar` (привет утконосам из Австралии;)).

```
<li class="foo bar">1</li>
<li class="foo">2</li>
<li class="foo bar">3</li>
<li class="foobar">4</li>
```

И вуаля, локатор который был предназначен для нахождения только "bar" элементов - теперь находит и элементы "foobar".

Поэтому когда мы работаем с определением элементов исходя из значений атрибута class - обычно лучше искать "по CSS class-у", то есть вхождению соответствующего "**целого** слова" в значение атрибута (учти, что *CSS class* - это одно из слов в значении атрибута `class`, а *class* - это имя одноименного атрибута)

Решение...

```
.bar
```

Построение локатора с помощью менее читабельных и понятных аттрибутов либо их значений

Пример

```
browser.all('#todo-list li[class="ember_view"]')\
    .should(have.exact_texts('a', 'c'))
```

Главная проблема в этом коде - это потенциальная хрупкость локатора как следствие привязки к целому значению аттрибута class.

Но есть и другая ошибка. Для построения локатора было выбрано значение `"ember-view"` - которое абсолютно ничего не говорит о том - о чем этот локатор. Этот локатор должен искать "активные задачи". А ищет какие-то "задачи с ember-view `o_O` ". Более, того, наше приложение построено на основе фронтенд-фреймворка "EmberJs" (вспомни каким был URL по которому мы открывали приложение). Зная это - можно догадаться что этот css-class `ember-view` еще и автосгенерирован скорее всего... А все что генерируется автоматически и по сути является "динамически изменяемым" - явно не то что будет "стабильным", а значит не то на что стоит полагаться при построении стабильного "не хрупкого" локатора. Иногда выбора нет, и более понятный локатор не построешь. Тогда что бы код все еще был понятным и его было просто поддерживать в будущем - используют переменные с более четким именем. Но в этом случае, можно подобрать намного более читабельный и даже более лаконичный локатор. Также не забываем, что в этом задании, мы не используем конструкции типа переменных или функций;)

Подсказка 1...

Сфокусируйся на том что "что должен найти локатор". А найти он должен "активные задачи". Каким другим способом можно перефразировать "активность" задачи?

Подсказка 2...

В этом случае можно построить более простой локатор исходя из того что "активные задачи" - это тоже самое что **"незавершенные задачи"**.

Решение...

```
browser.all('#todo-list li:not(.completed)')\
    .should(have.exact_texts('a', 'c'))
```

Слишком краткие и потому менее информативные, неочевидные локаторы

Часто с целью сделать локатор покороче, мы не учитываем о том что краткость может навредить читабельности, и сделать тесты более сложными в восприятии (мы еще вернемся к этой теме в следующих уроках - смотри раздел о читабельности и лаконичности...)

Если тест откроет человек который его не писал, и увидит там код с локатором вида:

Пример 1

```
browser.element('li:nth-child(2) .toggle').click()
```

... ему может быть не очевидно что `li` это элемент списка задач а не какого то другого списка.

В следующем коде также "идет речь о элементах списка задач" но при этом локатор никак об этом не "информирует":

Пример 2

```
browser.all('.completed').should(have.exact_texts('b'))
```

Решение...

Пример 1

```
browser.element('#todo-list li:nth-child(2) .toggle').click()
```

Пример 2

```
browser.all('#todo-list li.completed').should(have.exact_texts('b'))
```

Неконсистентные селекторы

Пример

```
# ...
browser.all('#todo-list li').should(have.exact_texts('a', 'b', 'c'))

browser.element('#todo-list>li:nth-of-type(2) .toggle').click()
# ...
```

Важно быть как можно более последовательным и единообразным в построении локаторов. Потому что чем больше мы сделаем код "разнообразным" – тем сложнее его будет понимать другим, тем менее читабельными будут тесты, тем больше у других членов команды будет уходить времени на поддержку тестов. А в тестах очень важна читабельность, учитывая то, что их очень много, их пишут разные люди и когда тесты падают – нужно как можно быстрее понять что произошло, найти и разобраться с причиной.

В коде выше для нахождения одних и тех же сущностей – элементов задач – используются разные селекторы:

- `#todo-list li` (ищет все элементы `li` внутри элемента `#todo-list` на любой глубине вложенности)
- `#todo-list>li` (ищет все элементы `li` внутри элемента `#todo-list` на первом уровне вложенности)

Это будет сбивать с толку во время разбора кода теста тех, кто его не писал. Ведь первый сигнал который поступит в мозг от глаз будет - "это разные селекторы". И только потом, затратив время на обработку - мозг поймет что это один и тот же селектор. Зачем же утруждать его лишней работой? ;)

Еще один минус в том, что при глобальной замене этой части селектора на какую то другую - не получится заменить все вхождения сразу, как раз за счет того что эти части селектора - неконсистентны. Конечно, можно сказать что если бы мы вынесли эту часть в переменную - то и проблемы такой не было бы. Но контексты могут быть разными на разных проектах, и не всегда может быть полезно использовать переменные.

Решение...

либо:

```
# ...
browser.all('#todo-list>li').should(have.exact_texts('a', 'b', 'c'))

browser.element('#todo-list>li:nth-of-type(2)').element('.toggle').click()
# ...
```

либо:

```
# ...
browser.all('#todo-list li').should(have.exact_texts('a', 'b', 'c'))

browser.element('#todo-list li:nth-of-type(2)').element('.toggle').click()
# ...
```

Первый вариант более строгий, поэтому если на проекте велика вероятность, что появятся другие элементы `li` внутри `#todo-list` (например глубже...), то такой селектор будет менее "стабильным".

Второй вариант менее строгий, поэтому если вдруг текущие элементы `li` вследствие каких-то изменений "опустятся ниже" – он все еще будет работать. Такой селектор может быть более "стабильным" и "живучим".

Лучший вариант выбрать между двух невозможно, все зависит от проекта ;) Но можно начать с второго, а там видно будет ;)

Другие примеры ошибок...

```
browser.all('#todo-list li').should(have.exact_texts('a', 'b', 'c'))
browser.element('li:nth-child(2) .toggle').click()
browser.element('.completed').should(have.exactText('b'))
browser.all('#todo-list [class|=ember-view]').should(have.exact_texts('a',
'c'))
```

Определившись с локатором для поиска "лист-айтема" (`li`) списка задач (`todo-list`):

```
browser.all('#todo-list li').should(have.exact_texts('a', 'b', 'c'))
```

... уже в следующей строке кода для отображания того самого элемента списка мы используем более другой, пускай более короткий, но "неконсистентный" и менее информативный локатор:

```
browser.element('li:nth-child(2) .toggle').click()
```

... а теперь мы вообще забыли хоть как то упомянуть в локаторе о том что мы работаем с задачами:

```
browser.element('.completed').should(have.exactText('b'))
```

... и последней вишенькой на тортике, мы решили "для разнообразия" в этот раз опустить `li`

```
browser.all('#todo-list [class|=ember-view]')\
    .should(have.exact_texts('a', 'c'))
```

А здесь кто-то решил побеспокоится о других разработчиках которые этот код не писали, что-бы как только они открыли код упавшего теста - глаза сразу увидели сквозь весь код где в каких строчках ведется работа с одними и теми же сущностям - `todo-list li` :

```
browser.all('#todo-list li').should(have.exact_texts('a', 'b', 'c'))
browser.element('#todo-list li:nth-child(2) .toggle').click()
browser.element('#todo-list li.completed').should(have.exactText('b'))
browser.all('#todo-list li*[class|=ember-view]*')\
    .should(have.exact_texts('a', 'c'))
```

При этом одна ошибка все еще осталась в последней строке (виделенная курсивом), но о ней в другой раз;)

Фэншуй

Есть в программировании вещи, которые могут показаться неоправданными усилиями ради какого-то "сомнительного фэншуя". Например "отступы в коде", "дополнительные пустые строчки", подбор читабельных понятных имен, и т. д. Но, на самом деле, чем больше растет количество написанного кода в проекте, тем он более становится похожим на сложный организм человека, в котором при проведении операций хирургу непростительно быть "неаккуратным", да и операции проще проводить в теле с более-менее хорошим состоянием. Поэтому стоит задуматься о том, чтобы выработать у себя некоторые "задротские но здоровые привычки" ;)

Хорошая новость в том, что большинство таких нюансов подсказывает IDE, PyCharm так точно хорошо с этим справляется (просто будь внимателен ко всему что он подсвечивает), но и другие редакторы при соответствующих настройках хороши. Плюс сам питон с его зависимостью от отступов - навязывает быть более акуратным.

Фэншуй. Лишние пустые строки

```
from selene import have
from selene.support.shared import browser



- -
def test_completes_task:
        - -
        - -
        browser.open('http://todomvc.com/examples/emberjs')

        - -
        # ...code

        # ...code
```

Лишние пустые строки, на первый взгляд, достаточно безобидны, но если их использовать "с умом" – то с их помощью можно также выделять структуру в коде, например – блоки связанных шагов и проверок в коде тест-функции (~ "тест-кейса'). Да и основные части кода, такие как методы – также удобно отделять друг от друга пустыми строками, для "усиления" выразительности кода. И если где-то их использовать "с умом", а где-то "как попало", то от "усилий ума" толку не будет никакого;)

Решение...

```
from selene import have
from selene.support.shared import browser


def test_completes_task:
        browser.open('http://todomvc.com/examples/emberjs')

        # ...code

        # ...code
```

Задание: Selene и XPath селекторы

Особенность Selene в том, что при его использовании, обычно, не нужны сложные CSS или XPath селекторы. Достаточно самых простых селекторов, которые ищут элементы по атрибутам, например:

```
browser.element(by.name('q'))  # ...

browser.element(by.id('new-todo'))  # ...
# или еще проще для последнего примера:
browser.element('#new-todo')  # ...
```

Более же сложные сложные варианты поиска элементов достигаются использованием дополнительных "команд" Selene, таких как `Collection#element_by(condition)`, `Element#element(selector)`, которые позволяют построить более читабельный и потому более легкий в поддержке код:

```
browser.all('#todo-list>li').element_by(have.exact_text('b'))\
        .element('.toggle').click()
```

Но чтобы прочувствовать все это на собственном опыте, да еще и закрепить знания в XPath селекторах, твоим заданием будет реализовать сценарий из раздела "Selene в действии":

```
def test_complete_task():
    # open TodoMVC page

    # add tasks: "a", "b", "c"
    # tasks should be "a", "b", "c"

    # toggle b
    # completed tasks should be b
    # active tasks should be a, c
```

для приложения TodoMVC (emberjs version), не используя при этом следующих методов Selene:

- `SeleneElement#element` (то есть `browser.element(".parent").element(".child")`)
- `SeleneElement#all` (то есть `browser.element(".parent").all(".child")`)
- `SeleneCollection#[index]` (то есть `browser.all(".item")[1]`)
- `SeleneCollection#first`
- `SeleneCollection#element_by(Condition condition)`
- `SeleneCollection#filtered_by(Condition condition)`
- и других им подобных ;)

По сути, все, чем можно пользоваться - это:

- `browser.element(...).*`, где `*` - это действия над элементом или ассерты (`should`)
- `browser.all(...).should(...)`
- и XPath селекторами для составления локаторов, которые можно передавать просто как строки, например: `browser.element('//*[@name="q"]')`

Цель - написать как можно более близкий по аналогии к реализации на Selene код (из раздела "Selene в действии"). Например, если в версии Selene мы используем выбор элемента по тексту среди других элементов - то нужно построить максимально аналогичный XPath селектор;)

Все еще, как и для задания о CSS, не спеши создавать переменные и функции для хранения/переиспользования локаторов либо тестовых данных. Мы к этому еще вернемся в следующих заданиях. Сейчас важно двигаться шаг за шагом, детально прорабатывая темы на которых мы фокусируемся в каждом из уроков.

Дополнительные материалы:

- Рецепты CSS и XPATH
- Видео от automated-testing.info о том, как писать локаторы (CSS, XPath)

В особо трудную минуту, можно подглядывать за подсказками в следующую главу с FAQ. В нем же можно будет найти список частых ошибок по которому проверить свое финальное решение;) Удачи!

Решение: Список частых вопросов

Как в XPATH искать по одному CSS классу а не по всему значению соответствуеющего аттрибута?

CSS-селекторы хотя и менее универасальны чем XPath-селекторы, но более лаконичны и удобны, например поиск по одному css классу `toggle` реализуется очень просто:

```
.toggle
```

В случае XPath же нам придется извратиться, чтобы достичь той же цели:

```
browser.element('//*[contains(concat(" ", normalize-space(@class), " "), "
toggle ")]').click();
```

Можно улучшить ситуацию, создав дополнительный "хелпер-метод" (helper-method), но именно в этом задании, мы так далеко забегать не будем, всему свое время;) Уже в следующем задании мы сфокусируемся именно на этом, а пока - не спеши ;)

Как найти элемент из списка однотипных элементов по нужному тексту его внутреннего элемента (XPath)?

Первый же результат поиска в Google по

> xpath find element by text of its inner element

даст ответ ;)

Если не помогло...

Вся идея в том, что любое "уточнение поиска элемента в XPath селекторах" прописывается в квадратных скобках после элемента, например, для (следующий пример взят с stackoverflow):

```
<list>
   <book>
      <author>
         <name>John</name>
         <number>4324234</number>
      </author>
      <title>New Book</title>
      <isbn>dsdaassda</isbn>
   </book>
   <book>...</book>
   <book>...</book>
</list>
```

следующим образом можно уточнять, какая именно книга нужна:

```
//book[author/name='John']
```

или явно указывая, что ищем от "текущего места":

```
//book[./author/name='John']
```

или без точного полного пути (ищем элемент name на любой глубине вложенности внутри элемента book):

```
//book[.//name='John']
```

или без привязки к тегу внутреннего элемента:

```
//book[.//text()='John']
```

Или используя сокращенный синтаксис (хотя и менее читабельный):

```
//book[.//.='John']
```

Ниже приведены примеры, актуальные для домашнего задания...

версия в стиле Selene:

```
browser.all('#todo-list
li').element_by(have.exact_text('b')).element('.toggle').click();
```

упрощенная версия с XPath:

```
browser.element('//*[@id="todo-list"]//li[.//text()="b"]//*
[@class="toggle"]').click();
```

Это не полний аналог версии на Selene касательно вибора нужного чекбокса. Этот код будет работать, потому что у искомого элемента только один css class – "toggle", если бы было больше, а так в будущем может и случиться, то нужно искать более универсальное и стабильное решение. Этот тип ошибки уже расматривался в предыдущем задании. Селектор в котором мы привязываемся к точному значению атрибута class - хрупкий. Один из вопросов этого FAQ отвечает на вопрос как правильно искать по css-классу в xpath-селекторах, но еще быстрее и полезней такие вещи находить через google;)

Решение: Список частых ошибок

Большинство вопросов и ошибок для решений задания "Selene и CSS селекторы" будут актуальны и для текущего задания "Selene и XPath селекторы". Поэтому обязательно просмотри их и учти в контексте проверки своего решения;) Особенно обрати внимание на:

- хрупкость локаторов
  - привязку к тегам
  - привязку к точным путям
  - привязку к точному значеныию атрибута class
  - привязку к частичному значению атрибута class
- привязку к нечитабельным атрибутам
- не точный аналог коду показаному в разделе "Selene в действии"

Все это уже должно быть пройденным этапом, но жизнь показывает что повод к повторению, которое "мать учения" - есть всегда ;)

Хрупкие строгие локаторы. Избыточность в путях - привязка к тегу

Ошибки, типичные и для задания о CSS-селекторах;)

Пример 1

```
browser.element('//input[@id="new-todo"]').setValue('a')
```

Пример 2

```
browser.element('//*[@id="todo-list"]/li[2]//input[@class="toggle"]')
```

Пример 3

```
browser.element('//ul[@id="todo-list"]/li[2]//*[@class="toggle"]')
```

Пример 4

```
browser.element('//*[@id="todo-list"]/li[2]//input')
```

Последний локатор еще более хрупкий чем предыдущие, поскольку совсем не использует уникальные атрибуты.

Решения...

Пример 1

```
browser.element('//*[@id="new-todo"]')
```

Пример 2

```
browser.element('//*[@id="todo-list"]/li[2]//*[@class="toggle"]')
```

Пример 3

```
browser.element('//*[@id="todo-list"]/li[2]//*[@class="toggle"]')
```

Пример 4

```
browser.element('//*[@id="todo-list"]/li[2]//*[@class="toggle"]')
```

Двойные кавычки внутри двойных кавычек

Пример

```
browser.element("//*[@id=\"new-todo\"]")
```

Всегда стоит помнить о читабельности кода, ведь это прямым образом влияет на поддержку этого кода в будущем, особенно людьми которые его не писали изначально. В даном случае "бэкслеши" явно ухудшают читабельность, а ведь их можно избежать используя другой тип "внутренних" кавычек по отношению к "внешним", который также будет валидным;)

Решение...

```
browser.element('//*[@id="new-todo"]')
```

При этом способ выше предпочтительней чем:

```
browser.element("//*[@id='new-todo']")
```

Ведь одинарные кавычки как "выбор номер один" для строк - лучше двойных, поскольку:

- они занимают "меньше пространства", а следовательно глазам проще "считывать" полезную информацию в незагроможденном коде
- их банально проще и быстрей печатать (не нужно зажимать shift)

Таким образом, если везде мы используем одинарные кавычки, то консистентней везде подстраиваться под это правило;)

Использование `position()` для поиска элемента по индексу

```
browser.element('//*[@id="todo-list"]/li[position()=2]//*[@class="toggle"]')
```

Всегда стоит стремиться к лаконичности, если это не ухудшает читабельности. В примере выше использование `position()=` избыточно. В XPath есть сокращенный синтаксис, который полностью очевидный всем, поскольку повсеместно используется в программировании.

Решение...

```
browser.element('//*[@id="todo-list"]/li[2]//*[@class="toggle"]')
```

Поиск элемента по индексу вместо поиска по тексту

```
browser.element('//*[@id="todo-list"]/li[2]//*[@class="toggle"]')
```

Во-первых, в задании было указано для решения на XPath использовать код, как можно более близкий к версии локаторов в стиле Selene:

```
browser.all('#todo-list>li').element_by(have.exact_text('b'))\
    .element('.toggle')
```

Во-вторых, выбор элемента среди списка элементов именно по тексту имеет два преимущества:

- код будет более читабельным и информативным – не нужно будет лишний раз задумываться – какая там у нас была задача под индексом `2`. Это было бы особенно актуально, если бы у нас в сценарии ранее были шаги, которые, например, удаляют некоторые задачи - соответственно индексы задач будут меняться со временем, что может еще больше запутывать.
- код будет ближе к тому как взаимодействует с приложением пользователь. Если мы хотим "завершить" задачу как пользователь – мы ищем ее именно по тексту глазами в списке, а не помним, по какому номеру она была создана, и отсчитываем все задачи от начала, пока найдем нужную;)

Подсказки, как выбрать нужный элемент по тексту, есть в FAQ к этому же заданию.

Решение...

```
browser.element('//*[@id="todo-list"]/li[.//label='b']//*[@class="toggle"]')
```

или, возможно, чуть лучше, потому что "менее строго и потому хрупко" (а вдруг label поменяется на какой-то другой элемент?):

```
browser.element('//*[@id="todo-list"]/li[.//text()='b']//*[@class="toggle"]')
```

или, используя сокращенный синтаксис, хотя, возможно, чуть менее очевидный:

```
browser.element('//*[@id="todo-list"]/li[.//.='b']//*[@class="toggle"]')
```

Задание: XPath DSL

В задании "Selene и XPath-селекторы" ты должен был убедиться что XPath-селекторы, будучи более мощными чем CSS-селекторы, при этом - более сложные, многословные и громоздкие, и как следствие - менее читабельные. Особенно это касается поиска по css-классам, что довольно важно с точки зрения построения стабильных локаторов.

Если так получилось, что в каких то случаях "без XPath" никак, то конечно же стоит что-то придумать что бы улучшить читабельность селекторов, следуя подходу само-документируемого кода. В этом могут помочь дополнительные хелпер-функции, которые скроют всю сложность внутри, и будучи соответственно именованными предоставят пользователю читабельный API к построению XPath-селекторов. Это и будет целью этого задания.

Какие именно функции нужны и сколько - подумай сам. Постарайся быть достаточно экономным в этом вопросе. Вдруг что, заглядывай в подсказки следующего за заданием списка частых вопросов;)

Удачи!

Решение: Список частых вопросов

Где лучше сохранить вспомогательные функции для построения сложных XPath селекторов? (Стратегия сохранения методов)

Программирование очень похоже на реальную жизнь, серьезно:) В нем нет никаких секретов, которые бы не относились к обычному быту. Любая хорошая домохозяйка, которая умеет убираться и превращать хаос в порядок, у которой на кухне все по "феншую" – имеет все возможности стать прекрасной программисткой! А все потому, что одна из главных стратегий в программировании – **"раскладывать яйца по своим корзинам"**, или**"раскладывать чашки/тарелки по своим полочкам"**.

Разные функции, но в небольшом количестве, в самом самом начале? – допустимо один модуль

Как и в реальной жизни, в программировании, в самом самом начале, "яиц" может быть не так много, чтобы раскладывать их по разным корзинам. Если у нас всего одно гусиное яйцо, одно куриное белое, и одно куриное красное, и кроме яиц больше никаких других продуктов нет – нам незачем под каждый "вид" выделять свою корзину. Весь этот "натюрморт" будет выглядеть достаточно "феншуйно" в одной корзине, и по надобности – будет достаточно удобно сразу бежать к одной корзине, а не перебирать множество.

Кардинально отличные по использованию функции? – отдельный модуль

При этом, если вдруг у нас появляется что-то кардинально отличное от яиц, например столовые приборы – наверное, таки сразу стоит им выделить отдельный ящичек, или хотя бы "стаканчик", если их немного.

Много функций одного вида? – отдельный модуль

Как только яиц определенного типа будет становиться много – уже станет сложней в одной куче находить нужные, и тут уже понадобится отдельная корзина, например, под куриные яйца.

Очень много функций одного вида? – выделяем подвиды и раскладываем по отдельным модулям, которые группируем в выделенную под них под-папку (пакет)

Как только яиц определенного вида станет так уж много, что одной корзины им будет маловато (будет сложно копаться, и быстро находить то что нужно) – придется выделить еще парочку, и чтобы как-то "организовать и структурировать" такие яйца – будет удобно выделить их подвиды – и хранить, например, красные яйца отдельно от белых, большие от маленьких, более свежие от менее свежих... И, конечно же, ни одна хозяйка не станет раскладывать корзины с яйцами по разным углам на кухне – она положит их все в одно место.

Также и в программировании стоит группировать модули/классы разных подвидов одного вида внутри одного пакета (папки). В python-проектах обычно для структурирования кода есть два "корневых" отдельных шкафа (имена могут быть другими) – папка `‹PROJECT_NAME›` (с именем самого проекта, главный корневой модуль) и папка `tests` – по которым можно раскладывать инструментарий соответствующего контекста использования. Все, что касается тестирования продукта/библиотеки – идет в `tests`, а все, что касается самого продукта/библиотеки – в `‹PROJECT_NAME›`. Если же проект только о тестировании, то в `‹PROJECT_NAME›` обычно складывают ту часть проекта, которую можно будет переиспользовать в других проектах, и, возможно, даже можно будет вынести в отдельную независимую библиотеку. Еще часто в таких "тест-проектах" в `‹PROJECT_NAME›` складывают не сам продукт, а его "модель" (в контексте тестирования пользовательского интерфейса – это могут быть модули или классы, описывающие объекты страниц веб-приложения, так называемые PageObjects).

Также, как и в быту уже есть общепринятые в употреблении слова для обозначения некоторых мест хранения, так и в программировании, например, пакеты с модулями, которые могут потенциально быть использованы не только в текущем проекте, но и в других – могут называть `common`, или `core`, если набор таких классов уже больше похож на потенциальную библиотеку, решающую определенные проблемы, а не просто набор вспомогательных инструментов.

Важность имен

Хорошая хозяйка также подготовит понятные таблички с надписями для каждой корзинки, чтобы если ее не будет дома, другие члены семьи смогли разобраться, что, где и как. Таблички будут однозначно описывать то, что в них лежит, будут понятными и читабельными для всех членов семьи, написанные на "общем для всех языке", использовать "общепринятые термины". Прочтя надпись, должно быть сразу

понятно, что лежит внутри. Надписи также будут по возможности достаточно краткими. Ни одна хозяйка на полке с приправами не будет на каждой из баночек писать: "Приправа: перец", "Приправа: карри", "Приправа: тмин" – и так ведь понятно что все, что лежит на полочке с надписью "приправы" – приправы и есть, достаточно на каждой из баночек просто написать название приправы: "Перец", "Карри", "Тмин"...

Какую именно часть длинного сложного XPath селектора стоит выносить в отдельные функции, а какие нет?

В первую очередь стоит сделать более читабельными именно те куски XPath, которые самые сложные и грозмоздкие. Главный кандидат - это кусок отвечающий за поиск или скорее даже "выбор нужного элемента среди всех" по CSS-классу, который в CSS-селекторах выглядит совсем просто, лаконично и удобно:

```
.toggle
```

А в случае XPath нам приходиться извращаться, чтобы достичь той же цели:

```
browser.element('//*[contains(concat(" ", normalize-space(@class), " "), "
toggle ")]').click();
```

Именно в этом месте стоит в первую очередь улучшить читабельность, создав дополнительную "хелпер-функцию" (helper-function):

```
browser.element('//*[' + x.has_css_class('toggle') + ']').click();
```

(в даном случае дополнительный хелпер, как видно, вынесен в отдельный модуль (`x`), на вопрос "зачем?" - отвечает следующий пункт в FAQ: Где лучше сохранить вспомогательные функции для построения сложных XPath-селекторов? (Стратегия сохранения методов))

Выше мы чуть пожертвовали читабельностью в имени класса в пользу лаконичности в использовании (`x` вместо `xpath` или `xpath_helpers`). Мы смогли позволить себе это сделать, исходя из того, что `x` – вероятно известное или интуитивно понятное сокращение в контексте поиска по XPath (в той же библиотеке JQuery есть команда `$x` , которая ищет элемент именно по XPath). Вспомни цитату из "стратегии хороших домохозяек о важности имен":

> Таблички будут однозначно описывать то, что в них лежит, будут понятными и читабельными для всех членов семьи, **написанные на "общем для всех языке", будут использовать "общепринятые термины"**

Также, в контексте работы с css-классами стоит обратить внимание на следующие вопросы:

- Стоит ли перенести в реализацию функции `has_css_class` открывающую и закрывающую квадратные скобки, или нет?
  - Почему?
    - Если да, то есть ли смысл переименовать функцию? Почему?
      - Если да, то какое тогда имя подойдет лучше всего?
- Нужны ли тут какие то дополнительные функции при работе с css-классами?

В принципе ответив на эти вопросы, и внеся, если нужно, соответствующие изменения в код, можно и остановиться. Особенно - если использование XPath-селекторов не будет сильно часто на проекте, и будет использовано только как "обходной путь" (workaround).

Если же по каким то причинам есть нужда использовать XPath-селекторы на постоянной основе, да еще и не очень опытными автоматизаторами, например ручными тестировщиками, то возможно стоит задуматься о полноценном DSL для XPath в проекте, построенном на полном наборе нужных функций-хелперов, что бы в итоге получить что то такое:

```
x.all().filter(by.id('todo-list'))\
    .child('li').filter(by.have_descendant_with_text('b'))\
    .descendant().filter(by.css_class('toggle'))\
    .x()
```

Создание такой DSL входит в следующее "бонусное" задание из этой книги;)

Решение: Список частых ошибок

Избыточность имен

Допустим у нас есть код как решение проблемы сложного xpath селектора для поиска по css-классу:

Пример 1

```
# selene-intro/selene_intro/common/selectors/xpath_selectors.py

# ...

def filter_by_css_class(value: str) -> str:
    return '[contains(' \
           'concat(" ", normalize-space(@class), " "), ' \
           f'" {value} ")]'
# ...
```

Пример 2

```
# selene-intro/selene_intro/common/selectors/xpath_filters.py

# ...

def filter_by_css_class(value: str) -> str:
    return '[contains(' \
           'concat(" ", normalize-space(@class), " "), ' \
           f'" {value} ")]'
# ...
```

Налицо избыточность в именах модуля или функции...

Вспомни правило хороших домохозяек о важности надписей;) Там есть фраза:

> Надписи также будут по возможности достаточно краткими. Ни одна хозяйка на полке с приправами не будет на каждой из баночек писать: "Приправа: перец", "Приправа: карри", "Приправа: тмин" – и так ведь понятно, что все, что лежит

на полочке с надписью "приправы" – приправы и есть, достаточно на каждой из баночек просто написать название приправы: "Перец", "Карри", "Тмин"...

Решения...

Пример 1

В первом примере незачем повторять в имени модуля слово `selectors`. Во-первых, файл с кодом уже лежит в пакете с таким именем. Во-вторых, ничего другого кроме "селекторов" в контексте использования xPath в нашем проекте быть не может, и так очевидно что речь идет о хелперах для составления селекторов.

```
# selene-intro/selene_intro/common/selectors/xpath.py

# ...

def filter_by_css_class(value: str) -> str:
    return '[contains(' \
           'concat(" ", normalize-space(@class), " "), ' \
           f'" {value} ")]'
# ...
```

Пример 2

Во втором примере слово `filter` повторяется как в имени модуля так и функции. Получается тафтология в стиле "масло маслянное", такая детализация излишня, она только удлиняет код усложняя его:

```
browser.element('//*' + xpath_filters.filter_by_css_class('toggle')).click()
```

Достаточно упомянуть только в одном месте, например так:

```
# selene-intro/selene_intro/common/selectors/xpath.py

#...

def filter_by_css_class(value: str) -> str:
    return '[contains(' \
           'concat(" ", normalize-space(@class), " "), ' \
           f'" {value} ")]'

# ...
```

или так:

```
# selene-intro/selene_intro/common/selectors/xpath_filter.py

#...

def by_css_class(value: str) -> str:
    # ...

# ...

by_no_css_class(value: str) -> str:
    #...

# ...
```

Последняя версия более оптимизирована, так как уже упомянув слово "filter" в имени модуля - не приидется повторять его каждый раз в именах функций при их определении. При использовании - контекст "filter" все равно будет задан именем модуля:

```
...xpath_filter.by_css_class('toggle')
#...
...xpath_filter.by_no_css_class('complete')
```

Неясные неполные имена вспомогательных функций

Допустим, есть код, как решение проблемы сложного xpath селектора для поиска по css-классу:

```
# selene-intro/selene_intro/utils/xpath.py

#...
def by_css_class(value: str) -> str:
    return 'contains(' \
           'concat(" ", normalize-space(@class), " "), ' \
           f'" {value} ")'

# ...
```

```
#...
browser.element(
    '//*[@id="todo-list"]/li[2]//*[' + xpath.by_css_class('toggle') + ']'
).click()
```

Вспоминаем правило хороших домохозяек о важности надписей;) В нем есть следующая фраза:

> Прочтя надпись, должно быть сразу понятно, что лежит внутри.

Теперь вопрос. Понятно ли по имени нашей "корзины" с хелпером – `xpath` – что внутри лежит хелпер-функция, которую можно использовать именно внутри квадратных скобок xpath-селектора? Возможно, этот хелпер можно использовать так:

```
#...
...'//*[@id="todo-list"]/li[2]//*' + xpath.by_css_class('toggle')
#...
```

или так:

```
..."//*[@id='todo-list']/li[2]//" + xpath.by_css_class('toggle')
```

или даже так:

```
browser.element(xpath.by_css_class('toggle')).click()
```

`o_O` ?

Да, с одной стороны мы, можно сказать, следовали одной из стратегий все той же домохозяйки:

> *Разные функции, но в небольшом количестве, в самом самом начале? – допустимо один модуль* Как и в реальной жизни, в программировании, в самом самом начале, "яиц" может быть не так много, чтобы раскладывать их по разным корзинам...

Но получается в этом случае, следование только этой стратегии - недостаточно, ведь с имени нельзя полностью понять контекст использования. Все это конечно относительно. "Очевидность" имен - в любом случае субъективна. В какой то из команд такого имени будет достаточно. Но давай в нашем случае попробуем хотя-бы слегка улучшить ситуацию ;)

Подумай над тем, как изменить имя, как уточнить его, используя соответствующую терминологию, что бы с имени функции (включая ее полное имя - вместе с именем модуля) - было более очевидно как его использовать. Таким образом улучшив код в соответствии с принципом самодокументируемого кода. Для этого, подумай над следующими вопросами:

- какую роль играют разные "части" xpath-синтаксиса?
  - квадратные скобки?
  - то что внутри квадратных скобок?
  - как можно назвать эти части?

Решение...

Получается, нам нужно как-то отобразить в полном имени функции `xpath.by_css_class` то, что она должна быть использован именно внутри квадратных скобок:

```
...'//*[@id="todo-list"]/li[2]//*[' + \
    xpath.by_css_class('toggle') + ']'
```

Скобки вместе с тем что у них внутри - по сути есть "уточнением селектора" или "фильтром элементов по пути отображенному в селекторе ранее". А фильтр этот - может состоять из одной или более "штук" которые в программировании принято называть "предикатами", то есть "функциями возвращающими `true` или `false` ". Получается наш `by_css_class` - и есть такой функцией, то есть предикатом (а не фильтром например, который бы требовал наличия квадратных скобок в контексте использования в xpath-селекторе). Понимая эту терминологию мы можем отобразить ее в имени нашей функции:

```
...'//*[@id="todo-list"]/li[2]//*[' + \
    xpath.predicate_css_class('toggle') + ']'
```

или так:

```
...'//*[@id="todo-list"]/li[2]//*[' + \
    xpath_predicate.css_class('toggle') + ']'
```

или так:

```
...'//*[@id="todo-list"]/li[2]//*[' + \
    xpath.predicate.css_class('toggle') + ']'
```

Последний способ - может быть самым универсальным, учитывая то, что набор наших "яиц в корзине" может быть достаточно разнообразным. Попробуй, ради тренировки, его реализовать, и если не получится - смотри подсказки на следующей странице...

Подсказки...

Есть два способа реализовать код вида `xpath.predicate.css_class` где `css_class` - точно функция (или метод).

Первый сопособ:

- внутри пакета `xpath` добавить модуль `predicate.py` с функцией `css_class(value`
- внутри `xpath/__init__.py` сделать пред-импорт модуля `predicate.py`

Второй способ:

- внутри модуля `xpath.py` добавить класс `class predicate:` в котором обьявить метод класса `css_class(cls, value)`

Попробуй реалиизовать оба ради тренировки ;)

Решение...

Первый способ:

```
# .../xpath/__init__.py

def filtered_by(predicate: str) -> str:
    return '[' + predicate + ']'


from selene_intro.tools.xpath import predicate, filter_by



# .../xpath/predicate.py

def css_class(value) -> str:
    return 'contains(' \
           'concat(" ", normalize-space(@class), " "), ' \
        f'" {value} ")'


def not_(predicate: str) -> str:
    return 'not(' + predicate + ')'



# .../xpath/filter_by.py

from selene_intro.tools.xpath import predicate, filtered_by


def css_class(cls, value: str) -> str:
    return filtered_by(predicate.css_class(value))


def no_css_class(cls, value: str) -> str:
    return filtered_by(predicate.not_(predicate.css_class(value)))
```

Второй способ:

```
# .../xpath.py

class predicate:
    @classmethod
    def css_class(cls, value) -> str:
      return 'contains(' \
             'concat(" ", normalize-space(@class), " "), ' \
             f'" {value} ")'

    @classmethod
    def not_(cls, predicate: str) -> str:
        return 'not(' + predicate + ')'


def filtered_by(predicate: str) -> str:
    return '[' + predicate + ']'


class filter_by:

    @classmethod
    def css_class(cls, value: str) -> str:
        return filtered_by(predicate.css_class(value))

    @classmethod
    def no_css_class(cls, value: str) -> str:
        return filtered_by(predicate.not_(predicate.css_class(value)))
```

... что позволит нам достичь цели теперь и без явного заворачивания в квадратные скобки и сделать код чуть "чище" и читабельней:

```
...'//*[@id="todo-list"]/li[2]//*' + xpath.filter_by.css_class('toggle')
```

В такой реализации как выше, мы пошли против общепринятого соглашения в Python - называть классы в `CamelCase` стиле (например, в нашем случае мы исполльзовали `filter_by` вместо `has_css_class`). Мы позволили себе это, потому что не используем классы по назначению. Они нужны нам не как "шаблоны или планы обьектов", а как "вложенные неймспейсы/модули". Но можно этого и не делать, возможно даже "лучше" так не делать;). Это уже выбор каждого;)

В любом случае, с такими темпами, можно пойти еще дальше... Если на каком то проекте мы вынуждены использовать часто XPath селекторы а тесты стараются писать не опытные автоматизаторы, то может стоит им помочь и еще больше расширить набор таких хелперов для составления XPath-селекторов:

```
browser.element(
    xpath.all + xpath.filter_by.id('todo-list') +\
    xpath.child('li') + xpath.filter_by.index(2) +\
    xpath.descendant() + xpath.filter_by.css_class('toggle')
).click()
```

Можешь, все таки, попробовать реализовать и все эти хелперы ради тренировки, хотя в этом задании это и не обязательно, и даже наоборот, желательно обойтись только "нужным минимумом". В следующем задании мы как раз и займемся построением похожей DSL, но еще более удобной, реализованной на основе подхода объектно-ориентированного программирования.

А я пока подниму еще одну тему... Кому то может показаться, что хотя синтаксис использования наших хелперов получился абсолютно читабельным - он при этом достаточно громоздкий, и возможно для кого-то - чересчур громоздкий. В какой то команде, могут отдавать предпочтение более "сухому" но лаконичному стилю... В таком случае, мы можем попробовать упростить наш стиль до чего-то такого:

```
browser.element(
    '//*[@id="todo-list"]/li[2]//*[' + x.has_css_class('toggle') + ']'
).click()
```

Мы сделали имя модуля более лаконичным, исходя из того что в "нашей команде" такое сокращение кажется довольно натуральным и очевидным, и никак не конфликтует ни с какими другими именами модулей. И мы убрали "вложенность", добавляя к имени функций, которые являются предикатами - префикс, указывающий на то, что они предикаты. В программировании как раз и есть общепринятой практикой добавлять к функциям возвращающим `boolean` значения (`true` или `false`) - префиксы типа `is` / `has` / `have` .

Таким образом мы и реализацию упростили, сделав ее структуру менее вложенной:

```
# .../x.py

def has_css_class(value: str) -> str:
    return 'contains(' \
        'concat(" ", normalize-space(@class), " "), ' \
        f'" {value} ")'


def not_(predicate: str) -> str:
    return 'not(' + predicate + ')'


def filtered_by(predicate: str) -> str:
    return '[' + predicate + ']'
```

... и упростили использование. Ведь, поскольку вложенность только одноуровневая - функция автодополнения кода в редакторе будет подсказывать сразу весь список хелперов после введения `x.` , что позволит быстрее находить пользователям то что им нужно, ведь они получают доступ сразу ко всему...

При этом функции типа:

```
def filtered_by_css_class(value: str) -> str:
    return filtered_by(has_css_class(value));

def filtered_by_no_css_class(value: str) -> str:
    return filtered_by(not_(has_css_class(value)));
```

... можно уже и не добавлять, учитывая например то, что команда наша "сильная" и ей незачем плодить кучу хелперов, достаточно и чистого использования кода вида `x.filtered_by(x.not_(x.has_css_class(value)))` там где это нужно;)

Задание: XPath DSL

Бывает, по каким то причинам... есть нужда использовать XPath-селекторы на постоянной основе, да еще и не очень опытными автоматизаторами, например ручными тестировщиками. Тогда возможно стоит задуматься о полноценном DSL для XPath в проекте, построенном на полном наборе нужных функций-хелперов, что бы в итоге получалось что то такое:

```
x.all().filter(by.id('todo-list'))\
    .child('li').filter(by.have_descendant_with_text('b'))\
    .descendant().filter(by.css_class('toggle'))\
    .x()
```

В этом задании, предлагается реализовать такой DSL;) Это бонусное и необязательное задание. Оно также довольно сложное, ибо включает в себя использование никоторых продвинутых практик объектно-ориентированного программирования. Если спешишь побыстрее освоить именно автоматизацию веб приложений, можешь пропустить это задание. Но если даже создание обычных хелпер-функций из предыдущего задания о "рефакторинге XPath селекторов" вызывало у тебя трудности, то это задание рекомендуется к освоению в ближайшие строки. К ему в придачу стоит добавить практику на ресурсах рекомендуемых в конце раздела Необходимые знания перед стартом (exercism.io: Python Track, CheckIO). Или может стоит даже вернуться к самой базе языка (code-basics: python, Hexlet.io: Python: Основы) ;)

Если же проблем с предыдущими заданиями не возникало, и тебя манит эта задача, но как приступить - пока не понятно, погугли на тему "python builder pattern", и подумай о том как используя похожие идеи, построить нужную тебе реализацию. Если все останентся в тумане, ищи более детальных инструкций в следующем разделе с частыми вопросами и ответами ;)

Также, учти, что пример выше, вероятно, показывает не весь функционал DSL, который отображал бы все возможности XPath. Попробуй подумать каких "фишек" в нем нет, и расширь DSL соответственно;)

Удачи!

Общие частые вопросы и ответы (FAQ)

Тестовая логика

Тестовая логика тест кейса состоит обычно из:

- предусловий
- тест-шагов
- проверок-ассертов (сравнений актуальных результатов с ожидаемыми)
- тестовых данных

Все это и определяет тестовый случай ("тест-кейс") в жизни приложения, который позволяет оценить качество последнего при соответствующих условиях. И конечно же вскрыть баги в приложении, если они есть.

С точки зрения автоматизации, очень важно что бы все эти аспекты - легко читались в коде автотеста. Ибо тест кейс - это и есть тест логика. И сложность понимания тест логики из кода теста приведет к усложнению поддержки последнего, и соответственно понизит эффективность автоматизации.

По сути - "читабельность и очевидность тест-логики" - и можно определить как базовый принцип в автоматизации тестирования. Все другие принципы и шаблоны - DRY, KISS, YAGNI, PageObject - как раз и призваны в помощь первому;)

Рекомендации и общепринятые договоренности в подборе имен

Именование любых сущностей в программировании – очень важно. От того как ты будешь называть свои модули и функции, переменные, классы, поля и методы - зависит понимание твоего кода другими программистами. Очевидно, что "сложное для восприятия имя" будет замедлять работу с кодом, а "имя приводящие к неправильному представлению о именованной сущности" рано или поздно приведет к появлению дефектов. Также, обычно важна "общепринятая в кругах программистов" стилистика, которой стоит следовать чтобы "поставлять как можно более привычный а следовательно и проще понимаемый большинству код". Рекомендуется нарушать любую такую общепринятую конвенцию только хорошо подумав, и поняв что плюсов от "нарушения" - больше чем минусов (например минусом может быть – неочевидность и дикость для других членов команды).

Начиная работать в новом для себя проекте, поинтересуйся о договоренностях в нейминге, принятых в конкретно этом проекте. Возможно, есть документ, описывающий это. Возможно, тебе придется сделать выводы самостоятельно, проанализировав уже существующий и использующийся код. В любом случае - стоит придерживаться "локальных" правил - так как для всей команды они уже очевидны.

Полезные ссылки

- PEP 8 -- Style Guide for Python Code: Naming Conventions
- 7 Popular Unit Test Naming Conventions
  - примеры для C#, поэтому стоит учитывать разницу в использовании регистра в именах

Имя модуля?

- Отражает контекст использования функций, переменных, либо классов в этом модуле
- не содержит нумераций и прочей структурной информации
  - нумерации и оргинизационное структурирование лучше реализовывать "пакетами"

- начинается с маленькой буквы
  - shortlowercase
    - longer_underscored_is_ok_if_improves_readability

Имя класса?

- Отражает сущность которую представляют объекты этого класса
  - Обычно характер сущности определяется - "поведениями обьекта" - представленными в виде его публичных методов
- не содержит нумераций и прочей структурной информации
  - нумерации и оргинизационное структурирование лучше реализовывать "пакетами" (python packages), а не уточнять в имени модуля либо класса
- начинается с большой буквы
  - UpperCamelCase (PascalCase)

Имя пакета python?

- отражает структурную информацию о коде внутри пакета, который по сути группирует python-модули "по контексту"
  - shortlowercase
    - long_underscored_is_discouraged

Имя функции либо метода?

- отражает/описывает цель которую выполняет функция или метод
- начинается с маленькой буквы
  - snake_case

Имя переменной?

- отражает/описывает то, "что сохраняет" переменная
- начинается с маленькой буквы
  - snake_case

Имя константы?

- `SCREAMING_SNAKE_CASE`

    - большие буквы
    - слова разделены подчеркиванием

Имя тест-модуля или тест-класса?

- Имя тест-модуля либо тест-класса играет роль имени тест-суита (набора тест-кейсов) и должно отражать в общем - что тестируют методы данного тест-класса. При этом учитывая соглашения о именах для модулей и классов в Python.
    - формат для имени тест-модуля
        - `test_<[имя_приложения][_часть_приложения][_фича_приложения]>.py`
        - или
        - `<[имя_приложения][_часть_приложения][_фича_приложения]>_test.py`
    - формат для имени тест-класса
        - `Test<[ИмяПриложения][ЧастьПриложения][ФичаПриложения]>`
            - пример хорошего имени: `class TestSomeSocialNetworkPostManagement: ...`
            - пример ошибки: Задание: Selenide и CSS-селекторы: Несоответствие имен тест-суита или тест-кейсов целям тестирования
- При этом стоит учитывать контекст, и по возможности подбирать более лаконичные имена тест-классов. Например, можно указать только имя приложения, если оно простое и у нас для его тестирования лишь один тест-модуль или класс:

только тест модуль:

```
# my-company-tests/tests/test_some_social_network.py
# ...
```

тест-модуль + тест-класс:

```
# my-company-tests/tests/test_some_social_network.py
# ...
class TestSomeSocialNetwork:
    # ...
```

- можно указать только **часть приложения**, если на каждую часть - приходится по

одному тест-набору, и мы уже задали *контекст положив тест-набор в соответствующий пакет*:

только тест модуль:

```
# my-company-tests/tests/some_social_network/test_post_management.py
# ...
```

только базовый тест модуль с тест-классами:

```
# my-company-tests/tests/test_some_social_network.py
# ...
class TestPostManagement:
    # ...

class TestProfileManagement:
    # ...
```

тест-модуль + тест-класс:

```
# my-company-tests/tests/some_social_network/test_post_management.py
# ...
class TestPostManagement:
    # ...
```

разные тест-классы (для еще более гранулированных частей приложения) в одном тест модуле:

```
# my-company-tests/tests/some_social_network/test_post_management.py
# ...

class TestPostTextEditing:
    # ...


class TestPostCommentsEditing:
    # ...
```

- можно указать только **фичу** - из *аналогичных предыдущему примеру соображений*:

только тест модуль:

```
# my-company-tests/tests/some_social_network/post_management/test_editing.py
# ...
```

тест модуль (для фичи) с классами (для "под-фич"):

```
# my-company-tests/tests/some_social_network/post_management/test_editing.py
# ...

class Test**Text**:
    # ...


class Test**Comments**:
    # ...
```

- стоит опустить пакет с именем приложения которое тестируем, если весь проект с тестами касается только этого приложения:

только тест модуль:

```
# **some_social_network**-tests/tests/post_management/test_editing.py
# ...
```

тест модуль с классами:

```
# **some_social_network**-tests/tests/post_management/test_editing.py
# ...

class TestText:
    # ...


class TestComments:
    # ...
```

- Обычно имя тест-класса начинают с Test, чтобы указывать, что это именно тест-класс а не обычный
  - Часто, по-умолчанию тест-раннеры настроены по умолчанию таким образом, что будут искать и запускать только те тесты, которые начинаются на Test (хотя это можно перенастроить).
  - Иногда этой рекомендации не следуют, например когда симулируют BDD, где файлы со сценариями обычно называют "фичами" (features) или "спеками" (specs от specifications) – и тогда либо никакой приставки не пишут, либо используют `Spec` вместо `Test` .
  - Рекомендуется нарушать любую общепринятую конвенцию только хорошо подумав, и поняв что плюсов от "нарушения" - больше чем минусов (например минусом может быть – неочевидность и дикость для других членов команды...)
- В имя тест-класса не нужно выносить структурную информацию (номер версии, вариант решения и т. п.) или свойства, которые не характеризуют тест-класс в целом с точки зрения тестового покрытия
  - лучше для этого использовать отдельные пакеты
  - пример ошибки: Задание: Selenide и CSS-селекторы: Структурная информация в именах тест-суита и тестов, которая не несет пользы с точки зрения тестирования

Пакеты в тестовом проекте?

- С помощью пакетов можно фиксировать любую структурную информацию, например
  - если много тестов в разных стилях... Например, часть тестов - атомарные (с фокусом на одной фиче), а часть - стиля "end to end", покрывающие несколько фич в контексте определенного "пути пользователя" ("user workflow", "user journey"), то можно сгруппировать тесты по следующим пакетам:
    - пример 1:
      - some-social-network-test/tests/e2e
      - some-social-network-test/tests/atomic
    - пример 2:
      - some-social-network-test/tests/workflows

        - some-social-network-test/tests/features
    - пример 3:
        - some-social-network-test/tests/journeys
        - some-social-network-test/tests/features
- при решении некоторых заданий, можно создавать разные пакеты для разных версий решений
    - примеры:
        - selene-intro/tests/**lesson**/test_todomvc.py
        - selene-intro/tests/**lesson**/test_google.py
        - selene-intro/tests/**tasks/selene_and_css**/test_todomvc.py
        - selene-intro/tests/**tasks/selene_and_xpath**/test_todomvc.py
        - selene-intro/tests/**tasks/xpath_refactoring**/test_todomvc.py
        - selene-intro/tests/**tasks/xpath_dsl**/test_todomvc.py
- пример ошибки: Задание: Selenide и CSS-селекторы: Структурная информация в именах тест-суита и тестов, которая не несет пользы с точки зрения тестирования

Тест-функции и тест-методы?

В первую очередь, имя тест-функции либо тест-метода играет роль имени тест-кейса и отображает то, что покрывают его шаги. Желательно, должно отвечать стандартным договоренностям Python по составлению имен функций или методов и может строиться по одной из схем:

- `test_<фича | под_фича>_[_]<ожидаемый_ввод | тестовое_состояние>_[_]<ожидаемое_поведение>`

Например:

```
test_register_new_user_existing_email_given_should_show_error_message()
```

либо чуть более выделяя структуру:

```
test_register_new_user__existing_email_given__should_show_error_message()
```

либо, хорошо подумав, все взвесив, и приняв решение пойти против стандартных соглашений в Python:

```
test_registerNewUser_ExistingEmailGiven_ShouldShowErrorMessage()
```

... аргументируя свое решение компактностью и тем, что зажимать шифт при слепом наборе проще чем тянуться правым мизинцем к подчеркиванию с зажатым левым мизинцем шифтом ;p

- `test_<фича | под_фича>_[_]<ожидаемое_поведение>_[_]<ожидаемый_ввод | тестовое_состояние>`

Примеры:

```
test_register_new_user_shows_error_message_when_email_exists()
```

```
test_register_new_user__shows_error_message__when_email_exists()
```

```
test_registerNewUser_ShowsErrorMessage_WhenEmailExists()
```

- `given_<предусловия>_when_<ожидаемыеДействия>_then_<ожидаемое_поведение>`

Для такого формата нужны будут дополнительные настройки тест-раннера, например для pytest:

```
# content of pytest.ini in the root of your tests-project
[pytest]
python_functions = given_*
```

Примеры:

```
given_email_exists_when_register_new_user_then_error_message_is_shown()
```

```
given_email_exists__when_register_new_user__then_error_message_is_shown()
```

```
GIVEN_email_exists_WHEN_register_new_user_THEN_error_message_is_shown()
```

```
givenEmailExists_whenRegisterNewUser_thenErrorMessageIsShown()
```

Как то длинновато получается? - Главное, что-бы было читабельно. Лаконичность это прекрасно но только если не ухудшает читабельности;).

Можно избавлятся повторений выделяя дополнительные тестовые наборы, например с помощью тест-классов, например

код с длинными именами тестов с повторяющимися частями:

```
def test_register_new_user_shows_error_message_when_email_exists():
```

```
    # ...


def test_register_new_user_shows_error_message_when_email_invalid():
    # ...

# ...
```

можно улучшить до:

```
class TestRegisterNewUserShowsErrorMessage:

    def test_when_email_exists():
        # ...

    def test_when_email_invalid():
        # ...

    # ...
```

или даже так:

```
# content of pytest.ini in the root of your tests-project
[pytest]
python_functions = test_* when_*


# content of your test module

class TestRegisterNewUserShowsErrorMessage:

    def when_email_exists():
        # ...

    def when_email_invalid():
        # ...

    # ...
```

Ну и конечно же стоит избавлятся от тех частей имени, в которых нет необходимости учитывая контекст.

Например, на уровне приемочного тестирования (на котором чаще всего и проводится автоматизация пользовательского интерфейса веб-приложение, и где не стоит копать глубже в контексте "пирамиды тестирования") часто можно оставить только:

- `test_<ожидаемое_поведение>`

Пример:

```
def test_registers_new_user():
    # ...
```

- `test<*ожидаемое*поведение><*предусловие* | *тестовое*состояние>`

Примеры:

```
def test_fails_to_register_new_user_when_email_already_in_use():
    # ...
```

или если есть договоренность на проекте все негативные тесты начинать с `fails_`

```
# content of pytest.ini in the root of your tests-project
[pytest]
python_functions = test_* fails_*


# content of your test module

def fails_to_register_new_user_when_email_already_in_use():
    # ...
```

или если таких "негативных тестов" больше чем один:

```
# content of pytest.ini in the root of your tests-project
[pytest]
python_functions = test_* when_*
```

```
# .../fails_to_register_new_user_test.py

def when_email_already_in_use():
    # ...


def when_email_invalid():
    # ...
```

При этом для сценариев End-to-End-стиля допустимы более абстрактные имена, отображающие определенный "бизнес-флоу" в приложении. Обычно, полезным в именах будет использование следующих ключевых слов:

- flow
- journey
- lifeCycle
- management
- common
- и т.д.

Пример для тестов интернет магазина:

```
test_provides_common_products_management()

test_product_lifecycle()

test_buying_product_jorney()
```

Для Unit-тестов в именовании будут свои нюансы - это как раз та область где "копают глубоко";) Неплохой список разных подходов к составлению имен для unit-тестов описан в статье 7 Popular Unit Test Naming Conventions, правда с примерами на C#, поэтому стоит учитывать разницу в использовании регистра в именах

Также, как было показано в некоторых примерах выше, удобно строить имя теста начиная с имени тест-модуля либо тест-класса, стараясь при этом что-бы полное имя строилось в валидное предложение на английском. Пример схемы:

- **инструмент**.py или class **Инструмент**
  - def *выполняет_действие*()

Примеры:

с тест-модулем:

```
# test_some_social_network.py

def test_provides_message_exchange():
    # ...
```

с тест-классом:

```
# ...

class TestSomeSocialNetwork:

    def test_provides_message_exchange():
        # ...
```

... что складывается в предложение "Some social network provides message exchange"

Другие похожие варианты именования тест-функции либо метода:

```
# глагол(инфинитив) + детали
def provide_message_exchange():
    #...

# детали + главное слово-существительное-обозначение процесса
def message_exchange():
    #...

# детали + общее выражение-обозначение процесса
def messages_life_cycle():
    #...

# ... или
def messages_common_flow():
    #...
```

Общие рекомендации к улучшению читабельности имен

Шаблон построения имени функции или метода

- `имя_метода = что_сделать[_над_чем_если_надо_уточнить] = verb_subject | predicate_subject`

Поскольку метод - это "команда", то есть действие, то логично называть метод используя глаголы в указательной форме, например:

```
add_product("The Art of Automation")
```

Могут быть и исключения. Например, если метод является не "командой", а *"запросом" (query)*, который возвращает объект/сущность - тогда можно использовать существительное, например:

```
products()
```

... или даже прилагательное, если из контекста и так будет понятно о чем речь, например...

`emails.unread()` возвращает список непрочитанных имейлов.

При этом можно использовать и глагол:

```
emails.fiter_unread()
```

Это уже зависит от контекста, что лучше…

- Если метод сразу выполняет поиск/фильтрование и возвращает результат - тогда можно упомянуть глагол filter/get в имени.

```
emails.filter_unread()
```

- Если же метод просто возвращает какой то объект, который представляет собой отфильтрованные элементы но реально их поиск еще не был осуществлен, а будет выполнен только после того как мы захотим что-то сделать с этими, например, имейлами - тогда можно просто:

```
emails.unread()
```

- или, может, мы не хотим чтобы пользователь метода задумывался о том - есть

там поиск сразу или нет, для нас главное читабельность и лаконичность? - тогда также можно оставить только:

`emails.unread()` вместо `emails.filterUnread`

`products()` вместо `get_products()`

Как видно, из некоторых примеров, мы иногда опускаем из имени метода даже существительное, потому что оно уже задано контекстом, то есть `emails.unread()` лучше чем `emails.unread_emails()` потому что и так понятно о чем речь, незачем дважды повторять. Но если где то контекст будет непонятен - тогда лучше существительное добавлять.

Шаблон построения имени переменной

- `имя_переменной = [какой_если_надо_уточнить_]_объект = adjective_object`

Поскольку переменные представляют значения/сущности/объекты - их и называть стоит с помощью существительных, ведь именно это мы и делаем в реальной жизни. Это очевидно и натурально. Использовать в имени переменных как основное слово - глагол - обычно будет ошибкой. Потому что глагол отвечает действию - а переменная ничего не может делать в момент использования - она просто хранилище данных/значений.

При этом названия должны быть **емкими и лаконичными**, оставаясь **читабельными и наглядными**. Например, использование слов `activate` или `reopen` в имени предпочтительней чем `mark_as_active` .

Использование общепринятой терминологии

Лучше использовать **более употребляемый и знакомый всем синоним** для определенных контекстов.

Например, `assert` или `should` лучше использовать, чем `check` для методов описывающих проверки в тестах. И вот почему...

Важно, что-бы читая код тестов, сразу было понятно - где шаги, а где проверки. Особенно, если кода много… Следовательно, лучше использовать какое-то ключевое слово, которое будет "сильным сигналом" - "вот - я проверка!"

Можно было-бы использовать слова типа "check"/"verify". Но, поскольку уже так сложилось, что в большинстве языков программирования для "проверок" используется слово **assert** - то довольно удобно использовать именно его.

Просто потому, что все, кто занимается программированием - уже знают это. В Python даже на уровне синтаксиса есть такое ключевое слово `assert`. Некоторые тест-фреймворки уже используют методы типа `assert_true`, `assert_equals`, `assert_that`, и т. п. И поскольку все это знают и знакомы с этим, то глазам сразу проще будет "зацепиться" именно за слово `assert`.

Иногда, реже, используется еще слово **should** – как и в Selene, кстати. При определенном стиле - с ним код лучше читается. Например:

```
element.should(appear)
```

лучше читается чем

```
assert_that(element, appear)
```

но вот если начать заворачивать такой код в методы типа:

```
component_should_appear();
#...
assert_component_appears();
```

то уже далеко не факт, что версия с "should" будет читаться лучше в контексте "визуального определения - где шаг, а где проверка". Потому что при чтении кода, глазу проще зацепиться за часть в начале слова а не в середине…

Есть здесь, конечно, и доля субъективизма… Да и всегда стоит расматривать ситуацию с разных сторон. Другой аспект, который всегда стоит учитывать - это **консистентность**. В нашем случае это будет значить - если мы используем в Selene - "should" - то почему тогда и для имен методов-проверок не использовать? – Так имена будут более "консистентными" и "последовательными". Следование одному стилю приводит обычно к простоте, и более легкому порогу входа.

Также на выбор влияет общая стилистика тестов. При использоварнии BDD-стиля специально подбирают термины таким образом что-бы язык теста описывал функционал с точки зрения пользователя. Это позволяет при разработке и тестировании факусироваться на том, что действительно важно пользователю, для него ведь все это делается. Он - конечный клиент приложения. В таком случае, как раз очень хорошо подходит `should`. Собственно от BDD он и пришел :)

Еще, на пользу глазам идет использование стиля объектно-ориентированного програмирование, в котором код вида...

```
component_should_appear();
```

... обычно, превращается в:

```
component.should_appear();
```

... где "should" уже в начале имени метода а не в середине имени функции ;)

В любом случае, выбор между `assert` или `should` за тобой. У каждого из вариантов есть свои плюсы и минусы.

Вот в "check"/"verify" точно плюсов меньше. И даже с ними - "все может зависить от проекта";)

В некоторых случаях используют еще и `expect` терминологию:

```
expect(element).to(appear)
```

;)

Не злоупотреблять сокращениями

Заботясь о читабельности и очевидности для большинства **не стоит использовать сокращения**, кроме общеупотребимых, например nth = н-ый.

Терминология в контексте

Стоит **использовать терминологию** контекста по воможности. Если мы реализуем методы которые представляют действия пользователя в приложении, то есть находимся в контексте "UI" приложения, то и имя нужно подбирать соответственно дабы не разводить зоопарк терминов и использовать тот язык, на котором в этом контексте уже говорят.

Пример. Если речь идет о локаторе для счетчика активных задач, который на UI подписан как "left items", то и назвать переменную лучше `left_items_counter` или даже просто `left_items` чем `active_items_counter` .

Говорим на языке местных

... Это же касается общего стиля. Если речь идет о приемочном тестировании поведений (behaviours) пользователя, то и имена методов которые отображают соответсвующие поведения должны быть подобраны соответственно.

Пример. Если речь идет о действии (поведении) пользователя по подтверждению формы, то и имя соответствующего метода лучше составить как `submit()` вместо `click_submit_button()` . Последний вариант - сильно технический для даного контекста и не отображает реальную "бизнес" ценность с точки зрения пользователя.

Недвусмысленность

Мы ведь ни кого не хотим запутать, наоборот, наша главная цель - писать код, который легек не только в реализации но и в поддержке. Следовательно, имена функций или методов должно всегда **однозначно описывать то что они делают**. Это же касается и переменных и классов. По возможности, тот кто читает код, всегда с имени сущности должен понять, что происходит (возможно полностью все прояснить подсмотрев на типы и имена параметров).

Также, "однозначность" означает "всегда однозначность"... если метод называется `включить_лампочку()` - он всегда должен включать лампочку… Если же его реализация может и "выключать лампочку", значит либо метод нужно переделать что бы он "только включал лампочку" либо переименовать на `переключить_лампочку()` . В противном случае такие методы будут вводить в заблуждение их пользователей и приводить к конфузам и багам в конце концов.

Лаконичность

Если есть возможность написать более краткое, ясное имя, не жертвуя при этом "полнотой", не ухудшая понятность и читабельность - то грех этого не сделать.

Пример: `submit()` против `click_submit_button()` .

При этом если учитывать контекст то это, обычно, тоже играет на пользу лаконичности.

Это же правило распространяется не только на имена сущностей, но и в принципе на любой код;)

Не повторять то, что уже задано контекстом

Мало кто любит когда "много воды" и "мало сути". Стоит всегда **учитывать контекст** и "не повторять в имени то, что уже известно из контекста в коде". Контекст для имени переменных может быть уже задан модулями или функциями, классами или методами - в которых они "живут". Контекст методов - именем его класса. Контекст класса или функции - их модулем. Контекст модуля - его пакетом.

Пример с именем метода:

```
# Лучше
post = Post();
post.delete();


# Чем
post = Post();
post.delete_post();
```

Это правило можно также запомнить как анти-паттерн "Масло маслянное" и есть обычным признаком тавтологии;)

Простота

Мы уже не раз замолвили слово о краткости, лаконичности - ради читабельности. Причина этому очевидна - все длинное, "составное" - суть сложное, и следовательно тяжеленее в понимании и поддержке. Следовательно всегда нужно стремиться упрощать вещи. Обычный прием - разбивать "большое и сложное" на "маленькое и более простое".

Если имя сущности получилось "слишком длинным" - это уже есть сигналом на подумать о том, что-бы упростить его. Самый простой прием - "выделить контекст" (положить сущность в свой класс/метод или модуль/функцию). Нужно помнить, что код лучше подавать "маленькими порциями" для "лучшей читабельности и понятности".

Материалы рекомендуемые к самостоятельному освоению

Статьи

О локаторах.

Как строить хорошие локаторы? Алексей Баранцев

О загрузке страниц (Алексея Баранцева).

- ...что означает "окончание загрузки страницы"?
- ...как Selenium ожидает завершения загрузки страницы?
- ...что делать в Selenium, если страница загружается слишком долго?

О тестовой модели.

PageObject (by Martin Fowler)

О покрытии

- The Practical Test Pyramid (by Ham Vocke)
- THAT'S NOT THE MAP I HAD IN MIND: MEANING, IMPRECISION AND TAXONOMY OF VISUAL TEST MODELS by AARON HODDER
    - Перевод на русский:
        - Это не та карта, которую я имел в виду: значение, неточности и таксономия визуальных моделей тестирования, часть 1
        - Это не та карта, которую я имел в виду: значение, неточности и таксономия визуальных моделей тестирования, часть 2

Review of Visual vs. Functional Testing with pDiff and Applitools (by Greg Sypolt)

- Mirror at linkedin

О инфраструктуре.

- Dockerization of real mobile device farm and scalable QA automation ecosystem (by Alex Khursevich)

О мобильной автоматизации.

- Mobile automation state in 2019 (by Dmitry Lemeshko)

О феншуе в коде :)

- Tips for Writing Self-Documenting Code (by Mike Cronin)

Книги

Clean Code (by Robert C. Martin)

Selenium WebDriver. From Foundations To Framework

Selenium Design Patterns and Best Practices

Unit Testing: Principles, Practices, and Patterns by Vladimir Khorikov

Блоги

Enterprise Craftsmanship. Software development principles and best practices