**УСТАНОВКА ПИТОНА, УСТАНОВКА ДРАЙВЕРА, ПРОПИСАТЬ В PATH**

Создание директории и окружения

1. Создание директории make directory и далее название

mkdir **environments**

1. перейти в директорию

cd **environments**

1. создание виртуального окружения (selenium\_env - название окружения)

python -m venv **selenium\_env**

1. активация окружения

**selenium\_env**\Scripts\activate.bat

1. в окружение можно утсановить библиотеки: селениум и прочее

установка библиоткеки селениум (можно выбрать версию)

pip install selenium==4.\*

УСТАНВОКА ВЕБДРАЙВЕРА

<https://sites.google.com/chromium.org/driver/downloads>

<https://sites.google.com/chromium.org/driver/>

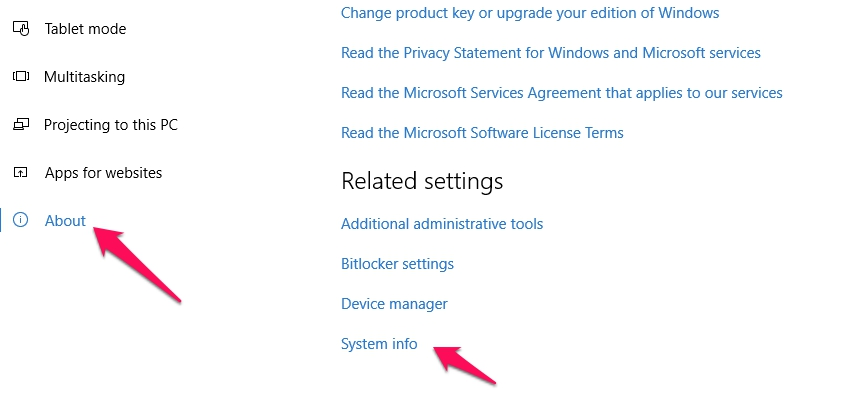
 оздайте на диске C: папку chromedriver и положите разархивированный ранее файл chromedriver.exe в папку C:\chromedriver.

 Добавьте в системную переменную PATH папку C:\chromedriver. Как это сделать в разных версиях Windows, описано здесь: <https://www.computerhope.com/issues/ch000549.htm>.

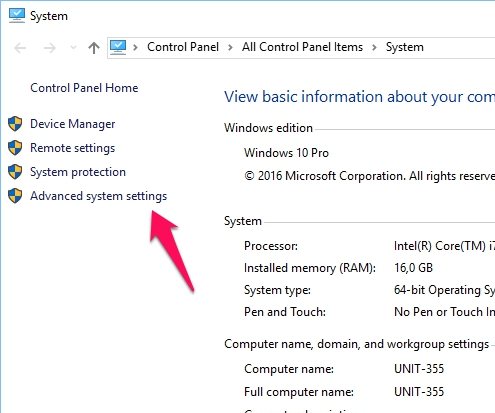
**Пример: как добавить путь в системную переменную PATH на Windows10**

1. Откройте настройки системы.

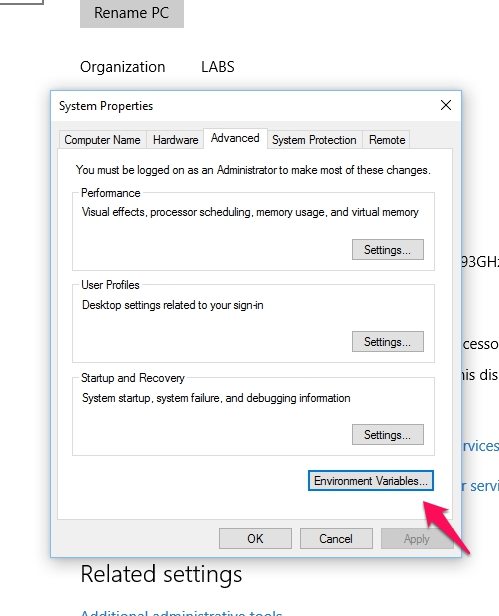
2. В настройках откройте вкладку About, затем System info:



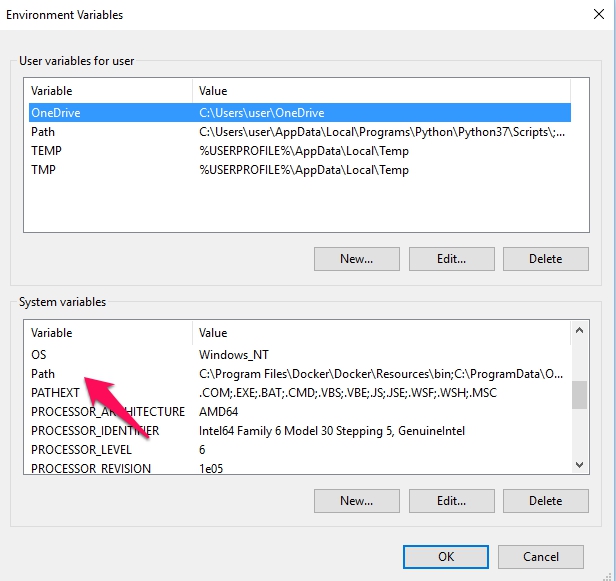
3. Выберите Advanced system settings:



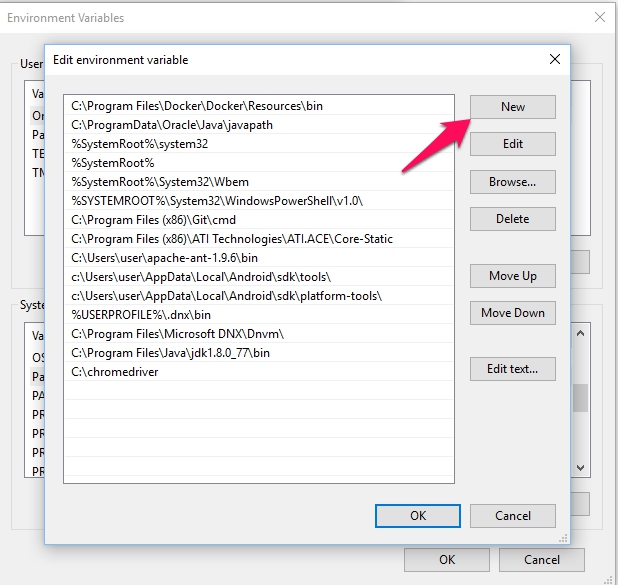
4. Выберите Environment Variables:



5. Кликните два раза на строчке Path в System variables:



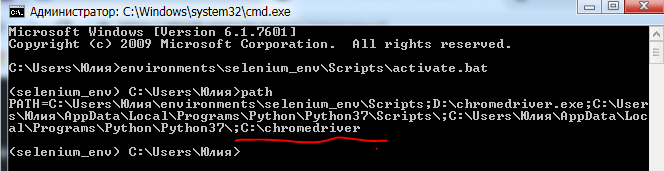
6. Нажмите кнопку New. Введите в новую строку путь к ChromeDriver — C:\chromedriver. Нажмите Enter. У вас должна появится строка с указанным путем:



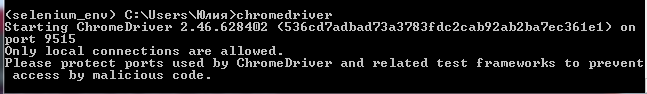
7. Если у вас была открыта командная строка Windows, не забудьте ее закрыть. Затем откройте новую командную строку, чтобы изменения переменной окружения стали доступны. Активируйте снова виртуальное окружение selenium\_env, которое мы создали в предыдущих шагах.

Давайте убедимся в том, что вебдрайвер установлен правильно.

Для начала проверим содержимое переменной path, для этого наберем в командной строке **Path:**



Ура, там есть папка с chromedriver! Попробуем вызвать его напрямую из командной строки:



Магия переменной path: хотя программа chromedriver находится где-то в другом каталоге, мы можем напрямую открывать её, используя имя chromedriver. Чтобы завершить процесс в консоли, нажмите Ctrl+C.

Знак на этом этапе, что пошло что-то не так:

https://ucarecdn.com/94268733-9715-405b-9da0-9397ae0bf9bf/

В таком случае попробуйте перезапустить консоль, перезапустить компьютер, перепроверить и добавить заново по инструкциям папку с chromedriver в переменную path.

Когда все получится, переходите к шагу <https://stepik.org/lesson/25969/step/11>, в котором мы запустим браузер с помощью Selenium WebDriver и выполним простые команды.

## Запуск браузера и первый скрипт

### ****Для всех ОС:****

В нашем виртуальном окружении запустим интерпретатор python:

~/environments$ python

Теперь напишем скрипт, который откроет следующий шаг в данном уроке на Stepik и отправит правильное решение в задаче. Рекомендуем выполнять команды последовательно по одной, чтобы лучше понять, что происходит в каждой строчке. Не пугайтесь, если не понимаете, что делают те или иные команды, — подробнее мы их разберем в следующих уроках.

import time

# webdriver это и есть набор команд для управления браузером

from selenium import webdriver

# импортируем класс By, который позволяет выбрать способ поиска элемента

from selenium.webdriver.common.by import By

# инициализируем драйвер браузера. После этой команды вы должны увидеть новое открытое окно браузера

driver = webdriver.Chrome()

# команда time.sleep устанавливает паузу в 5 секунд, чтобы мы успели увидеть, что происходит в браузере

time.sleep(5)

# Метод get сообщает браузеру, что нужно открыть сайт по указанной ссылке

driver.get("https://stepik.org/lesson/25969/step/12")

time.sleep(5)

# Метод find\_element позволяет найти нужный элемент на сайте, указав путь к нему. Способы поиска элементов мы обсудим позже

# Метод принимает в качестве аргументов способ поиска и значение, по которому мы будем искать

# Ищем поле для ввода текста

textarea = driver.find\_element(By.CSS\_SELECTOR, ".textarea")

# Напишем текст ответа в найденное поле

textarea.send\_keys("get()")

time.sleep(5)

# Найдем кнопку, которая отправляет введенное решение

submit\_button = driver.find\_element(By.CSS\_SELECTOR, ".submit-submission")

# Скажем драйверу, что нужно нажать на кнопку. После этой команды мы должны увидеть сообщение о правильном ответе

submit\_button.click()

time.sleep(5)

# После выполнения всех действий мы должны не забыть закрыть окно браузера

driver.quit()

Вы можете запустить этот же скрипт из файла. Скачайте файл [first\_script.py](https://stepik.org/media/attachments/lesson/25969/first_script.py) к себе в папку и запустите его. Для этого создадим папку selenium\_course и скопируем в нее скачанный файл. Затем запустим файл в нашем виртуальном окружении.

**Пример для Windows**

(Вместо user везде подставляйте имя пользователя в системе)

При выполнении следующих команд не забудьте изменить путь к вашей пользовательской папке вместо C:\Users\user и к папке с загруженным файлом.

Выполните следующие команды в командной строке Windows:

c:\Users\user> mkdir selenium\_course

C:\Users\user> copy C:\Users\user\Downloads\first\_script.py c:\Users\user\selenium\_course

C:\Users\user> environments\selenium\_env\Scripts\activate.bat

(selenium\_env) C:\Users\user>python c:\Users\user\selenium\_course\first\_script.py

**Пример для Ubuntu и macOS**

Выполните следующие команды в консоли:

mkdir ~/selenium\_course

mv ~/Downloads/first\_script.py ~/selenium\_course

python ~/selenium\_course/first\_script.py

## Поиск элементов с помощью составных CSS-селекторов

Теперь предположим, что не можем найти элемент на странице, используя простой селектор, так как такой селектор находит сразу несколько элементов. Ниже мы привели часть кода простой HTML-страницы, описывающей блог. Саму страницу вы можете посмотреть по [ссылке](http://suninjuly.github.io/blog_example.html).

Вопрос: как нам найти селектор для подписи у второй картинки? Вот здесь нам поможет иерархическая структура страницы и возможность комбинировать CSS-селекторы. CSS-селекторы позволяют использовать одновременно любые селекторы, рассмотренные ранее, а также имеют некоторые дополнительные возможности для уточнения поиска.

<div id="posts" class="post-list">

  <div id="post1" class="item">

    <div class="title">Как я провел лето</div>

    <img src="./images/summer.png">

  </div>

  <div id="post2" class="item">

    <div class="title second">Ходили купаться</div>

    <img src="./images/bad\_dog.jpg">

  </div>

  <div id="post3" class="item">

    <div class="title">С друзьями</div>

    <img src="./images/friends.jpg">

  </div>

</div>

**Использование потомков**

Попробуем найти элемент с текстом "Ходили купаться". Для решения этой задачи мы можем взять элемент, стоящий выше в иерархии нужного нам элемента, и написать следующий селектор:

#post2 .title

Здесь символ **#** означает, что надо искать элемент с id post2, пробел - что также нужно найти элемент-потомок, а **.**, что элемент-потомок должен иметь класс со значением title.

Элемент .title называется **потомком** (англ. **descendant**) элемента #post2. Потомок может находиться на любом уровне вложенности, все элементы с селектором .title также являются и потомками элемента #posts, хотя и расположены от него на два уровня ниже. #posts .title найдет все 3 элемента с классом title.

!Внимание. Символ пробела " " является значащим в CSS-селекторах. Это важный символ, который разделяет описание предка и потомка. Если бы мы записали селектор #post2.title без пробела, то в данном примере не было найдено ни одного элемента. Такая запись означала бы, что мы хотим найти элемент, который одновременно содержит id "post2" и класс "title". Таким образом #post2 .title и #post2.title — это разные селекторы**.**

**Использование дочерних элементов**

Другой способ найти этот элемент:

#post2 > div.title

Здесь мы указали еще тег элемента divи уточнили, что нужно взять элемент с тегом и классом: div.title, который находится строго на один уровень иерархии ниже чем элемент #post2. Для этого используется символ >.

Элемент #post2 в этом случае называется **родителем** (англ. **parent**) для элементаdiv.title, а элемент div.title называется **дочерним элементом** (англ. **child**) для элемента #post2. Если символа > нет, то будет выполнен поиск всех элементов div.title на любом уровне ниже первого элемента.

!Внимание. В данном случае символы пробела вокруг символа ">" не несут важного значения в отличие от предыдущего примера, и могут быть опущены. Запись #post2>div.titleаналогична записи #post2 > div.title.

**Использование порядкового номера дочернего элемента**

Еще один способ найти этот элемент:

#posts > .item:nth-child(2) > .title

Псевдо-класс :nth-child(2) — позволяет найти второй по порядку элемент среди дочерних элементов для #posts. Затем с помощью конструкции > .title мы указываем, что нам нужен элемент .title, родителем которого является найденный ранее элемент .item.

**Использование нескольких классов**

Также мы можем использовать сразу несколько классов элемента, чтобы его найти. Для этого классы записываются подряд через точку: .title.second

Мы рассмотрели базовые селекторы, которых будет достаточно для написания простых UI-тестов. Если вы захотите разобраться подробнее в css-селекторах, то мы рекомендуем вам посмотреть следующие статьи:

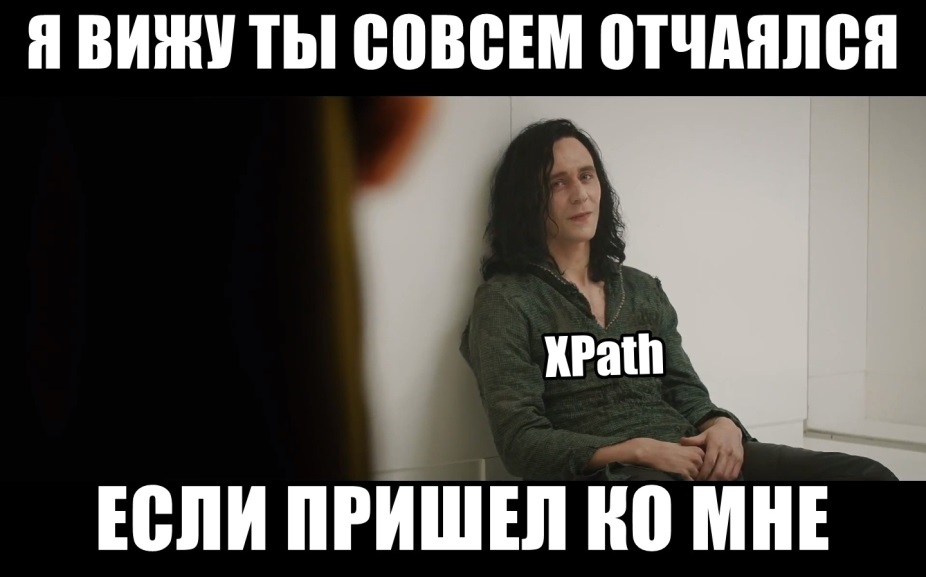
<https://learn.javascript.ru/css-selectors>

<https://www.w3schools.com/cssref/css_selectors.asp>

<https://developer.mozilla.org/ru/docs/Web/CSS/CSS_Selectors>

## Поиск элементов с помощью XPath

В работе с веб-страницами не всегда получается найти селектор, однозначно описывающий путь к нужному элементу. В такой ситуации для тестировщика лучшим решением проблемы будет пойти к фронтенд-разработчику проекта и договориться о специальном атрибуте, который будет использоваться в автотестах. Таким образом можно повысить тестируемость приложения. Увы, проекты бывают разные, и не всегда это возможно. И когда другого выхода больше нет, а автоматизировать как-то надо, можно обратиться к помощи языка запросов **XPath**.



На тему XPath мнения расходятся, но, как бы то ни было, это мощный и гибкий инструмент, который позволяет писать сложные запросы для поиска элементов.

Прежде всего, XPath (XML Path Language) это язык запросов, который использует древовидную структуру документа. Проверять XPath-запросы можно точно так же как и CSS-селекторы — в консоли разработчика. Откройте консоль на странице с котиками <http://suninjuly.github.io/cats.html>, и давайте на её примере разберемся в основах синтаксиса. Попробуйте вбить каждый из запросов-примеров в строку поиска, чтобы увидеть, что именно находит поисковый запрос.

### 1. XPath запрос всегда начинается с символа / или //

Символ / аналогичен символу > в CSS-селекторе, а символ // — пробелу. Их смысл:

* el1/el2 — выбирает элементы el2, являющиеся прямыми потомками el1;
* el1//el2 — выбирает элементы el2, являющиеся потомками el1 любой степени вложенности.

Разница состоит в том, что в XPath, когда мы начинаем запрос с символа /,  мы должны указать элемент, являющийся корнем нашего документа. Корнем всегда будет элемент с тегом <html>. Пример: /html/body/header

Мы можем начинать запрос и с символа //. Это будет означать, что мы хотим найти всех потомков корневого элемента без указания корневого элемента. В этом случае, для поиска того же хедера, мы можем выполнить запрос //header, так как других заголовков у нас нет.

Важно! Такой поиск может быть неоднозначным. Например, запрос //div вернет вам все элементы с тегом <div>. Избегайте неоднозначных ситуаций, они плохо влияют на здоровье ваших автотестов.

### 2. Символ [ ] — это команда фильтрации

Если по запросу найдено несколько элементов, то будет произведена фильтрация по правилу, указанному в скобках.

Правил фильтрации очень много:

* по любому **атрибуту**, будь то id, class, title (или любой другой). Например, мы хотим найти картинку с летящим котом, для этого можно выполнить запрос //img[@id='bullet']
* по **порядковому номеру**. Допустим, мы хотим выбрать вторую по порядку карточку с котом. Для этого найдем элемент с классом "row" и возьмем его второго потомка: //div[@class="row"]/div[2]
* по **полному совпадению текста.** Да, XPath — это единственный способ найти элемент по внутреннему тексту. Если мы хотим найти блок текста с котом-Лениным, можно воспользоваться XPath селектором //p[text()="Lenin cat"]. Такой селектор вернет элемент, только если текст полностью совпадет. Здесь важно сказать, что не всегда поиск по тексту — это хорошая практика, особенно в случае мультиязычных сайтов.
* по **частичному совпадению** текста или атрибута. Для этого нужна функция contains. Запрос //p[contains(text(), "cat")] вернет нам все абзацы текста, которые содержат слово cat. Точно так же можно искать по частичному совпадению других атрибутов, это удобно, если у элемента несколько классов. Посмотрите на код навбара сайта с котами. Его можно найти селектором //div[contains(@class, "navbar")]
* в фильтрации еще можно использовать булевы операции (and, or, not) и некоторые простые арифметические выражения (но вообще не стоит, наверное). Допустим, мы хотим найти картинку обязательно с data-type "animal" и именем "bullet-cat", для этого подойдет запрос: //img[@name='bullet-cat' and @data-type='animal']

### 3. Символ \* — команда выбора всех элементов

* Например можем найти текст в заголовке запросом //div/\*[@class="jumbotron-heading"]. Это может быть удобно, когда мы не знаем точно тег элемента, который ищем.

### 4. Поиск по классу в XPath регистрозависим

Также как и в случае поиска по CSS-селектором будьте внимательными к регистру при поиске по классам:

**//div/\*[@class="Jumbotron-heading"]** не найдет элемент на нашей странице.

Что важно знать про XPath, чтобы пользоваться им безболезненно:

* Не используйте селекторы вида //div[1]/div[2]/div[3] без крайней нужды: по такому селектору невозможно с первого раза понять, что за элемент вы ищете. А когда структура страницы хоть немного изменится, то ваш селектор с большой вероятностью перестанет работать;
* Если есть возможность использовать CSS-селекторы: сlass, id или name — лучше использовать их вместо поиска по XPath;
* Можно искать по полному или частичному совпадению текста или любого атрибута;
* Можно использовать булевы операции и простую арифметику;
* Можно удобно перемещаться по структуре документа (переходить к потомкам и к родителям);
* Подойдет, когда у сайта всё плохо с атрибутами и нет возможности достучаться до разработчиков;
* Есть мнение, что поиск по XPath в среднем медленнее, чем по css. Но достоверно это неизвестно;
* Не стоит использовать разные расширения для браузеров по поиску XPath: они подбирают нечитабельные и переусложненные селекторы. Лучше потратить немного времени и разобраться в синтаксисе самостоятельно, тем более, что он не очень сложный.

В курсе мы не будем работать с XPath-селекторами, и в основном будем использовать CSS. В случае необходимости можно познакомиться с XPath подробнее по следующим ссылкам:

<https://www.w3schools.com/xml/xpath_syntax.asp>

<https://msdn.microsoft.com/ru-ru/library/ms256086(v=vs.120).aspx>

<https://msiter.ru/tutorials/xpath/syntax>

<https://habr.com/post/114772/>

<https://testerslittlehelper.wordpress.com/2016/07/10/real-xpath/>

**Поиск элементов с помощью Selenium**

Для поиска элементов на странице в Selenium WebDriver используются несколько стратегий, позволяющих искать по атрибутам элементов, текстам в ссылках, CSS-селекторам и XPath-селекторам. Для поиска Selenium предоставляет метод find\_element, который принимает два аргумента - тип локатора и значение локатора. Существуют следующие методы поиска элементов:

* **find\_element(By.ID, value)** — поиск по уникальному атрибуту id элемента. Если ваши разработчики проставляют всем элементам в приложении уникальный id, то вам повезло, и вы чаще всего будет использовать этот метод, так как он наиболее стабильный;
* **find\_element(By.CSS\_SELECTOR, value)** — поиск элемента с помощью правил на основе CSS. Это универсальный метод поиска, так как большинство веб-приложений использует CSS для вёрстки и задания оформления страницам. Если find\_element\_by\_id вам не подходит из-за отсутствия id у элементов, то скорее всего вы будете использовать именно этот метод в ваших тестах;
* **find\_element(By.XPATH, value)** — поиск с помощью языка запросов XPath, позволяет выполнять очень гибкий поиск элементов;
* **find\_element(By.NAME, value)** — поиск по атрибуту name элемента;
* **find\_element(By.TAG\_NAME, value)** — поиск элемента по названию тега элемента;
* **find\_element(By.CLASS\_NAME, value)** — поиск по значению атрибута class;
* **find\_element(By.LINK\_TEXT, value)**— поиск ссылки на странице по полному совпадению;
* **find\_element(By.PARTIAL\_LINK\_TEXT, value)**— поиск ссылки на странице, если текст селектора совпадает с любой частью текста ссылки.

Например, мы хотим найти кнопку со значением id="submit\_button":

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/simple\_form\_find\_task.html")

button = browser.find\_element(By.ID, "submit")

Обратите внимание, что мы импортировали класс By, который содержит все возможные локаторы.

Если страница у вас загрузилась, но дальше ничего не происходит, вернитесь обратно в консоль, в которой вы запускали ваш скрипт. Скорее всего, вы увидите там ошибку **NoSuchElementException**. Она будет выглядеть следующим образом:

selenium.common.exceptions.NoSuchElementException: Message: no such element: Unable to locate element: {"method":"id","selector":"submit"}

Ошибка очевидна: мы неправильно указали локатор — значит, кнопки с таким id на странице нет.

Исправим локатор, чтобы наш код проходил без ошибок:

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/simple\_form\_find\_task.html")

button = browser.find\_element(By.ID, "submit\_button")

**Поиск нескольких элементов**

Вы можете столкнуться с ситуацией, когда на странице будет несколько элементов, подходящих под заданные вами параметры поиска. В этом случае WebDriver вернет вам только первый элемент, который встретит во время поиска по HTML. Если вам нужен не первый, а второй или следующие элементы, вам нужно либо задать более точный селектор для поиска, либо использовать методы **find\_elements**, которые мы рассмотрим чуть позже.

Иногда в статьях про Selenium WebDriver вы также будете встречать термин "локаторы", под которым подразумеваются стратегии поиска и значения, по которым должен выполняться поиск. Например, можно искать по локатору By.ID со значением "send\_button".

## Работа с браузером в Selenium

Если вы уже пробовали запускать примеры скриптов, то могли заметить, что браузер не всегда закрывается после выполнения кода. Поэтому обратите внимание на то, что необходимо явно закрывать окно браузера в нашем коде при помощи команды **browser.quit().** Каждый раз при открытии браузера browser = webdriver.Chrome() в системе создается процесс, который останется висеть, если вы вручную закроете окно браузера. Чтобы не остаться без оперативной памяти после запуска нескольких скриптов, всегда добавляйте к своим скриптам команду закрытия:

from selenium import webdriver

from selenium.webdriver.common.by import By

link = "http://suninjuly.github.io/simple\_form\_find\_task.html"

browser = webdriver.Chrome()

browser.get(link)

button = browser.find\_element(By.ID, "submit\_button")

button.click()

# закрываем браузер после всех манипуляций

browser.quit()

Важно еще пояснить разницу между двумя командами: **browser.close()** и**browser.quit()**. Какая между ними разница, ведь на первый взгляд обе они осуществляют одно и то же?

На самом деле, **browser.close()**закрывает текущее окно браузера. Это значит, что если ваш скрипт вызвал всплывающее окно, или открыл что-то в новом окне или вкладке браузера, то закроется только текущее окно, а все остальные останутся висеть. В свою очередь **browser.quit()**закрывает все окна, вкладки, и процессы вебдрайвера, запущенные во время тестовой сессии. Подробнее можно посмотреть здесь: [Difference between webdriver.Dispose(), .Close() and .Quit()](https://stackoverflow.com/questions/15067107/difference-between-webdriver-dispose-close-and-quit). Будьте внимательны с этими методами и, в общем случае, всегда используйте **browser.quit().**

Но что будет, если скрипт не дойдет до выполнения этого финального шага, а упадет с ошибкой где-то раньше?

Для того чтобы гарантировать закрытие, даже если произошла ошибка в предыдущих строках, проще всего использовать конструкцию **try/finally**:

from selenium import webdriver

from selenium.webdriver.common.by import By

link = "http://suninjuly.github.io/simple\_form\_find\_task.html"

try:

browser = webdriver.Chrome()

browser.get(link)

button = browser.find\_element(By.ID, "submit\_button")

button.click()

finally:

# закрываем браузер после всех манипуляций

browser.quit()

Можете попробовать запустить оба примера и обратить внимание на разницу.

Подробно говорить об обработке исключений мы сейчас не будем, здесь важно понимать только то, что даже если в коде внутри блока **try** произойдет какая-то ошибка, то код внутри блока **finally** выполнится в любом случае. Советуем добавлять такую обработку ко всем своим скриптам при выполнении задач этого и следующего модулей, а в третьем модуле мы обсудим более лаконичные конструкции.

Если хотите узнать больше про исключения, как их кидать, ловить и как с ними жить, то советуем к прохождению вот этот урок:  [Ошибки и исключения](https://stepik.org/lesson/24463/step/1?unit=6771).

(За замечание и дополнение спасибо за него нашему студенту [Михаилу λ](https://stepik.org/users/41632287))

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

link = "http://suninjuly.github.io/simple\_form\_find\_task.html"

try:

browser = webdriver.Chrome()

browser.get(link)

input1 = browser.find\_element(By.TAG\_NAME, "input")

input1.send\_keys("Ivan")

input2 = browser.find\_element(By.NAME, "last\_name")

input2.send\_keys("Petrov")

input3 = browser.find\_element(By.CLASS\_NAME, "city")

input3.send\_keys("Smolensk")

input4 = browser.find\_element(By.ID, "country")

input4.send\_keys("Russia")

button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

finally:

# успеваем скопировать код за 30 секунд

time.sleep(30)

# закрываем браузер после всех манипуляций

browser.quit()

# не забываем оставить пустую строку в конце файла

## Задание: поиск элемента по тексту в ссылке

В этой задаче мы попробуем искать элементы по тексту ссылки, для этого воспользуемся методом find\_element\_by\_link\_text:

link = browser.find\_element(By.LINK\_TEXT, text)

В качестве аргумента в метод передается такой текст, ссылку с которым мы хотим найти. Это тот самый текст, который содержится между открывающим и закрывающим тегом <a> вот тут </a>

Допустим, на странице <https://www.degreesymbol.net/> мы хотим найти ссылку с текстом "Degree symbol in Math" и перейти по ней. Если хотим найти элемент по полному соответствию текста, то нам подойдет такой код:

link = browser.find\_element(By.LINK\_TEXT, "Degree Symbol in Math")

link.click()

А если хотим найти элемент со ссылкой по подстроке, то нужно написать следующий код:

link = browser.find\_element(By.PARTIAL\_LINK\_TEXT, "Math")

link.click()

Обычно поиск по подстроке чуть более удобный и гибкий, но с ним надо быть вдвойне аккуратными и проверять, что находится нужный элемент.

### Задание

На указанной ниже странице вам нужно найти зашифрованную ссылку и кликнуть по ней:

1. Добавьте в самый верх своего кода import math
2. Добавьте в код команду, которая откроет страницу: <http://suninjuly.github.io/find_link_text>
3. Добавьте команду, которая найдет ссылку с текстом. Текст ссылки, который нужно найти, зашифрован формулой:

str(math.ceil(math.pow(math.pi, math.e)\*10000))

(можно вставить данное выражение в свой код, а можно выполнить в интерпретаторе, скопировать оттуда результат и уже его использовать в вашем коде)

1. Добавьте команду для клика по найденной ссылке: она перенесет вас на форму регистрации
2. Заполните скриптом форму так же как вы делали в предыдущем шаге урока
3. После успешного заполнения вы получите код - отправьте его в качестве ответа на это задание

**Важно!**Поиск по тексту ссылки бывает очень удобным, так часто тексты меняются реже, чем атрибуты элементов. Но лучше избегать такого метода поиска. Например, если приложение имеет несколько языков интерфейса, ваши тесты будут проходить только с определенным языком интерфейса. Применяйте этот метод с осторожностью и помните про возможные ограничения.

Читать больше:

<https://selenium-python.readthedocs.io/locating-elements.html#locating-hyperlinks-by-link-text>

РЕШЕНИЕ

from selenium import webdriver

from selenium.webdriver.common.by import By

import math

import time

link = "http://suninjuly.github.io/find\_link\_text"

#224592

try:

browser = webdriver.Chrome()

browser.get(link)

d = str(math.ceil(math.pow(math.pi, math.e)\*10000))

link1 = browser.find\_element(By.LINK\_TEXT, d)

link1.click()

input1 = browser.find\_element(By.TAG\_NAME, "input")

input1.send\_keys("Ivan")

input2 = browser.find\_element(By.NAME, "last\_name")

input2.send\_keys("Petrov")

input3 = browser.find\_element(By.CLASS\_NAME, "city")

input3.send\_keys("Smolensk")

input4 = browser.find\_element(By.ID, "country")

input4.send\_keys("Russia")

button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

finally:

# успеваем скопировать код за 30 секунд

time.sleep(30)

# закрываем браузер после всех манипуляций

browser.quit()

# не забываем оставить пустую строку в конце файла

## Поиск всех необходимых элементов с помощью find\_elements

Мы уже упоминали, что метод **find\_element** возвращает только первый из всех элементов, которые подходят под условия поиска. Иногда возникает ситуация, когда у нас есть несколько одинаковых по сути объектов на странице, например, иконки товаров в корзине интернет-магазина. В тесте нам нужно проверить, что отображаются все выбранные для покупки товары. Для этого существует метод **find\_elements**, которые в отличие от **find\_element** вернёт список всех найденных элементов по заданному условию. Проверив длину списка, мы можем удостовериться, что в корзине отобразилось правильное количество товаров. Пример кода (код приведен только для примера, сайта fake-shop.com скорее всего не существует):

# подготовка для теста

# открываем страницу первого товара

# данный сайт не существует, этот код приведен только для примера

browser.get("https://fake-shop.com/book1.html")

# добавляем товар в корзину

add\_button = browser.find\_element(By.CSS\_SELECTOR, ".add")

add\_button.click()

# открываем страницу второго товара

browser.get("https://fake-shop.com/book2.html")

# добавляем товар в корзину

add\_button = browser.find\_element(By.CSS\_SELECTOR, ".add")

add\_button.click()

# тестовый сценарий

# открываем корзину

browser.get("https://fake-shop.com/basket.html")

# ищем все добавленные товары

goods = browser.find\_elements(By.CSS\_SELECTOR, ".good")

# проверяем, что количество товаров равно 2

assert len(goods) == 2

!Важно. Обратите внимание на важную разницу в результатах, которые возвращают методы **find\_element** и **find\_elements**. Если первый метод не смог найти элемент на странице, то он вызовет ошибку **NoSuchElementException**, которая прервёт выполнение вашего кода. Второй же метод всегда возвращает валидный результат: если ничего не было найдено, то он вернёт пустой список и ваша программа перейдет к выполнению следующего шага в коде.

## Задание: использование метода find\_element****s****

В этой задаче вам нужно заполнить форму (<http://suninjuly.github.io/huge_form.html>). С помощью неё отдел маркетинга компании N захотел собрать подробную информацию о пользователях своего продукта. В награду за заполнение формы становится доступен код на скидку. Но маркетологи явно переусердствовали, добавив в форму 100 обязательных полей и ограничив время на ее заполнение. Теперь эта задача под силу только роботам ﻿🤖﻿.

Используйте WebDriver, метод **find\_elements**, нужные локатор и его значение. Введите полученный код в качестве ответа к этой задаче.

Используйте приведенный ниже шаблон: в цикле for мы можем последовательно взять каждый элемент из найденного списка текстовых полей и отправить произвольный текст в каждое поле. Если скрипт не успевает заполнить форму, выберите текст покороче.

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

try:

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/huge\_form.html")

elements = browser.find\_elements("вставьте сюда нужный локатор", "вставьте сюда нужное значение")

for element in elements:

element.send\_keys("Мой ответ")

button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

finally:

# успеваем скопировать код за 30 секунд

time.sleep(30)

# закрываем браузер после всех манипуляций

browser.quit()

# не забываем оставить пустую строку в конце файла

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

try:

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/huge\_form.html")

elements = browser.find\_elements(By.TAG\_NAME, "input")

for element in elements:

element.send\_keys("Мой ответ")

button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

finally:

# успеваем скопировать код за 30 секунд

time.sleep(30)

# закрываем браузер после всех манипуляций

browser.quit()

# не забываем оставить пустую строку в конце файла

## Задание: поиск элемента по XPath

На этот раз воспользуемся возможностью искать элементы по XPath.

На странице <http://suninjuly.github.io/find_xpath_form> вы найдете такую же форму регистрации, как в шаге 3, при этом в нее добавилась куча одинаковых кнопок отправки. Но сработает только кнопка с текстом "Submit", и наша задача нажать в коде именно на неё.

Ваши шаги:

1. В коде из [шага 4](https://stepik.org/lesson/138920/step/4) замените ссылку на  <http://suninjuly.github.io/find_xpath_form>.
2. Подберите уникальный XPath-селектор так, чтобы он находил только кнопку с текстом **Submit**. XPath можете формулировать как угодно (по тексту, по структуре, по атрибутам) - главное, чтобы он работал.
3. Модифицируйте код из шага 3 таким образом, чтобы поиск кнопки происходил с помощью XPath.
4. Запустите ваш код.

Если вы подобрали правильный селектор и все прошло хорошо, то вы получите код, который нужно отправить в качестве ответа на это задание.

РЕШЕНИЕ

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

link = "http://suninjuly.github.io/find\_xpath\_form"

try:

browser = webdriver.Chrome()

browser.get(link)

input1 = browser.find\_element(By.TAG\_NAME, "input")

input1.send\_keys("Ivan")

input2 = browser.find\_element(By.NAME, "last\_name")

input2.send\_keys("Petrov")

input3 = browser.find\_element(By.CLASS\_NAME, "city")

input3.send\_keys("Smolensk")

input4 = browser.find\_element(By.ID, "country")

input4.send\_keys("Russia")

button = browser.find\_element(By.XPATH, "//button[@type='submit']")

button.click()

finally:

# успеваем скопировать код за 30 секунд

time.sleep(5)

# закрываем браузер после всех манипуляций

browser.quit()

# не забываем оставить пустую строку в конце файла

## Уникальность селекторов: часть 1

Мы уже упоминали, что идеальный селектор — это такой селектор, который позволяет найти только один искомый элемент на странице. Благодаря уникальным селекторам наши тесты становятся стабильнее и меньше зависят от изменений в вёрстке страницы. Небольшие изменения разработчики делают достаточно часто, а мы бы не хотели постоянно исправлять наши тесты.

Другое важное замечание: хороший тест проверяет только маленькую, атомарную часть функциональности. Простые тесты, которые проверяют небольшой сценарий, лучше, чем один большой тест, проверяющий сразу много сценариев. Благодаря простым тестам мы быстрее локализуем место в продукте, где появился баг, а также можем найти одновременно несколько новых багов. Упавший большой автотест укажет только на первую встреченную проблему, так как он заканчивает работу при первой же найденной ошибке. В этом их отличие от ручных тестов, в которых мы, проверяя функциональность продукта по тест-кейсу, можем гибко обойти встречающиеся проблемы и пройти тест-кейс до конца, найдя все баги.

Рассмотрим следующий пример: у нас есть форма регистрации, в которой есть обязательные и необязательные поля для заполнения. Нужно проверить, что можно успешно зарегистрироваться на сайте.

**Сценарий плохого автотеста:**

1

* Открыть страницу с формой
* Заполнить все поля
* Нажать кнопку "Регистрация"
* Проверить, что есть сообщение об успешной регистрации

**Лучше разбить предыдущий тест на набор более простых автотестов:**

1

* Открыть страницу с формой
* Заполнить только обязательные поля
* Нажать кнопку "Регистрация"
* Проверить, что есть сообщение об успешной регистрации

2

* Открыть страницу с формой
* Заполнить все обязательные поля
* Заполнить все необязательные поля
* Нажать кнопку "Регистрация"
* Проверить, что есть сообщение об успешной регистрации

3

* Открыть страницу с формой
* Заполнить только необязательные поля
* Проверить, что кнопка "Регистрация" неактивна

## Уникальность селекторов: часть 2

Попробуем реализовать один из автотестов из предыдущего шага. Вам дана [страница](http://suninjuly.github.io/registration1.html) с формой регистрации. Проверьте, что можно зарегистрироваться на сайте, заполнив только обязательные поля, отмеченные символом \*: First name, last name, email. Текст для полей может быть любым. Успешность регистрации проверяется сравнением ожидаемого текста "Congratulations! You have successfully registered!" с текстом на странице, которая открывается после регистрации. Для сравнения воспользуемся стандартной конструкцией assert из языка Python.

Ниже дан шаблон кода, который вам нужно использовать для своего теста. Не забывайте, что селекторы должны быть уникальными.

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

try:

link = "http://suninjuly.github.io/registration1.html"

browser = webdriver.Chrome()

browser.get(link)

# Ваш код, который заполняет обязательные поля

...

# Отправляем заполненную форму

button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

# Проверяем, что смогли зарегистрироваться

# ждем загрузки страницы

time.sleep(1)

# находим элемент, содержащий текст

welcome\_text\_elt = browser.find\_element(By.TAG\_NAME, "h1")

# записываем в переменную welcome\_text текст из элемента welcome\_text\_elt

welcome\_text = welcome\_text\_elt.text

# с помощью assert проверяем, что ожидаемый текст совпадает с текстом на странице сайта

assert "Congratulations! You have successfully registered!" == welcome\_text

finally:

# ожидание чтобы визуально оценить результаты прохождения скрипта

time.sleep(10)

# закрываем браузер после всех манипуляций

browser.quit()

Углубимся немного в использовании конструкции assert из данного примера, Если результат проверки "Поздравляем! Вы успешно зарегистрировались!" == welcome\_text вернет значение False, то далее выполнится код **assert False**.Он бросит исключение AssertionError и номер строки, в которой произошла ошибка. Если код написан правильно и работал ранее, то такой результат равносилен тому, что наш автотест обнаружил баг в тестируемом веб-приложении. Если результат проверки вернет True, то выполнится выражение **assert True**. В этом случае код завершится без ошибок — тест прошел успешно. Подробнее про использование assert в коде мы поговорим в третьем модуле этого курса.

В этом задании нет автоматических проверок вашего кода. Просто убедитесь, что ваш тест проходит успешно, и вы не видите AssertionError в результатах работы вашего кода.

**Замечание**

В этом примере мы использовали метод **time.sleep(1)**, чтобы дождаться загрузки следующей страницы, прежде чем выполнять проверки. Если вы будете запускать код без этого метода, ваш код может внезапно упасть, хотя проходил ранее. Без использования такой паузы WebDriver может перейти к поиску тега h1 слишком рано, когда новая страница еще не загрузилась. В таком случаем будем видеть в терминале ошибку:

NoSuchElementException... Unable to locate element: {"method":"tag name","selector":"h1"}

Метод time.sleep(1) говорит Python подождать 1 секунду, прежде чем выполнять следующую строчку кода. Если вы всё равно видите эту ошибку, просто увеличьте количество секунд ожидания.

Проблема со своевременным поиском элемента — одна из самых больших проблем, которую приходится решать при разработке автотестов для UI. В условиях постоянно изменяющейся скорости сетевого соединения и неравномерности нагрузки на серверы скорость загрузки страницы может сильно варьироваться. Еще одним фактором, влияющим на стабильность работы тестов, является принцип асинхронности выполнения кода JavaScript. На простых страницах вы можете этого и не заметить, но на функционально богатых страницах время появления элементов страницы может быть непредсказуемо. Хорошо было бы организовать тесты так, чтобы не сложилось ситуации, когда они не проходят по причине нестабильной скорости интернета или других причин, которые от нас не зависят.

Решать эту проблему с помощью time.sleep() считается плохой практикой, так как заранее трудно указать нужное время ожидания. Если выставить слишком большое время ожидания, то тесты будут идти неоправданно долго. В дальнейших уроках мы рассмотрим более красивые и эффективные способы решения этой проблемы, а пока будем использовать time.sleep() из-за его простоты и наглядности.

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

try:

link = "http://suninjuly.github.io/registration1.html"

browser = webdriver.Chrome()

browser.get(link)

# Ваш код, который заполняет обязательные поля

...

#in1 = browser.find\_element(By.TAG\_NAME, "input")

#in1.send\_keys("ХУЙЭ1")

in1 = browser.find\_element(By.CSS\_SELECTOR, '[placeholder="Input your first name"]')

in1.send\_keys("ХУЙЭ1")

#in2 = browser.find\_element(By.XPATH, "//div[2]/input")

#in2.send\_keys("ХУЙЭ2")

in2 = browser.find\_element(By.CSS\_SELECTOR, '[placeholder="Input your last name"]')

in2.send\_keys("ХУЙЭ2")

in3 = browser.find\_element(By.CSS\_SELECTOR, "[placeholder='Input your email']")

in3.send\_keys("ХУЙЭ3")

# Отправляем заполненную форму

button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

# Проверяем, что смогли зарегистрироваться

# ждем загрузки страницы

time.sleep(1)

# находим элемент, содержащий текст

welcome\_text\_elt = browser.find\_element(By.TAG\_NAME, "h1")

# записываем в переменную welcome\_text текст из элемента welcome\_text\_elt

welcome\_text = welcome\_text\_elt.text

# с помощью assert проверяем, что ожидаемый текст совпадает с текстом на странице сайта

assert "Congratulations! You have successfully registered!" == welcome\_text

finally:

# ожидание чтобы визуально оценить результаты прохождения скрипта

time.sleep(10)

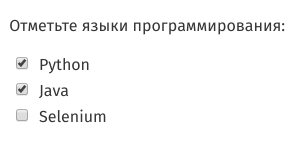
# закрываем браузер после всех манипуляций

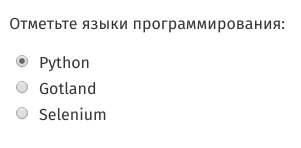
browser.quit()

**Как работать с элементами типа checkbox и radiobutton?**

Checkbox (чекбокс или флажок) и radiobutton (радиобаттон или переключатель) — часто используемые в формах элементы. Основная разница между ними состоит в том, что флажки позволяют выбирать/отключать любой из представленных вариантов, а переключатели позволяют выбрать только один из вариантов. Далее мы будем называть эти элементы на англоязычный манер: checkbox и radiobutton.

Так выглядят checkboxes:



А это radiobuttons:  
  
Оба этих элемента создаются при помощи тега input со значением атрибута type равным checkbox или radio соответственно. В html-коде страницы вы увидите:

<input type="checkbox">

<input type="radio">

Если checkbox или radiobutton выбран, то у элемента появится новый атрибут checked без значения. Часто атрибут checked уже установлен для одного из элементов по умолчанию.

<input type="checkbox" checked>

<input type="radio" checked>

Radiobuttons объединяются в группу, где все элементы имеют одинаковые значения атрибута name, но разные значения атрибута value:

<input type="radio" name="language" value="python" checked>

<input type="radio" name="language" value="selenium">

Checkboxes могут иметь как одинаковые, так и разные значения атрибута name. Поэтому и те, и другие лучше искать с помощью значения id или значения атрибута value. Если вы видите на странице чекбокс с уникальным значением name, то можете искать по name.

Чтобы снять/поставить галочку в элементе типа checkbox или выбрать опцию из группы radiobuttons, надо указать WebDriver метод поиска элемента и выполнить для найденного элемента метод click():

option1 = browser.find\_element(By.CSS\_SELECTOR, "[value='python']")

option1.click()

Также вы можете увидеть тег label рядом с input. Этот тег используется, чтобы сделать кликабельным текст, который отображается рядом с флажком. Этот текст заключен внутри тега label. Элемент label связывается с элементом input с помощью атрибута for, в котором указывается значение атрибута id для элемента input:

<div>

<input type="radio" id="python" name="language" checked>

<label for="python">Python</label>

</div>

<div>

<input type="radio" id="java" name="language">

<label for="java">Java</label>

</div>

В этом случае можно также отметить нужный пункт с помощью WebDriver, выполнив метод click() на элементе label.

option1 = browser.find\_element(By.CSS\_SELECTOR, "[for='java']")

## Задание: кликаем по checkboxes и radiobuttons (капча для роботов)

Продолжим использовать силу роботов 🤖 для решения повседневных задач. На данной [странице](https://suninjuly.github.io/math.html) мы добавили капчу для роботов, то есть тест, являющийся простым для компьютера, но сложным для человека.

Ваша программа должна выполнить следующие шаги:

1. Открыть страницу <https://suninjuly.github.io/math.html>.
2. Считать значение для переменной x.
3. Посчитать математическую функцию от x (код для этого приведён ниже).
4. Ввести ответ в текстовое поле.
5. Отметить checkbox "I'm the robot".
6. Выбрать radiobutton "Robots rule!".
7. Нажать на кнопку Submit.

Для этой задачи вам понадобится использовать атрибут .text для найденного элемента. Обратите внимание, что скобки здесь не нужны:

x\_element = browser.find\_element(By.CSS\_SELECTOR, selector\_value)

x = x\_element.text

y = calc(x)

Атрибут text возвращает текст, который находится между открывающим и закрывающим тегами элемента. Например, text для данного элемента <div class="message">У вас новое сообщение.</div> вернёт строку: "У вас новое сообщение".

Используйте функцию calc(), которая рассчитает и вернет вам значение функции, которое нужно ввести в текстовое поле. Не забудьте добавить этот код в начало вашего скрипта:

import math

def calc(x):

  return str(math.log(abs(12\*math.sin(int(x)))))

Если все сделано правильно и достаточно быстро (в этой задаче тоже есть ограничение по времени), вы увидите окно с числом. Скопируйте его в поле ниже.

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

import math

def calc(x):

return str(math.log(abs(12\*math.sin(int(x)))))

try:

link = "https://suninjuly.github.io/math.html"

browser = webdriver.Chrome()

browser.get(link)

x = browser.find\_element(By.ID, "input\_value")

x = int(x.text)

y = calc(x)

input\_y = browser.find\_element(By.ID, "answer")

input\_y.send\_keys(y)

browser.find\_element(By.ID, "robotCheckbox").click()

browser.find\_element(By.ID, "robotsRule").click()

browser.find\_element(By.TAG\_NAME, "button").click()

#print(x)

finally:

# ожидание чтобы визуально оценить результаты прохождения скрипта

time.sleep(10)

# закрываем браузер после всех манипуляций

browser.quit()

## Метод get\_attribute

Мы уже знаем, как найти нужный элемент на странице и как получить видимый пользователю текст. Для более детальных проверок в тесте нам может понадобиться узнать значение атрибута элемента. Атрибуты могут быть стандартными свойствами, которые понимает и использует браузер для отображения и вёрстки элементов или для хранения служебной информации, например, name, width, height, color и многие [другие](https://www.w3schools.com/tags/ref_attributes.asp). Также атрибуты могут быть созданы разработчиками проекта для задания собственных стилей или правил.

Значение атрибута представляет собой строку. Если значение атрибута отсутствует, то это равносильно значению атрибута равному "false". Давайте еще раз взглянем на страницу <http://suninjuly.github.io/math.html>. На ней есть radiobuttons, для которых выбрано значение по умолчанию. В автотесте нам может понадобиться проверить, что для одного из radiobutton по умолчанию уже выбрано значение. Для этого мы можем проверить значение атрибута checked у этого элемента. Вот HTML-код элемента:

<input class="check-input" type="radio" name="ruler" id="peopleRule" value="people" checked>

Найдём этот элемент с помощью WebDriver:

people\_radio = browser.find\_element(By.ID, "peopleRule")

Найдём атрибут "checked" с помощью встроенного метода get\_attribute и проверим его значение:

people\_checked = people\_radio.get\_attribute("checked")

print("value of people radio: ", people\_checked)

assert people\_checked is not None, "People radio is not selected by default"

Т.к. у данного атрибута значение не указано явно, то метод get\_attribute вернёт "true". Возможно, вы заметили, что "true" написано с маленькой буквы, — все методы WebDriver взаимодействуют с браузером с помощью JavaScript, в котором булевые значения пишутся с маленькой буквы, а не с большой, как в Python.

Мы можем написать проверку другим способом, сравнив строки:

assert people\_checked == "true", "People radio is not selected by default"

Если атрибута нет, то метод get\_attribute вернёт значение **None**. Применим метод get\_attribute ко второму radiobutton, и убедимся, что атрибут отсутствует.

robots\_radio = browser.find\_element(By.ID, "robotsRule")

robots\_checked = robots\_radio.get\_attribute("checked")

assert robots\_checked is None

Так же мы можем проверять наличие атрибута disabled, который определяет, может ли пользователь взаимодействовать с элементом. Например, в предыдущем задании на странице с капчей для роботов JavaScript устанавливает атрибут disabled у кнопки **Submit**, когда истекает время, отведенное на решение задачи.

<button type="submit" class="btn btn-default" disabled>Submit</button>

## Задание: поиск сокровища с помощью get\_attribute

В данной задаче вам нужно с помощью роботов решить ту же математическую задачу, как и в прошлом задании. Но теперь значение переменной **х** спрятано в "сундуке", точнее, значение хранится в атрибуте **valuex** у картинки с изображением сундука.

Ваша программа должна:

1. Открыть страницу <http://suninjuly.github.io/get_attribute.html>.
2. Найти на ней элемент-картинку, который является изображением сундука с сокровищами.
3. Взять у этого элемента значение атрибута **valuex**, которое является значением x для задачи.
4. Посчитать математическую функцию от x (сама функция остаётся неизменной).
5. Ввести ответ в текстовое поле.
6. Отметить checkbox "I'm the robot".
7. Выбрать radiobutton "Robots rule!".
8. Нажать на кнопку "Submit".

Для вычисления значения выражения в п.4 используйте функцию calc(x) из предыдущей задачи.

Если все сделано правильно и достаточно быстро (в этой задаче тоже есть ограничение по времени), вы увидите окно с числом. Скопируйте его в поле ниже и нажмите кнопку "Submit", чтобы получить баллы за задание.

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

import math

def calc(x):

return str(math.log(abs(12\*math.sin(int(x)))))

try:

link = "http://suninjuly.github.io/get\_attribute.html"

browser = webdriver.Chrome()

browser.get(link)

treasure = browser.find\_element(By.ID, "treasure")

number = treasure.get\_attribute("valuex")

print(number)

val = calc(number)

input\_y = browser.find\_element(By.ID, "answer")

input\_y.send\_keys(val)

browser.find\_element(By.ID, "robotCheckbox").click()

browser.find\_element(By.ID, "robotsRule").click()

browser.find\_element(By.TAG\_NAME, "button").click()

#print(x)

# Отправляем заполненную форму

'''button = browser.find\_element(By.CSS\_SELECTOR, "button.btn")

button.click()

# Проверяем, что смогли зарегистрироваться

# ждем загрузки страницы

time.sleep(1)

# находим элемент, содержащий текст

welcome\_text\_elt = browser.find\_element(By.TAG\_NAME, "h1")

# записываем в переменную welcome\_text текст из элемента welcome\_text\_elt

welcome\_text = welcome\_text\_elt.text

# с помощью assert проверяем, что ожидаемый текст совпадает с текстом на странице сайта

assert "Congratulations! You have successfully registered!" == welcome\_text

'''

finally:

# ожидание чтобы визуально оценить результаты прохождения скрипта

time.sleep(10)

# закрываем браузер после всех манипуляций

browser.quit()

## Работа со списками

На веб-страницах мы также встречаем раскрывающиеся (выпадающие) списки. У таких списков есть несколько важных особенностей:

1. У каждого элемента списка обычно есть уникальное значение атрибута value
2. В списках может быть разрешено выбирать как только один, так и несколько вариантов, в зависимости от типа списка
3. Визуально списки могут различаться тем, что в одном случае все варианты скрыты в выпадающем меню (<http://suninjuly.github.io/selects1.html>), а в другом все варианты или их часть видны (<http://suninjuly.github.io/selects2.html>)

Но для взаимодействия с любым вариантом списка мы будем использовать одни и те же методы Selenium.

Посмотрим, как выглядит html для списка:

<label for="dropdown">Выберите язык программирования:</label>

<select id="dropdown" class="custom-select">

 <option selected>--</option>

 <option value="1">Python</option>

 <option value="2">Java</option>

 <option value="3">JavaScript</option>

</select>

Варианты ответа задаются тегом option, значение value может отсутствовать. Можно отмечать варианты с помощью обычного метода click(). Для этого сначала нужно применить метод click() для элемента с тегом select, чтобы список раскрылся, а затем кликнуть на нужный вариант ответа:

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

browser.get(link)

browser.find\_element(By.TAG\_NAME, "select").click()

browser.find\_element(By.CSS\_SELECTOR, "option:nth-child(2)").click()

Последняя строчка может выглядеть и так:

browser.find\_element(By.CSS\_SELECTOR, "[value='1']").click()

Это не самый удобный способ, так как нам приходится делать лишний клик для открытия списка.

Есть более удобный способ, для которого используется специальный класс **Select** из библиотеки WebDriver. Вначале мы должны инициализировать новый объект, передав в него WebElement с тегом select. Далее можно найти любой вариант из списка с помощью метода **select\_by\_value(value):**

from selenium.webdriver.support.ui import Select

select = Select(browser.find\_element(By.TAG\_NAME, "select"))

select.select\_by\_value("1") # ищем элемент с текстом "Python"

Можно использовать еще два метода: **select.select\_by\_visible\_text("text")** и **select.select\_by\_index(index)**. Первый способ ищет элемент по видимому тексту, например, **select.select\_by\_visible\_text("Python")** найдёт "Python" для нашего примера.

Второй способ ищет элемент по его индексу или порядковому номеру. Индексация начинается с нуля. Для того чтобы найти элемент с текстом "Python", нужно использовать **select.select\_by\_index(1)**, так как опция с индексом 0 в данном примере имеет значение по умолчанию равное "--".

## Задание: работа с выпадающим списком

Для этой задачи мы придумали еще один вариант капчи для роботов. Придется немного переобучить нашего робота, чтобы он справился с новым заданием.

Напишите код, который реализует следующий сценарий:

1. Открыть страницу <http://suninjuly.github.io/selects1.html>
2. Посчитать сумму заданных чисел
3. Выбрать в выпадающем списке значение равное расчитанной сумме
4. Нажать кнопку "Submit"

Если все сделано правильно и достаточно быстро (в этой задаче тоже есть ограничение по времени), вы увидите окно с числом. Отправьте полученное число в качестве ответа для этого задания.

Когда ваш код заработает, попробуйте запустить его на странице <http://suninjuly.github.io/selects2.html>. Ваш код и для нее тоже ﻿должен пройти успешно.

Подсказка: если вы получаете ошибку в духе "argument of type 'int' is not iterable", перепроверьте тип переменной, которую вы передаете в функцию поиска. Нужно передать строку!

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

#import math

from selenium.webdriver.support.ui import Select

try:

link = "http://suninjuly.github.io/selects1.html"

browser = webdriver.Chrome()

browser.get(link)

summ = int(browser.find\_element(By.ID, "num1").text) + int(browser.find\_element(By.ID, "num2").text)

select = Select(browser.find\_element(By.TAG\_NAME, "select"))

select.select\_by\_value(str(summ)) # ищем элемент с текстом суммы цифр num1 and num2

browser.find\_element(By.TAG\_NAME, "button").click()

finally:

# ожидание чтобы визуально оценить результаты прохождения скрипта

time.sleep(10)

# закрываем браузер после всех манипуляций

browser.quit()

## Метод execute\_script

Рассмотрим еще один очень полезный и мощный метод, но он требует хотя бы минимальных знаний JavaScript. С помощью метода execute\_script можно выполнить программу, написанную на языке JavaScript, как часть сценария автотеста в запущенном браузере. Зачем это может понадобиться, если в автотестах мы стараемся взаимодействовать с интерфейсом сайта как обычный пользователь, нажимая кнопки, выбирая пункты меню и вводя текст в текстовые поля?

Дело в том, что стандартные методы, доступные в Selenium, не могут покрыть всех возможных ситуаций работы с веб-приложением. Сайты в интернете могут решать самые разные задачи, начиная от простого блога до сложных финансовых или графических приложений. Разработчики имеют доступ к огромному количеству различных библиотек для решения бизнес-сценариев, что приводит к появлению на веб-странице нестандартных редакторов текстов, уникальных меню, оригинальных видео-плееров и т.д. Порой это приводит к тому, что для нажатия вроде бы обычной кнопки тестировщику понадобится писать настоящий JavaScript-сценарий. Если вы столкнулись с такой ситуацией, то в первую очередь обратитесь за помощью к вашим фронтенд-разработчикам, чтобы они подсказали  пример нужного скрипта. Прежде чем использовать данный скрипт в тестах, вы можете проверить, как он работает прямо в браузере, выполнив код в консоли браузера. Затем можете добавить его в ваш автотест с помощью execute\_script(javascript\_code).

Давайте попробуем вызвать alert в браузере с помощью WebDriver. Пример сценария:

from selenium import webdriver

browser = webdriver.Chrome()

browser.execute\_script("alert('Robots at work');")

Обратите внимание, что исполняемый JavaScript нужно заключать в кавычки (двойные или одинарные). Если внутри скрипта вам также понадобится использовать кавычки, а для выделения скрипта вы уже используете двойные кавычки, то в скрипте следует поставить одинарные:

browser.execute\_script("document.title='Script executing';")

Такой формат записи тоже будет работать:

browser.execute\_script('document.title="Script executing";')

Можно с помощью этого метода выполнить сразу несколько инструкций, перечислив их через точку с запятой. Изменим сначала заголовок страницы, а затем вызовем alert:

browser.execute\_script("document.title='Script executing';alert('Robots at work');")

## Пример задачи для execute\_script

Давайте теперь рассмотрим реальную ситуацию, когда пользователь должен кликнуть на элемент, который внезапно оказывается перекрыт другим элементом на странице.

Для клика в WebDriver мы используем метод click(). Если элемент оказывается перекрыт другим элементом, то наша программа вызовет следующую ошибку:

selenium.common.exceptions.WebDriverException: Message: unknown error: Element <button type="submit" class="btn btn-default" style="margin-bottom: 1000px;">...</button> is not clickable at point (87, 420). Other element would receive the click: <p>...</p>

Из описания ошибки можно понять, что указанный нами элемент нельзя кликнуть в данной точке, т.к. клик произойдёт на другом элементе с тегом <p>.

Чтобы увидеть пример данной ошибки, запустите следующий скрипт:

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

link = "https://SunInJuly.github.io/execute\_script.html"

browser.get(link)

button = browser.find\_element(By.TAG\_NAME, "button")

button.click()

Теперь вы можете сами посмотреть на эту [страницу](https://SunInJuly.github.io/execute_script.html) и увидеть, что огромный футер действительно перекрывает нужную нам кнопку. Футером (footer) называется нижний блок, который обычно одинаков для всех страниц сайта. Чтобы понять, как решить эту проблему, нужно разобраться, как работает метод **click()**.

В первую очередь WebDriver проверит, что ширина и высота элемента больше 0, чтобы по нему можно было кликнуть.

Затем, если элемент находится за границей окна браузера, WebDriver автоматически проскроллит страницу, чтобы элемент попал в область видимости, то есть не находился за границей экрана. Но это не гарантирует того, что элемент не перекрыт другим элементом, который тоже находится в области видимости.

А в какую точку элемента будет происходить клик? Selenium рассчитывает координаты центра элемента и производит клик в вычисленную точку. Это тоже приведёт к ошибке, если часть элемента всё-таки видна, но элемент перекрыт больше чем на половину своей высоты или ширины.

Если мы столкнулись с такой ситуацией, мы можем заставить браузер дополнительно проскроллить нужный элемент, чтобы он точно стал видимым.  
Делается это с помощью следующего скрипта:

"return arguments[0].scrollIntoView(true);"

Мы дополнительно передали в метод scrollIntoView аргумент true, чтобы элемент после скролла оказался в области видимости. Другие возможные параметры метода можно посмотреть здесь: <https://developer.mozilla.org/ru/docs/Web/API/Element/scrollIntoView>

В итоге, чтобы кликнуть на перекрытую кнопку, нам нужно выполнить следующие команды в коде:

button = browser.find\_element\_by\_tag\_name("button")

browser.execute\_script("return arguments[0].scrollIntoView(true);", button)

button.click()

В метод execute\_script мы передали текст js-скрипта и найденный элемент button, к которому нужно будет проскроллить страницу. После выполнения кода элемент button должен оказаться в верхней части страницы. Подробнее о методе см <https://developer.mozilla.org/ru/docs/Web/API/Element/scrollIntoView> .

Также можно проскроллить всю страницу целиком на строго заданное количество пикселей. Эта команда проскроллит страницу на 100 пикселей вниз:

browser.execute\_script("window.scrollBy(0, 100);")

!Важно. Мы не будем в этом курсе изучать, как работает JavaScript, и обойдемся только приведенным выше примером скрипта с прокруткой страницы. Для сравнения приведем скрипт на этом языке, который делает то же, что приведенный выше пример для WebDriver:

// javascript

button = document.getElementsByTagName("button")[0];

button.scrollIntoView(true);

Можете попробовать исполнить его в консоли браузера на странице <http://suninjuly.github.io/execute_script.html>. Для этого откройте инструменты разработчика в браузере, перейдите на вкладку **консоль (console)**, скопируйте туда этот код и нажмите Enter. Таким образом можно протестировать кусочки js кода прежде чем внедрять его в свои тесты на python.

Обратите внимание, что в коде в WebDriver нужно использовать ключевое слово **return**. Также его нужно будет использовать, когда вы захотите получить какие-то данные после выполнения скрипта. При этом при тестировании скрипта в консоли браузера слово **return** использовать не надо.

**2.2 Работа с файлами, списками и js-скриптами**

8 из 8 шагов пройдено

7 из 7 баллов  получено

## Задание на execute\_script

В данной задаче вам нужно будет снова преодолеть капчу для роботов и справиться с ужасным и огромным футером, который дизайнер всё никак не успевает переделать. Вам потребуется написать код, чтобы:

1. Открыть страницу <http://SunInJuly.github.io/execute_script.html>.
2. Считать значение для переменной x.
3. Посчитать математическую функцию от x.
4. Проскроллить страницу вниз.
5. Ввести ответ в текстовое поле.
6. Выбрать checkbox "I'm the robot".
7. Переключить radiobutton "Robots rule!".
8. Нажать на кнопку "Submit".

Если все сделано правильно и достаточно быстро (в этой задаче тоже есть ограничение по времени), вы увидите окно с числом. Отправьте полученное число в качестве ответа для этого задания.

Для этой задачи вам понадобится использовать метод execute\_script, чтобы сделать прокрутку в область видимости элементов, перекрытых футером.

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import math

import time

def calc(x):

return str(math.log(abs(12\*math.sin(int(x)))))

try:

browser = webdriver.Chrome()

link = "http://SunInJuly.github.io/execute\_script.html"

browser.get(link)

x = browser.find\_element(By.ID, "input\_value").text

answer = calc(int(x))

#!!!!!!!!!!

ans = browser.find\_element(By.ID, "answer")

#второй аргумент в функции - переменная - без кавычек!!

browser.execute\_script("return arguments[0].scrollIntoView(true);", ans)

browser.find\_element(By.ID, "answer").send\_keys(answer)

r\_box = browser.find\_element(By.ID, "robotCheckbox")

browser.execute\_script("return arguments[0].scrollIntoView(true);", r\_box)

r\_box.click()

browser.find\_element(By.ID, "robotsRule").click()

browser.find\_element(By.TAG\_NAME, "button").click()

time.sleep(5)

finally:

browser.quit()

**2.2 Работа с файлами, списками и js-скриптами**

8 из 8 шагов пройдено

7 из 7 баллов  получено

## Загрузка файлов

﻿Если нам понадобится загрузить файл на веб-странице, мы можем использовать уже знакомый нам метод send\_keys. Только теперь нам нужно в качестве аргумента передать путь к нужному файлу на диске вместо простого текста.

Чтобы указать путь к файлу, можно использовать стандартный модуль Python для работы с операционной системой — **os**. В этом случае ваш код не будет зависеть от операционной системы, которую вы используете. Добавление файла будет работать и на Windows, и на Linux, и даже на MaсOS.

Пример кода, который позволяет указать путь к файлу **'file.txt**', находящемуся в той же папке, что и скрипт, который вы запускаете:

import os

current\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_)) # получаем путь к директории текущего исполняемого файла

file\_path = os.path.join(current\_dir, 'file.txt') # добавляем к этому пути имя файла

element.send\_keys(file\_path)

Попробуйте добавить в файл отдельно команды **print(os.path.abspath(\_\_file\_\_))** и **print(os.path.abspath(os.path.dirname(\_\_file\_\_)))** и посмотрите на разницу. Подробнее о методах модуля **os** можете почитать самостоятельно в документации: <https://docs.python.org/3/library/os.path.html>. Обратите внимание, что это будет работать только при запуске кода из файла, в интерпретаторе не сработает.

Если совсем непонятно что происходит, пример:

Допустим, мы написали код скрипта и сохранили код в lesson2\_step7.py в своей локальной папке D:\stepik\_homework. Активируем виртуальное окружение и запускаем его **python lesson2\_step7.py.** В таком случае конструкция **os.path.abspath(os.path.dirname(\_\_file\_\_))**вернет нам путь до директории файла с кодом, то есть D:\stepik\_homework**.**В эту же папку кладем файл, который хотим прикрепить, то есть file.txt. Тогда, после выполнения команды:

file\_path = os.path.join(current\_dir, 'file.txt')

В переменной file\_path будет полный путь к файлу '**D:\stepik\_homework\file.txt'**. Фишка в том, что если мы файлы lesson2\_step7.py вместе с file.txt перенесем в другую папку, или на компьютер с другой ОС, то такой код без правок заработает и там.

Элемент в форме, который выглядит, как кнопка добавления файла, имеет атрибут **type="file"**. Мы должны сначала найти этот элемент с помощью селектора, а затем применить к нему метод **send\_keys(file\_path)**.

## Задание: загрузка файла

В этом задании в форме регистрации требуется загрузить текстовый файл.

Напишите скрипт, который будет выполнять следующий сценарий:

1. Открыть страницу <http://suninjuly.github.io/file_input.html>
2. Заполнить текстовые поля: имя, фамилия, email
3. Загрузить файл. Файл должен иметь расширение .txt и может быть пустым
4. Нажать кнопку "Submit"

Если все сделано правильно и быстро, вы увидите окно с числом. Отправьте полученное число в качестве ответа для этого задания.

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import math

import time

import os

def calc(x):

return str(math.log(abs(12\*math.sin(int(x)))))

print(os.path.abspath(\_\_file\_\_))

print(os.path.abspath(os.path.dirname(\_\_file\_\_)))

try:

browser = webdriver.Chrome()

link = "http://suninjuly.github.io/file\_input.html"

browser.get(link)

in1 = browser.find\_element(By.NAME, "firstname")

in1.send\_keys("Duke")

in2 = browser.find\_element(By.NAME, "lastname")

in2.send\_keys("Nukem")

in3 = browser.find\_element(By.NAME, "email")

in3.send\_keys("123@mail.com")

current\_dir = os.path.abspath(os.path.dirname(\_\_file\_\_))

file\_path = os.path.join(current\_dir, 'file1.txt')

in4 = browser.find\_element(By.NAME, "file")

in4.send\_keys(file\_path)

button = browser.find\_element(By.TAG\_NAME, "button")

button.click()

time.sleep(8)

finally:

browser.quit()

**2.3 Работа с окнами**

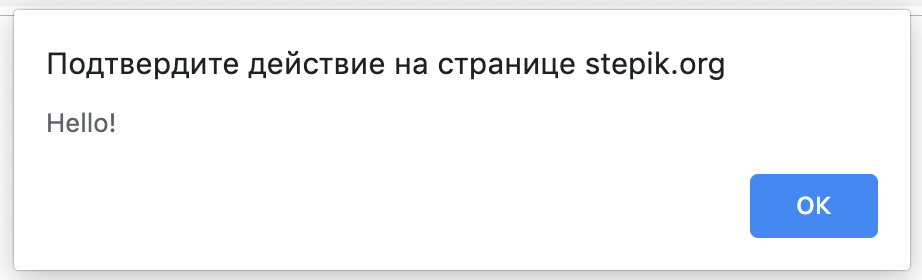
6 из 6 шагов пройдено

5 из 5 баллов  получено

## Alerts и как с ними жить

Мы уже встречали alert в нашем курсе, когда получали число-ответ в задачах. Также мы узнали, что можно самостоятельно вызвать alert с помощью JavaScript:

alert('Hello!');



Теперь рассмотрим ситуацию, когда в сценарии теста возникает необходимость не только получить содержимое alert, но и нажать кнопку OK, чтобы закрыть alert. **Alert** является модальным окном: это означает, что пользователь не может взаимодействовать дальше с интерфейсом, пока не закроет alert. Для этого нужно сначала переключиться на окно с alert, а затем принять его с помощью команды **accept()**:

alert = browser.switch\_to.alert

alert.accept()

Чтобы получить текст из alert, используйте свойство text объекта alert:

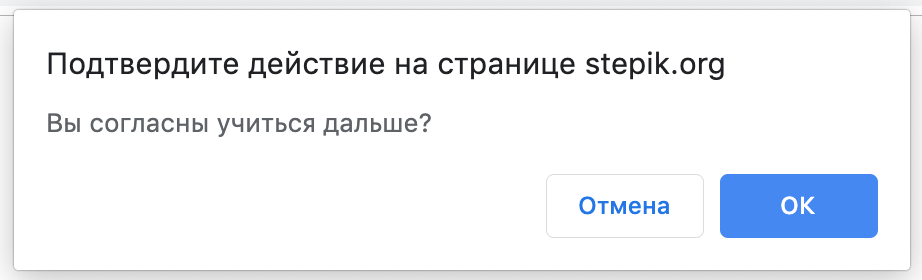
alert = browser.switch\_to.alert

alert\_text = alert.text

Другой вариант модального окна, который предлагает пользователю выбор согласиться с сообщением или отказаться от него, называется **confirm**. Для переключения на окно confirmиспользуется та же команда, что и в случае с alert:

confirm = browser.switch\_to.alert

confirm.accept()



Для confirm**-**окон можно использовать следующий метод для отказа:

confirm.dismiss()

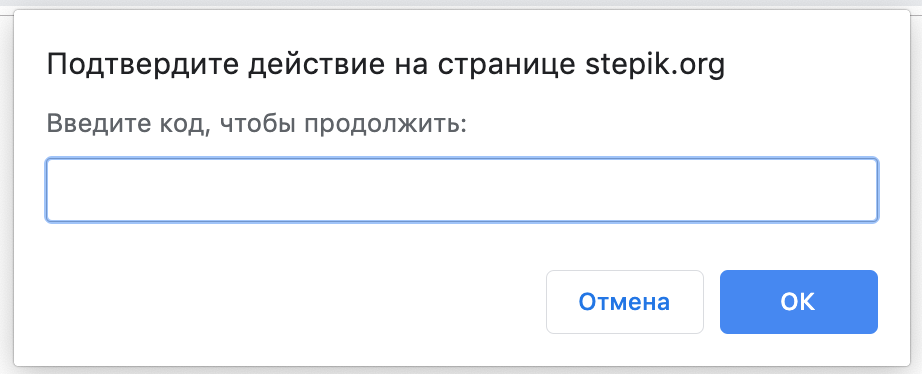
То же самое, что и при нажатии пользователем кнопки "Отмена".

Третий вариант модального окна — **prompt**— имеет дополнительное поле для ввода текста. Чтобы ввести текст, используйте метод **send\_keys()**:

prompt = browser.switch\_to.alert

prompt.send\_keys("My answer")

prompt.accept()



## Задание: принимаем alert

В этой задаче вам нужно написать программу, которая будет выполнять следующий сценарий:

1. Открыть страницу <http://suninjuly.github.io/alert_accept.html>
2. Нажать на кнопку
3. Принять confirm
4. На новой странице решить капчу для роботов, чтобы получить число с ответом

Если все сделано правильно и достаточно быстро (в этой задаче тоже есть ограничение по времени), вы увидите окно с числом. Отправьте полученное число в качестве ответа на это задание.

**РЕШЕНИЕ**

**from selenium import webdriver**

**from selenium.webdriver.common.by import By**

**import math**

**import time**

**import os**

**def calc(x):**

**return str(math.log(abs(12\*math.sin(int(x)))))**

**try:**

**browser = webdriver.Chrome()**

**link = "http://suninjuly.github.io/alert\_accept.html"**

**browser.get(link)**

**button = browser.find\_element(By.CSS\_SELECTOR, '[type="submit"]')**

**button.click()**

**confirm = browser.switch\_to.alert**

**confirm.accept()**

**x = int(browser.find\_element(By.CSS\_SELECTOR, '[id="input\_value"]').text)**

**y = calc(x)**

**in1 = browser.find\_element(By.ID, 'answer')**

**in1.send\_keys(y)**

**button = browser.find\_element(By.TAG\_NAME, "button")**

**button.click()**

**time.sleep(5)**

**finally:**

**browser.quit()**

## Переход на новую вкладку браузера

При работе с веб-приложениями приходится переходить по ссылкам, которые открываются в новой вкладке браузера. WebDriver может работать только с одной вкладкой браузера. При открытии новой вкладки WebDriver продолжит работать со старой вкладкой. Для переключения на новую вкладку надо явно указать, на какую вкладку мы хотим перейти. Это делается с помощью команды switch\_to.window:

browser.switch\_to.window(window\_name)

Чтобы узнать имя новой вкладки, нужно использовать метод window\_handles, который возвращает массив имён всех вкладок. Зная, что в браузере теперь открыто две вкладки, выбираем вторую вкладку:

new\_window = browser.window\_handles[1]

Также мы можем запомнить имя текущей вкладки, чтобы иметь возможность потом к ней вернуться:

first\_window = browser.window\_handles[0]

После переключения на новую вкладку поиск и взаимодействие с элементами будут происходить уже на новой странице.

## Задание: переход на новую вкладку

В этом задании после нажатия кнопки страница откроется в новой вкладке, нужно переключить WebDriver на новую вкладку и решить в ней задачу.

Сценарий для реализации выглядит так:

1. Открыть страницу <http://suninjuly.github.io/redirect_accept.html>
2. Нажать на кнопку
3. Переключиться на новую вкладку
4. Пройти капчу для робота и получить число-ответ

Если все сделано правильно и достаточно быстро (в этой задаче тоже есть ограничение по времени), вы увидите окно с числом. Отправьте полученное число в качестве ответа на это задание.

**РЕШЕНИЕ**

from selenium import webdriver

from selenium.webdriver.common.by import By

import math

import time

import os

def calc(x):

return str(math.log(abs(12\*math.sin(int(x)))))

"""

print(os.path.abspath(\_\_file\_\_))

print(os.path.abspath(os.path.dirname(\_\_file\_\_)))

"""

try:

browser = webdriver.Chrome()

link = "http://suninjuly.github.io/redirect\_accept.html"

browser.get(link)

first\_window = browser.window\_handles[0]

button = browser.find\_element(By.TAG\_NAME, "button")

button.click()

new\_window = browser.window\_handles[1]

browser.switch\_to.window(new\_window)

"""

browser.switch\_to.window(first\_window)

"""

x = int(browser.find\_element(By.ID, "input\_value").text)

y = calc(x)

in1 = browser.find\_element(By.ID, "answer")

in1.send\_keys(y)

button = browser.find\_element(By.TAG\_NAME, "button")

button.click()

time.sleep(5)

finally:

browser.quit()

## 2.4 Настройка ожиданий

## Немного про современный веб

Разработчики хорошо потрудились, чтобы в 2022 году веб-страницы выглядели красиво и быстро открывались, а переходы между страницами были практически незаметны. Страницы сайтов интерактивны и мгновенно реагируют на действия пользователя. Для реализации такого комфортного пользовательского опыта чаще всего используют подход [Single-Page Application](https://en.wikipedia.org/wiki/Single-page_application) (или [одностраничных приложений](https://ru.wikipedia.org/wiki/%D0%9E%D0%B4%D0%BD%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5)), что в общем случае означает наличие одной страницы на сайте. Содержимое страницы при этом динамически обновляется с помощью JavaScript, который незаметно обменивается с сервером информацией, например, посредством REST API.

В целом все довольны. Разве что создателям автотестов на интерфейсы приходится туго. Неожиданно появляющиеся или пропадающие элементы на странице, непредсказуемое время полной отрисовки страницы, изменяющийся текст в кнопках или в сообщениях веб-сайта — эти особенности работы SPA-приложений приходится учитывать в автотестах, и, стоит признать, это является одним из самых сложных и головоломных аспектов разработки автотестов на Selenium (да и в других фреймворках для написания end-to-end тестов тоже).

В этом уроке мы рассмотрим подробнее самые распространенные проблемы и познакомимся со способами их решения.

## Как работают методы get и find\_element

Разберем еще один простой тест на WebDriver, проверяющий работу кнопки.

Тестовый сценарий выглядит так:

1. Открыть страницу <http://suninjuly.github.io/wait1.html>
2. Нажать на кнопку "Verify"
3. Проверить, что появилась надпись "Verification was successful!"

Для открытия страницы мы используем метод get, затем находим нужную кнопку с помощью одного из методов find\_element\_by\_ и нажимаем на нее с помощью метода click. Далее находим новый элемент с текстом и проверяем соответствие текста на странице ожидаемому тексту.

Вот как выглядит код автотеста:

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/wait1.html")

button = browser.find\_element(By.ID, "verify")

button.click()

message = browser.find\_element(By.ID, "verify\_message")

assert "successful" in message.text

Попробуйте сначала выполнить тест вручную, а затем запустить автотест. В первом случае, вы завершите тест успешно, во втором случае автотест упадет с сообщением NoSuchElementException для элемента c **id="verify"**. Почему так происходит?

Команды в Python выполняются синхронно, то есть, строго последовательно. Пока не завершится команда get, не начнется поиск кнопки. Пока кнопка не найдена, не будет сделан клик по кнопке и так далее.

Но тест будет работать абсолютно стабильно, только если в данной веб-странице не используется JavaScript (что маловероятно для современного веба). Метод get дожидается информации от браузера о том, что страница загружена, и только после этого наш тест переходит к поиску кнопки. Если страница интерактивная, то браузер будет считать, что страница загружена, при этом продолжат выполняться загруженные браузером скрипты. Скрипт может управлять появлением кнопки на странице и показывать ее, например, с задержкой, чтобы кнопка красиво и медленно возникала на странице. В этом случае наш тест упадет с уже известной нам ошибкой NoSuchElementException, так как в момент выполнения команды button = browser.find\_element(By.ID, "verify") элемент с **id="verify"** еще не отображается на странице. На данной странице пауза перед появлением кнопки установлена на 1 секунду, метод **find\_element()** сделает только одну попытку найти элемент и в случае неудачи уронит наш тест.

## Давайте быстрее это починим: time.sleep()

Теперь, когда мы уже знаем, что кнопка появляется с задержкой, мы можем добавить паузу до начала поиска элемента. Мы уже использовали библиотеку **time** ранее. Давайте применим ее и сейчас:

from selenium import webdriver

from selenium.webdriver.common.by import By

import time

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/wait1.html")

time.sleep(1)

button = browser.find\_element(By.ID, "verify")

button.click()

message = browser.find\_element(By.ID, "verify\_message")

assert "successful" in message.text

Теперь тест проходит. Но что если элемент с сообщением тоже будет появляться с задержкой? Добавить еще один **time.sleep()** перед поиском сообщения? А если изменится время задержки при появлении кнопки? Увеличим длительность паузы? А еще на разных машинах с разной скоростью интернета кнопка может появляться через разные промежутки времени. Можно перед каждым действием добавить задержку, но тогда значительную часть времени прогона тестов будут занимать бесполезные ожидания, при этом с увеличением количества тестов эта проблема будет только расти.

## Есть способы получше: Selenium Waits (Implicit Waits)

Надеемся, вы поняли, что решение с **time.sleep()** плохое: оно не масштабируемое и трудно поддерживаемое.

Идеальное решение могло бы быть таким: нам всё равно надо избежать ложного падения тестов из-за асинхронной работы скриптов или задержек от сервера, поэтому мы будем ждать появление элемента на странице в течение заданного количества времени (например, 5 секунд). Проверять наличие элемента будем каждые 500 мс. Как только элемент будет найден, мы сразу перейдем к следующему шагу в тесте. Таким образом, мы сможем получить нужный элемент в идеальном случае сразу, в худшем случае за 5 секунд.

В Selenium WebDriver есть специальный способ организации такого ожидания, который позволяет задать ожидание при инициализации драйвера, чтобы применить его ко всем тестам. Ожидание называется **неявным** (**Implicit wait**), так как его не надо явно указывать каждый раз, когда мы выполняем поиск элементов, оно автоматически будет применяться при вызове каждой последующей команды.

Улучшим наш тест с помощью неявных ожиданий. Для этого нам нужно будет убрать time.sleep() и добавить одну строчку с методом **implicitly wait**:

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

# говорим WebDriver искать каждый элемент в течение 5 секунд

browser.implicitly\_wait(5)

browser.get("http://suninjuly.github.io/wait1.html")

button = browser.find\_element(By.ID, "verify")

button.click()

message = browser.find\_element(By.ID, "verify\_message")

assert "successful" in message.text

Теперь мы можем быть уверены, что при небольших задержках в работе сайта наши тесты продолжат работать стабильно. На каждый вызов команды **find\_element** WebDriver будет ждать 5 секунд до появления элемента на странице прежде, чем выбросить исключение **NoSuchElementException**.

## Задание: Про Exceptions

Теперь мы знаем, как настроить ожидание поиска элемента. Во время поиска WebDriver каждые 0.5 секунды проверяет, появился ли нужный элемент в DOM-модели браузера (Document Object Model — «объектная модель документа», интерфейс для доступа к HTML-содержимому сайта). Если произойдет ошибка, то WebDriver выбросит одно из следующих исключений (**exceptions**):

* Если элемент не был найден за отведенное время, то мы получим **NoSuchElementException.**
* Если элемент был найден в момент поиска, но при последующем обращении к элементу DOM изменился, то получим **StaleElementReferenceException**. Например, мы нашли элемент **Кнопка** и через какое-то время решили выполнить с ним уже известный нам метод click. Если кнопка за это время была скрыта скриптом, то метод применять уже бесполезно — элемент "устарел" (stale) и мы увидим исключение.
* Если элемент был найден в момент поиска, но сам элемент невидим (например, имеет нулевые размеры), и реальный пользователь не смог бы с ним взаимодействовать, то получим **ElementNotVisibleException**.

Знание причин появления исключений помогает отлаживать тесты и понимать, где находится баг в случае его возникновения.

**Задание:**

Какую ошибку вы увидите в консоли, если попытаетесь выполнить команду **browser.find\_element(By.ID,** **"button")** после открытия страницы <http://suninjuly.github.io/cats.html>?

## Explicit Waits (WebDriverWait и expected\_conditions)

В предыдущем шаге мы решили проблему с ожиданием элементов на странице. Однако методы **find\_element** проверяют только то, что элемент появился на странице. В то же время элемент может иметь дополнительные свойства, которые могут быть важны для наших тестов. Рассмотрим пример с кнопкой, которая отправляет данные:

* Кнопка может быть неактивной, то есть её нельзя кликнуть;
* Кнопка может содержать текст, который меняется в зависимости от действий пользователя. Например, текст "Отправить" после нажатия кнопки поменяется на "Отправлено";
* Кнопка может быть перекрыта каким-то другим элементом или быть невидимой.

Если мы хотим в тесте кликнуть на кнопку, а она в этот момент неактивна, то WebDriver все равно проэмулирует действие нажатия на кнопку, но данные не будут отправлены.

Давайте попробуем запустить следующий тест:

from selenium import webdriver

from selenium.webdriver.common.by import By

browser = webdriver.Chrome()

# говорим WebDriver ждать все элементы в течение 5 секунд

browser.implicitly\_wait(5)

browser.get("http://suninjuly.github.io/wait2.html")

button = browser.find\_element(By.ID, "verify")

button.click()

message = browser.find\_element(By.ID, "verify\_message")

assert "successful" in message.text

Мы видим, что WebDriver смог найти кнопку с**id="verify"** и кликнуть по ней, но тест упал на поиске элемента "**verify\_message**" с итоговым сообщением:

no such element: Unable to locate element: {"method":"id","selector":"verify\_message"}

Это произошло из-за того, что WebDriver быстро нашел кнопку и кликнул по ней, хотя кнопка была еще неактивной. На странице мы специально задали программно паузу в 1 секунду после загрузки сайта перед активированием кнопки, но неактивная кнопка в момент загрузки — обычное дело для реального сайта.

Чтобы тест был надежным, нам нужно не только найти кнопку на странице, но и дождаться, когда кнопка станет кликабельной. Для реализации подобных ожиданий в Selenium WebDriver существует понятие **явных** ожиданий (**Explicit Waits**), которые позволяют задать специальное ожидание для конкретного элемента. Задание явных ожиданий реализуется с помощью инструментов WebDriverWait и **expected\_conditions**. Улучшим наш тест:

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

from selenium import webdriver

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/wait2.html")

# говорим Selenium проверять в течение 5 секунд, пока кнопка не станет кликабельной

button = WebDriverWait(browser, 5).until(

EC.element\_to\_be\_clickable((By.ID, "verify"))

)

button.click()

message = browser.find\_element(By.ID, "verify\_message")

assert "successful" in message.text

**element\_to\_be\_clickable** вернет элемент, когда он станет кликабельным, или вернет **False** в ином случае.

Обратите внимание, что в объекте WebDriverWait используется функция **until**, в которую передается правило ожидания, элемент, а также значение, по которому мы будем искать элемент. В модуле **expected\_conditions** есть много других правил, которые позволяют реализовать необходимые ожидания:

* title\_is
* title\_contains
* presence\_of\_element\_located
* visibility\_of\_element\_located
* visibility\_of
* presence\_of\_all\_elements\_located
* text\_to\_be\_present\_in\_element
* text\_to\_be\_present\_in\_element\_value
* frame\_to\_be\_available\_and\_switch\_to\_it
* invisibility\_of\_element\_located
* element\_to\_be\_clickable
* staleness\_of
* element\_to\_be\_selected
* element\_located\_to\_be\_selected
* element\_selection\_state\_to\_be
* element\_located\_selection\_state\_to\_be
* alert\_is\_present

Описание каждого правила можно найти на [сайте](https://selenium-python.readthedocs.io/api.html#module-selenium.webdriver.support.expected_conditions).

Если мы захотим проверять, что кнопка становится неактивной после отправки данных, то можно задать негативное правило с помощью метода **until\_not**:

# говорим Selenium проверять в течение 5 секунд пока кнопка станет неактивной

button = WebDriverWait(browser, 5).until\_not(

EC.element\_to\_be\_clickable((By.ID, "verify"))

)

## Задание: ждем нужный текст на странице

Попробуем теперь написать программу, которая будет бронировать нам дом для отдыха по строго заданной цене. Более высокая цена нас не устраивает, а по более низкой цене объект успеет забронировать кто-то другой.

В этой задаче вам нужно написать программу, которая будет выполнять следующий сценарий:

1. Открыть страницу <http://suninjuly.github.io/explicit_wait2.html>
2. Дождаться, когда цена дома уменьшится до $100 (ожидание нужно установить не меньше 12 секунд)
3. Нажать на кнопку "Book"
4. Решить уже известную нам математическую задачу (используйте ранее написанный код) и отправить решение

Чтобы определить момент, когда цена аренды уменьшится до $100, используйте метод **text\_to\_be\_present\_in\_element** из библиотеки **expected\_conditions**.

Если все сделано правильно и быстро, то вы увидите окно с числом. Отправьте его в качестве ответа на это задание.

**РЕШЕНИЕ**

from selenium.webdriver.common.by import By

from selenium.webdriver.support.ui import WebDriverWait

from selenium.webdriver.support import expected\_conditions as EC

from selenium import webdriver

import math

import time

import os

def calc(x):

return str(math.log(abs(12\*math.sin(int(x)))))

try:

browser = webdriver.Chrome()

link = "http://suninjuly.github.io/explicit\_wait2.html"

browser.get(link)

# говорим Selenium проверять в течение 12 секунд, значение в выбранном локаторе не станет нужным, **присваивать переменную не нужно!!!**

WebDriverWait(browser, 12).until(

EC.text\_to\_be\_present\_in\_element((By.ID, "price"),"$100")

)

button = browser.find\_element(By.ID, "book")

button.click()

x = int(browser.find\_element(By.ID, "input\_value").text)

y = calc(x)

in1 = browser.find\_element(By.ID, "answer")

in1.send\_keys(y)

button = browser.find\_element(By.ID, "solve")

button.click()

time.sleep(10)

finally:

browser.quit()

**2.6 Полезные ссылки к первому и второму модулям**

В этом уроке мы постарались собрать ссылки на ресурсы, где вы сможете найти дополнительную информацию по использованию Selenium и о тонкостях при работе с ним:

**Общее**

* <http://chromedriver.chromium.org/getting-started>﻿
* [﻿https://www.guru99.com/selenium-tutorial.html](https://www.guru99.com/selenium-tutorial.html) — ﻿Туториал на английском, ориентирован на Java.﻿
* <https://www.guru99.com/live-selenium-project.html> — ﻿Можно попробовать писать автотесты для демо-сайта ﻿банка. Тоже Java.
* <http://barancev.github.io/good-locators/> — что такое хорошие селекторы
* <http://barancev.github.io/what-is-path-env-var/> — что за PATH переменная?

**Ожидания в Selenium WebDriver**

* <https://www.selenium.dev/documentation/webdriver/waits/>﻿﻿
* <https://stackoverflow.com/questions/15122864/selenium-wait-until-document-is-ready>
* <https://blog.codeship.com/get-selenium-to-wait-for-page-load/>
* <http://barancev.github.io/slow-loading-pages/>
* <http://barancev.github.io/page-loading-complete/>

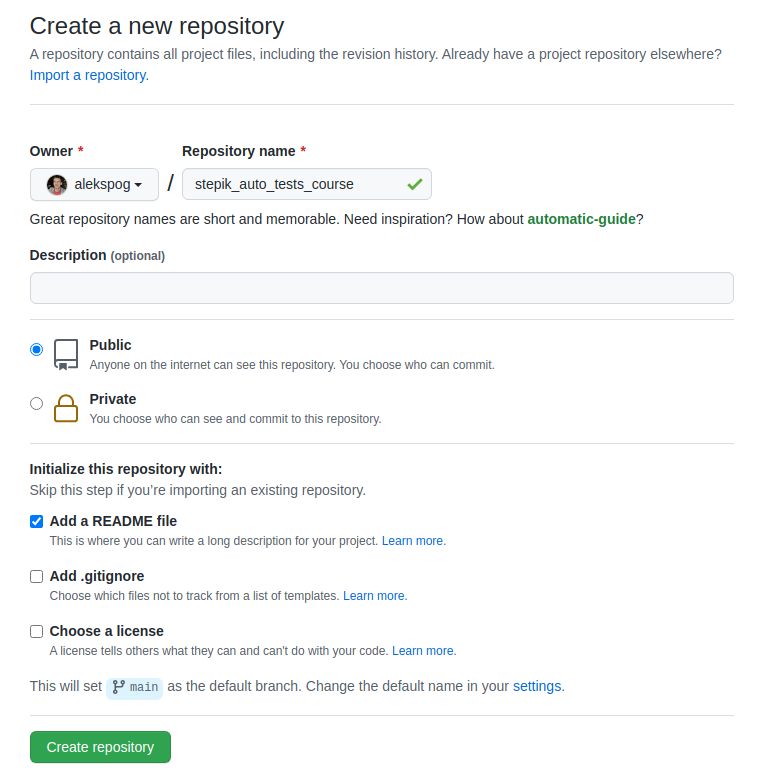
**3.1 Лирическое отступление про Git**

## Первые шаги: создаем аккаунт и свой первый репозиторий

Для начала нужно создать аккаунт на <https://github.com>. Выберите имя пользователя, введите почту и пароль. Лучше сразу подтвердить адрес электронной почты.

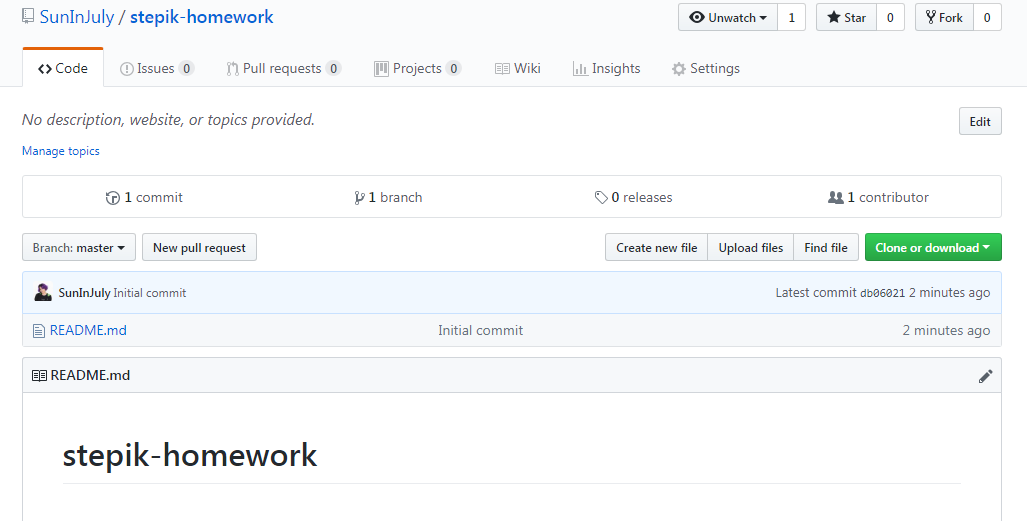
Время создать свой первый репозиторий — это место, где будет храниться ваш проект.

1. В правом верхнем углу рядом с аватаркой нажмите **+** и выберите "New repository".
2. Выберите имя для вашего репозитория. Лучше всего выбирать что-то осмысленное — чтобы через год легко было догадаться, что это и о чем. Например, "stepik\_auto\_tests\_course".
3. Сформулируйте короткое описание для проекта. Здесь можно написать о том, что это домашние задания к курсу.
4. Выбираем Public (публичный) репозиторий: это означает, что любой сможет его найти и открыть.
5. Ставим галочку Initialize this repository with README.
6. Нажимаем кнопку Create repository.



Ура! Теперь у вас есть собственный репозиторий! Пока здесь ничего нет, но мы это скоро исправим.

У вас должно получиться примерно следующее:



## Клонируем репозиторий

Дальше мы будем работать с файлами локально — прямо на своем компьютере. Для этого нам понадобится клиент Git — это программа, которая отслеживает и записывает изменения в файлах.

**Ubuntu:**

**apt-get install git**

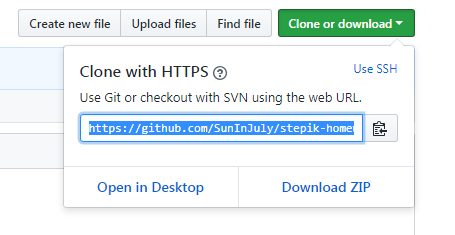
**Windows:**

[**https://gitforwindows.org/**](https://gitforwindows.org/)—  скачать и установить клиент.

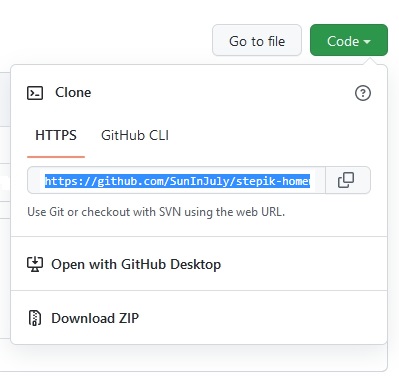
В Ubuntu можно выполнять команды в родном терминале, в Windows нужно запустить установленный **git bash.**

Для начала нужно скопировать к себе только что созданный репозиторий. Для этого применяется термин "склонировать".

На страничке своего проекта нажмите на зеленую кнопку Clone or download и скопируйте адрес из строки:



Некоторое время назад Git немного изменил дизайн. Теперь нужно нажать на зеленую кнопку Code и скопировать адрес из строки:

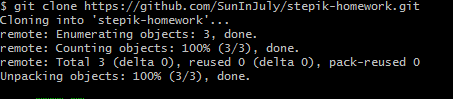


Чтобы склонировать к себе введите в консоли:

**git clone \*адрес вашего репозитория\***

Чтобы перейти в каталог репозитория, введите:

**cd \*имя репозитория \***



В итоге, у вас на компьютере оказался ваш удаленный репозиторий. Давайте сделаем там парочку изменений!

## Мануал и полезные команды

В этом шаге мы рассмотрим несколько полезных команд, которые вам могут пригодиться при работе с Git.

Для начала посмотрим, что Git умеет. Введите в консоль ключевое слово **git**, чтобы увидеть популярные команды:



Если вы хотите посмотреть подсказку по какой-то конкретной команде, то можете ввести **git help <команда>** или **git <команда> --help**. Вы увидите подробную документацию по этой команде из руководства пользователя. Мануал, как и поисковик, — ваш лучший друг, не пренебрегайте его помощью.

Обратите внимание на две команды: **status** и **log**.

Попробуйте запустить команду **git status** и посмотрите на вывод. Данная команда позволяет отслеживать, какие изменения в репозитории вы сделали, что добавили в отслеживаемые файлы, что попало в коммит, что нет и так далее. Словом, это полезная команда, которая позволяет понимать, что происходит.

Попробуйте самостоятельно открыть мануал для команды **git add**. Скопируйте cтроку, которая выводится сразу за словом **NAME**. Введите скопированную строку в качестве ответа на данное задание.

## Первый коммит

Для первого изменения будем использовать уже существующий файл: README.md.

**ФАЙЛ лежит в той же папке, где и создается вирт окр**

**Пользователи и так далее**

Откройте его в текстовом редакторе (например, блокноте). Напишите пару слов о своем проекте: что вы собираетесь в нем хранить. Например, можете оставить там ссылку на этот курс 😉.

Сохраните и закройте файл.

Чтобы Git отслеживал изменения в ваших файлах, нужно их "добавить". Это специальная команда, которая говорит Git, на какие файлы нужно смотреть и записывать их изменения, а все остальные файлы будут игнорироваться.

Для того чтобы добавить файлы под бдительный взор Git, нужно выполнить команду:

**git add README.md**

Таким образом можно добавить любой файл, просто укажите его имя. Обратите внимание, что имена файлов зависят от регистра (case sensitive)!

Теперь гит будет отслеживать все изменения, которые вы будете производить в этом файле. Попробуйте выполнить **git status**и посмотрите, что изменилось в выводе.

Для того чтобы зафиксировать и сохранить свою работу нужно выполнить "коммит". Коммит — это небольшой кусочек вашей работы. Хорошей практикой считается делать коммиты не слишком маленькими (не на каждую новую написанную строку), но и не слишком большими (всю недельную работу сливать одним коммитом — не лучшая идея). Старайтесь добавлять в коммит одну небольшую функцию или класс.

Еще к коммитам пишут короткие сообщения, описывающие изменения — постарайтесь писать их как можно более осмысленными. Просто представьте, что их будут читать ваши коллеги  (или вы сами через год) в попытках разобраться, что вы сделали. Сообщение добавляется с помощью флага **-m**.

Чтобы сделать коммит, нужно ввести команду:

**git commit -m "тут ваше сообщение о коммите"**

(обратите внимание на кавычки!)

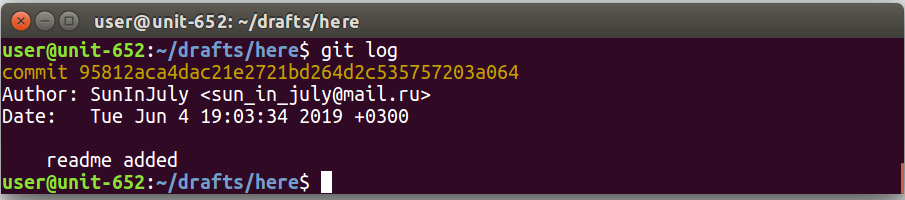
Если все прошло как надо, в ответе вы увидите, сколько строк и файлов изменилось:

https://ucarecdn.com/bb01d09e-d20b-4945-8418-f69855f70212/

Ура, вы сделали первый коммит!

Попробуйте снова выполнить **git status**  и посмотрите на вывод.

Чтобы посмотреть историю коммитов, можно вывести **git log**:



## Добавление изменений на сервер (push)

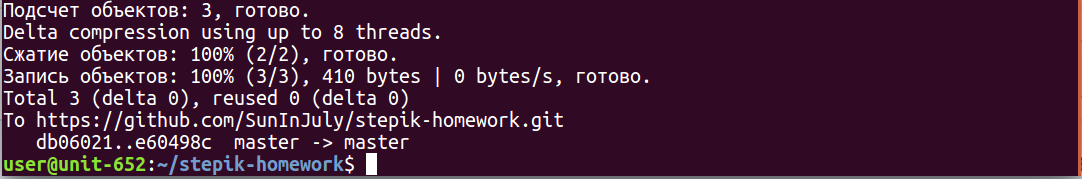
Сейчас у вашего репозитория есть две разные копии — одна локальная, которая уже содержит изменения в файле, и удаленная — на гитхабе. Необходимо наши локальные коммиты положить в удаленный репозиторий. Для этого есть специальная команда **git push <репозиторий><название ветки>**.

Сейчас мы не будем вдаваться подробно в тему ветвления. Достаточно знать, что основная ветка, на которой вы находитесь по умолчанию — это **master**. Мы будем пушить в удаленный репозиторий **origin** — оригинальный репозиторий, откуда мы скопировали к себе на компьютер локальную версию.  
Обратите внимание, что основная ветка в репозитории будет называться **master**, если он был создан до осени 2020 года. Увидеть это можно в Git Bash после перехода в папку проекта.  
https://ucarecdn.com/9a787061-a9b1-4625-b085-3f23ef69c66b/  
В репозитиориях, которые были созданы позднее, основная ветка по умолчанию может называться **main**.  
https://ucarecdn.com/20247aa0-d0a5-455b-acb9-b0fff284f148/

Выполним команду:

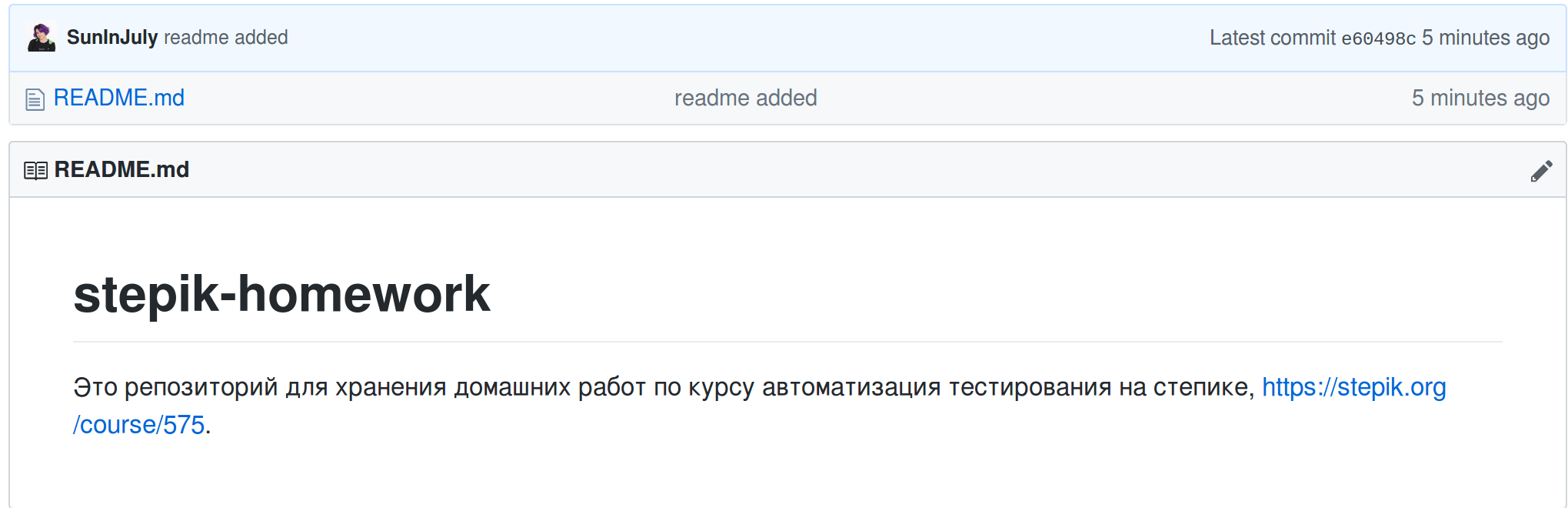
**git push origin master**

Или **git push origin main** для новых репозиториев.  
  
Git попросит ввести ваш логин и пароль на GitHub, и покажет примерно следующее:



Это сообщение о том, сколько данных откуда и куда отправились.

Теперь откройте свой репозиторий на гитхабе в браузере. Если вы все сделали правильно, то гитхаб подтянет описание проекта из файла и красиво отобразит на страничке:



Обратите внимание, что рядом с файлом появилась дополнительная информация: когда файл был последний раз модифицирован и какое было сообщение у этого коммита.

Можно, например, кликнуть на сообщение коммита и посмотреть, какие были изменения:

## Полезные ссылки

<https://learngitbranching.js.org/> — отличный интерактивный туториал

<https://git-scm.com/book/ru/v2/> — лучшая книга вообще

<https://hyperskill.org/learn/topic/257/>﻿

<https://stepik.org/course/4138/>﻿

<https://stepik.org/course/3145/>

<http://www-cs-students.stanford.edu/~blynn/gitmagic/intl/ru/index.html>

<https://habr.com/company/intel/blog/344962/>

<https://githowto.com/ru>

Настоятельно советуем самостоятельно прочитать про ветки (бранчи) и пулл-реквесты — это основной инструмент коллективной работы в Git.

## Тестирование web-приложений и тестовые фреймворки

Далее мы рассмотрим, как использовать Selenium Webdriver для написания автоматических тестов. Почему мы еще не можем назвать тестами скрипты, которые мы писали в предыдущих модулях?

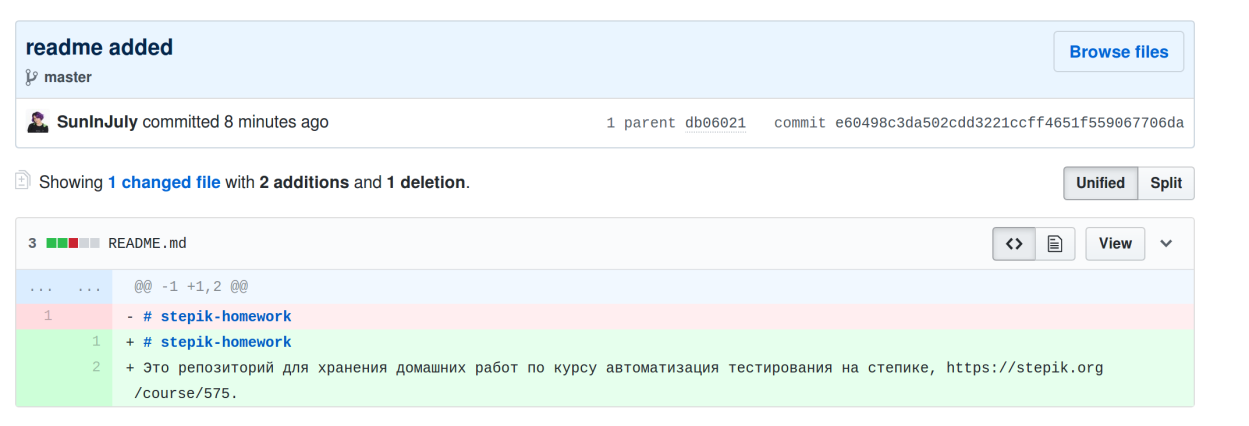
Для этого нам придётся познакомиться с тестовыми фреймворками unittest и PyTest, которые позволяют создавать легко читаемые проверки ожидаемых результатов в тестах, удобно настраивать запуск большого количества тестов в нужных окружениях, организовывать хранение тестов и генерацию отчётов для последующего анализа.

В качестве основы для данного урока мы адаптировали руководство про написание юнит-тестов в Python:

<https://realpython.com/python-testing/>

## Содержание урока

* [Автоматизированное и ручное тестирование](https://stepik.org/lesson/36285/step/2)
* [Юнит-тесты и интеграционные тесты](https://stepik.org/lesson/36285/step/3)
* [Задание: пирамида тестирования](https://stepik.org/lesson/36285/step/4)
* [Структура теста](https://stepik.org/lesson/36285/step/5)
* [Проверка ожидаемого результата](https://stepik.org/lesson/36285/step/6)
* [Составные сообщения об ошибках](https://stepik.org/lesson/36285/step/7)
* [Задание: составные сообщения об ошибках](https://stepik.org/lesson/36285/step/8)
* [Задание: составные сообщения об ошибках и поиск подстроки](https://stepik.org/lesson/36285/step/9)
* [Тестовый сценарий](https://stepik.org/lesson/36285/step/10)
* [Выбор test runner](https://stepik.org/lesson/36285/step/11)
* [unittest](https://stepik.org/lesson/36285/step/12)
* [Задание: оформляем тесты в стиле unittest](https://stepik.org/lesson/36285/step/13)



Зеленые строки — это те, которые добавились, а красные — те, которые были удалены.

Вот и все, вы молодец!

## Автоматизированное и ручное тестирование

Чтобы начать писать хорошие автотесты, нужно ﻿разобраться, в чем плюсы и минусы ручного и автоматизированного тестирования.

| **Ручные тесты** | **Автотесты** |
| --- | --- |
| Шаги могут быть описаны достаточно абстрактно, поэтому тестировщик может увидеть новые проблемы, следуя сценарию теста.  В ручном тесте может содержаться шаг "Зарегистрировать нового пользователя" без пояснений, как это делать, в автотесте должны быть строго описаны данные для теста и сценарий регистрации. | Строгий порядок шагов и их чёткая детализация. Проверяют только то, что было заложено изначально в сценарий теста, поэтому могут быть пропущены новые ошибки в продукте. |
| - Из-за абстрактности описания шагов ручные тесты иногда могут приводить к ложноположительным результатам, когда ожидаемый результат достигнут, но в реальности тестировщик выполнил сценарий неправильно, пропустив баг. | + Запуск автотестов обычно приводит к одним и тем же результатам, т.к. сценарий описан точно. |
| - Требуется много времени для проверки регрессии, что приводит к усталости тестировщика и ошибкам в проверках из-за человеческого фактора (человеку сложно долго выполнять монотонную работу). | + Можно запускать на каждый коммит в тестируемом приложении, что позволяет раньше обнаруживать ошибки в продукте. |

Отдельно стоит описать отрицательные стороны автоматизированного тестирования:

* возможная нестабильность теста, которая не связана с качеством самого теста, а вызывается внешними проблемами: нестабильное сетевое соединение, проблемы с серверами, обновление кода продукта в момент запуска тестов;
* требуется достаточно много времени на разработку и поддержку набора автотестов.

Несмотря на наличие минусов автотестов, в большинстве случаев их использование на проекте помогает быстрее находить ошибки в коде приложения и поддерживать качество продукта на достаточном уровне. Автотесты помогают отделу тестирования оптимизировать свою работу, чтобы сделать счастливее пользователей, которые быстрее получают новые фичи и меньше страдают от ошибок в продукте, так как разработчики раньше узнают о багах и могут заранее принять меры для их устранения, не откладывая релиз продукта. Таким образом, благодаря автотестам налаживается бизнес, который может увеличить скорость внедрения фич, приносящих прибыль.

Ручные тесты в идеальном случае остаются только на этапе проверки новых фич и в виде исследовательского тестирования, которое позволяет найти проблемы в сложных пользовательских сценариях.

## Юнит-тесты и интеграционные тесты

Если вы работаете в тестировании, то уже знаете разницу между юнит-тестами и интеграционными тестами. Юнит-тесты проверяют очень маленький кусок кода, обычно конкретную функцию, и чаще всего их пишут разработчики, которые хорошо понимают возможные крайние случаи для своего стека технологий. Интеграционные тесты проверяют взаимодействие сразу нескольких систем. Они могут создаваться и поддерживаться как разработчиками и тестировщиками, так и аналитиками (если для них разработан удобный фреймворк для написания тестов).

Юнит-тесты всегда автоматизированы, так как проверяют непосредственно работу кода. Интеграционные тесты могут быть ручными и автоматизированными. Иногда выделяют отдельную категорию end-to-end (е2е) тестов, которые проверяют полный стек технологий приложения и пользовательский сценарий взаимодействия с приложением как с черным ящиком. Если говорить про UI-тесты, которые разрабатываются с помощью Selenium, то их стоит отнести к разряду end-to-end тестов, так как они проверяют совместную работу всех систем web-продукта: работу frontend и backend, работу базы данных, дополнительные сервисы, такие как аналитика, платежные системы и так далее.

Подробно про разные типы автотестов мы говорить не будем, но советуем вам изучить теорию самостоятельно. Вот, например, отличная и подробная статья: [Пирамида тестов на практике](https://habr.com/ru/post/358950/).

## Задание: пирамида тестирования

Изучите материалы про пирамиду тестирования.

Отсортируйте типы тестов по их рекомендованному количеству в проекте, начиная с наименьшего количества. Низ пирамиды — это конец списка, внизу экрана.

### Расположите элементы списка в правильном порядке

Верно решили **8 311** учащихся

Из всех попыток **26%** верных

Отлично!

Вы решили сложную задачу, поздравляем! Вы можете помочь остальным учащимся в [комментариях](https://stepik.org/lesson/36285/step/4?unit=162401#discussions), отвечая на их вопросы, или сравнить своё решение с другими на [форуме решений](https://stepik.org/lesson/36285/step/4?thread=solutions&unit=162401#discussions).

**ручные исследовательские тесты**

**автоматизированные end-to-end тесты**

**интеграционные тесты**

**юнит-тесты**

## Структура теста

Для написания UI-тестов можно использовать те же возможности Python, что и для написания юнит-тестов, которые создаются разработчиками.

Любой тест должен содержать:

1. Входные данные.
2. Тестовый сценарий, то есть набор шагов, которые надо выполнить для получения результата.
3. Проверка ожидаемого результата.

Давайте обсудим, как именно можно производить проверки.

## Проверка ожидаемого результата

Как можно проверить ожидаемый результат? Для этого используется встроенная в Python инструкция **assert**, которая проверяет истинность утверждений. **assert True** не приводит к выводу дополнительных сообщений, а вот **assert False** вызовет исключение **AssertionError**.

Рассмотрим работу assert на примере встроенной функции **abs**(), которая возвращает абсолютное значение числа по модулю. Для этого активируйте созданное ранее виртуальное окружение и запустите интерпретатор Python. Например, для Linux выполните:

source selenium\_env/bin/activate

python

Теперь будем вводить приведенные ниже команды и смотреть на результат их выполнения.

Если значение выражения истинно, то в консоли не должно появиться дополнительных сообщений. Выполним:

>>> assert abs(-42) == 42

Если условие не выполнено, то в консоли выводится лог ошибки с названием файла и номером строчки, в которой произошла ошибка, а также тип ошибки **AssertionError**:

>>> assert abs(-42) == -42

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AssertionError

Простое сообщение **AssertionError** не очень информативно. Когда тестов становится много, бывает сложно вспомнить, что именно мы проверяем в данном тесте. Для добавления дополнительного сообщения можно при вызове assert через запятую написать нужное сообщение, которое будет выведено в случае ошибки проверки результата:

>>> assert abs(-42) == -42, "Should be absolute value of a number"

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

AssertionError: Should be absolute value of a number

## Составные сообщения об ошибках

Отдельно хочется поговорить про качество сообщений об ошибках, которые показываются при падении теста. Почему это важно? Хорошо написанный текст помогает быстро локализовать найденный баг и разобраться в том, что произошло и из-за чего тест упал. Хороший assert сэкономит вам часы вашей работы, особенно когда количество тестов переходит за сотню.

В целом, тут как с любым фидбеком: важно давать его точно и актуально. Если вы проверяете наличие элемента, то обязательно пишите, что это за элемент по смыслу на странице:

assert self.is\_element\_present('create\_class\_button', timeout=30), "No create class button"

Примечание: Функция is\_element\_present() вспомогательная. Как её реализовать и использовать, мы разберемся чуть позжe.

Если элемент встречается на нескольких страницах приложения, не лишним будет указать, где именно произошла ошибка:

assert self.is\_element\_present('new\_announcement\_button', timeout=30), "No new announcement button on profile page"

Если вы работаете с каким-то текстом (например, проверяете информационное сообщение, текущий url, ссылку, placeholder в input-элементе или любой другой текст), в сообщении об ошибке всегда лучше выводить оба значения: то, которое ожидалось, и то, которое получили по факту. Всё как в хорошем багрепорте: ожидаемый и фактический результат.

### Форматирование строк с помощью конкатенации

В питоне такое можно провернуть с помощью конкатенации строк, например:

actual\_result = "abrakadabra"

print("Wrong text, got " + actual\_result + ", something wrong")

Но из-за обилия кавычек, знаков сложения и вот этого всего этот способ не самый удобный и читается тоже плохо.

### Форматирование строк с помощью str.format

Гораздо лучше воспользоваться возможностью python для форматирования строк. Дополнительно можно почитать здесь: <https://realpython.com/python-string-formatting/#2-new-style-string-formatting-strformat>

Если вкратце, то python умеет подставлять пользовательские значения в строки с помощью функции **.format()**. Синтаксис выглядит примерно так:

"Let's count together: {}, then goes {}, and then {}".format("one", "two", "three")

Попробуйте запустить её в интерпретаторе:

print("Let's count together: {}, then goes {}, and then {}".format("one", "two", "three"))

Такая строка при исполнении кода превратится в:

Let's count together: one, then goes two, and then three

Таким образом мы можем удобно компоновать ожидаемое и фактическое значение в одну строку.

### Форматирование строк с помощью f-strings

И наконец наиболее современный способ форматирования строк, который появился в Python3.6, носит название f-strings. Он позволяет исполнять выражения на Python прямо внутри строк, обладает еще большей лаконичностью и удобством использования. Для использования возможностей f-strings нужно указывать символ f перед строкой в таком формате: f"ваша строка {my\_var}". В фигурных скобках указывается имя переменной, значение которой надо подставить в строку, или выражение, результат исполнения которого также требуется подставить в вашу строку.

Подробнее про f-strings можно почитать здесь: <https://realpython.com/python-string-formatting/#3-string-interpolation-f-strings-python-36>. Так как мы предполагаем, что вы используете последнюю версию Python, то предлагаем вам применять именно этот подход в данном курсе.

Пример 1:

str1 = "one"

str2 = "two"

str3 = "three"

print(f"Let's count together: {str1}, then goes {str2}, and then {str3}")

Итог выполнения выражений в интерпретаторе:

Let's count together: one, then goes two, and then three

Пример 2:

actual\_result = "abrakadabra"

f"Wrong text, got {actual\_result}, something wrong"

Итог выполнения выражений в интерпретаторе:

Wrong text, got abrakadabra, something wrong

Пример 3:

>>> f"{2+3}"

'5'

Еще один важный момент: когда вы работаете с текстом элементов на странице или любым другим контентом, который может измениться, всегда записывайте его в отдельную переменную для сравнения.

**неправильно:**

assert self.catalog\_link.text == "Каталог", \

f"Wrong language, got {self.catalog\_link.text} instead of 'Каталог'"

Дважды считывать атрибут — это плохая практика, потому что при повторном считывании текст на странице может измениться, и вы получите неактуальный текст об ошибке. Результат выполнения такого теста сложно анализировать:

"Wrong language, got 'Каталог' instead of 'Каталог'"

**правильно:**

catalog\_text = self.catalog\_link.text # считываем текст и записываем его в переменную

assert catalog\_text == "Каталог", \

f"Wrong language, got {catalog\_text} instead of 'Каталог'"

## Задание: составные сообщения об ошибках

Для закрепления материала реализуйте проверку самостоятельно.

Вам дана функция **test\_input\_text**,  которая принимает два значения: **expected\_result** — ожидаемый результат, и **actual\_result** — фактический результат. Обратите внимание, input использовать **не нужно**!

Функция должна проверить совпадение значений с помощью оператора **assert** и, в случае несовпадения, предоставить исчерпывающее сообщение об ошибке.

**Важно!** Формат ошибки должен точно совпадать с приведенным в примере, чтобы его засчитала проверяющая система!

Маленький совет: попробуйте воспользоваться кнопкой "Запустить код" и протестируйте ваш код на разных введенных значениях, проверьте вывод вашей функции на разных парах. Обрабатывать ситуацию с пустым или невалидным вводом не нужно.

**Sample Input 1:**

8 11

**Sample Output 1:**

expected 8, got 11

**Sample Input 2:**

11 11

**Sample Output 2:**

**Sample Input 3:**

11 15

**Sample Output 3:**

expected 11, got 15

def test\_input\_text(expected\_result, actual\_result):

# ваша реализация, напишите assert и сообщение об ошибке

assert actual\_result == expected\_result, f"expected {expected\_result}, got {actual\_result}"

## Задание: составные сообщения об ошибках и поиск подстроки

Иногда при работе с текстами не нужны жёсткие проверки на полное совпадение, и требуется проверить, что некий текст является подстрокой другого текста. Это можно сделать либо с помощью ключевого слова **in**, либо с помощью функции **find**:

s = 'My Name is Julia'

if 'Name' in s:

print('Substring found')

index = s.find('Name')

if index != -1:

print(f'Substring found at index {index}')

Попробуйте запустить этот код в интерпретаторе, чтобы понять разницу в подходах.

Конструкция **'Name' in s** возвращает просто **True** или **False**, a **find()** возвращает индекс первого вхождения подстроки в строку и -1, если подстрока не найдена. Обычно в автотестах достаточно использовать **in**, потому что это более читабельный вариант.

Например, для проверки того, что в текущем url содержится строка login:

assert "login" in browser.current\_url, # сообщение об ошибке

Реализуйте подобную проверку самостоятельно.

Вам дан шаблон для функции **test\_substring**, которая принимает два значения: **full\_string** и **substring**.

Функция должна проверить вхождение строки **substring** в строку **full\_string**с помощью оператора **assert** и, в случае несовпадения, предоставить исчерпывающее сообщение об ошибке.

**Важно!** Формат ошибки должен точно совпадать с приведенным в примере, чтобы его засчитала проверяющая система!

Маленький совет: попробуйте воспользоваться кнопкой "Запустить код" и протестируйте ваш код на разных введенных значениях, проверьте вывод вашей функции на разных парах. Обрабатывать ситуацию с пустым или невалидным вводом не нужно.

**Sample Input 1:**

fulltext some\_value

**Sample Output 1:**

expected 'some\_value' to be substring of 'fulltext'

**Sample Input 2:**

1 1

**Sample Output 2:**

**Sample Input 3:**

some\_text some

**Sample Output 3:**