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.

**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'"