Auditoría, Calidad y Gestión de Sistemas
(ACG)

Práctica 5
Testeando Interfaces de Usuario Web

Con Selenium y Pytest

Curso 22/23

1. Interfaces de Usuario Web (Web UI)

Selenium es un entorno de pruebas de software para aplicaciones basadas en la web. Las pruebas se pueden escribir en un amplio número de lenguajes de programación populares incluyendo Java, C#, Ruby, Groovy, Perl, PHP y Python. Las pruebas pueden ejecutarse entonces usando la mayoría de los navegadores web modernos en diferentes sistemas operativos como Windows, Linux y OSX. En esta practica utilizaremos Chrome en Windows y programaremos las pruebas en python.

Como algunos alumnos pueden no estar familiarizados con los test unitarios en python y Seleium, esta práctica la hemos dividido en dos sesiones:

- Sesión 1: nos familiarizaremos con los conceptos basicos de test unitario en python usando pytest.
- Sesión 2: realizaremos la pruebas con el framework Selenium.

2. Sesión 1

2.1. Instalando el entorno Thonny

En esta asignatura usaremos *Thonny* (https://thonny.org/) un entorno integrado de desarrollo (IDE) pensado para principiantes. Es sencillo de usar, consume pocos recursos y además ya incorpora el interprete de Python versión 3.

La pantalla de Thonny está compuesta de varias ventanas. Esta presentación es configurable pero nosotros usaremos la disposición que se ve en la Figura 1 y que explicamos a continuación.

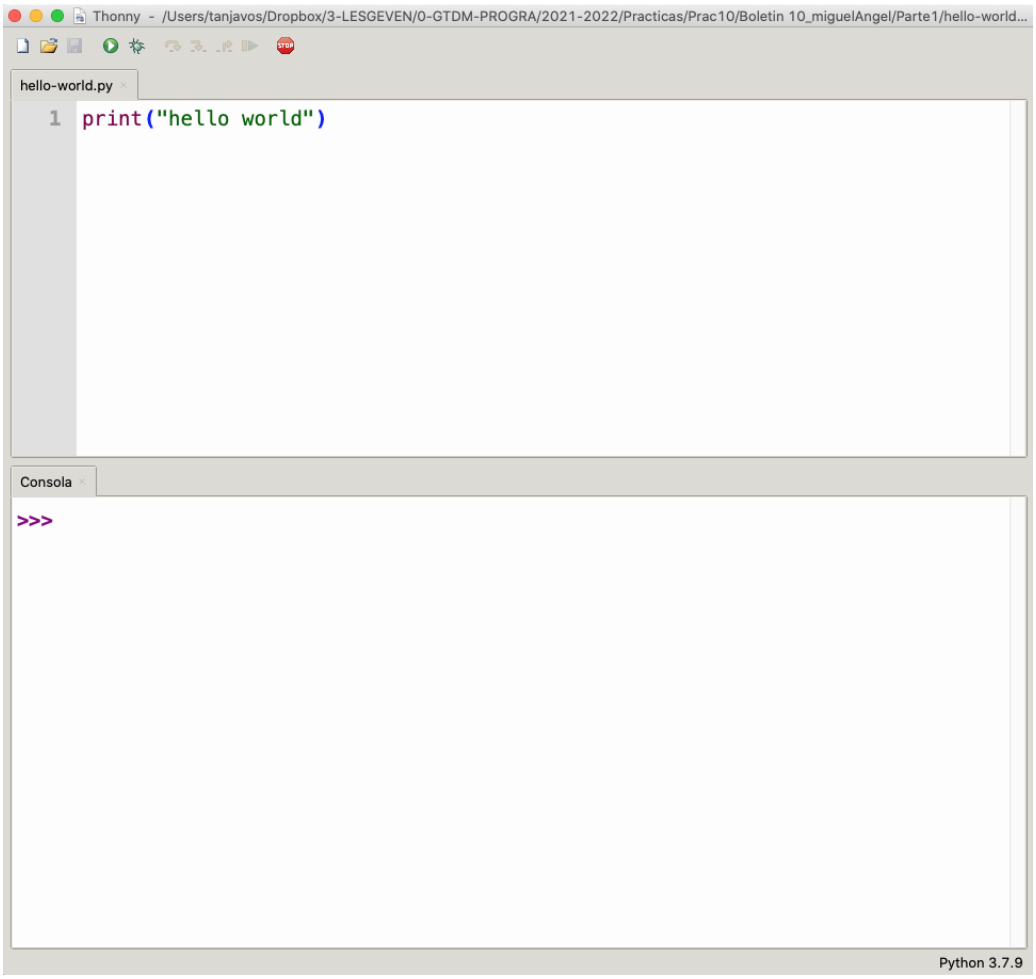

Figura 1: Interfaz de Thonny

En la parte superior encontramos diferentes botones para por ejemplo abrir, ejecutar y parar programas.

Debajo de ella nos encontramos las ventanas principales de trabajo:

- el editor de código donde podremos crear y editar nuestros programas.
- debajo de ella se encuentra el shell o consola de ejecución, mediante ella nos podremos conversar con Python y nuestros programas, ya sea introduciendo datos o viendo los resultados que este presenta. También servirá para ejecutar de forma interactiva ordenes de Python.

  El indicador `>>>` es el modo que tiene el intérprete de Python de indicar que está a la espera de órdenes.

2.2. Instalando Pytest

Pytest es una libreria de test basado en Python, que se utiliza para escribir y ejecutar códigos de prueba.

Para instalar pytest en Thonny debes hacerlo en el menú *Heramientas->Gestión de Paquetes*. Buscamos pytest (Figura 2), seleccionamos pytest (Figura 3) e instalamos.

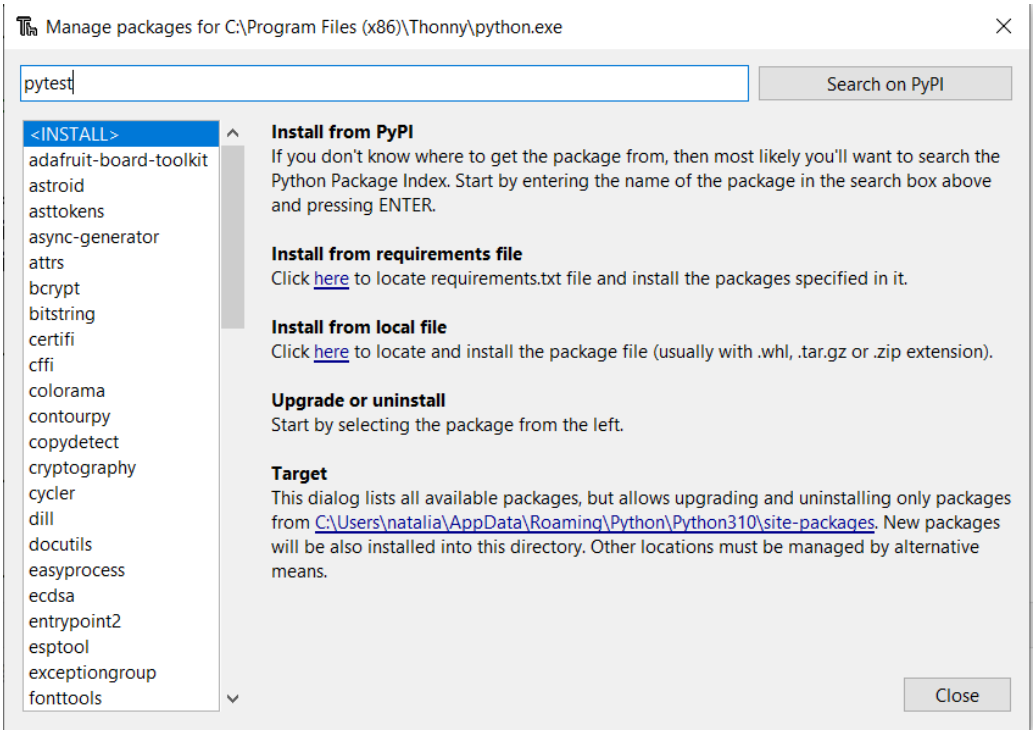

Figura 2: *Heramientas->Gestión de Paquetes*

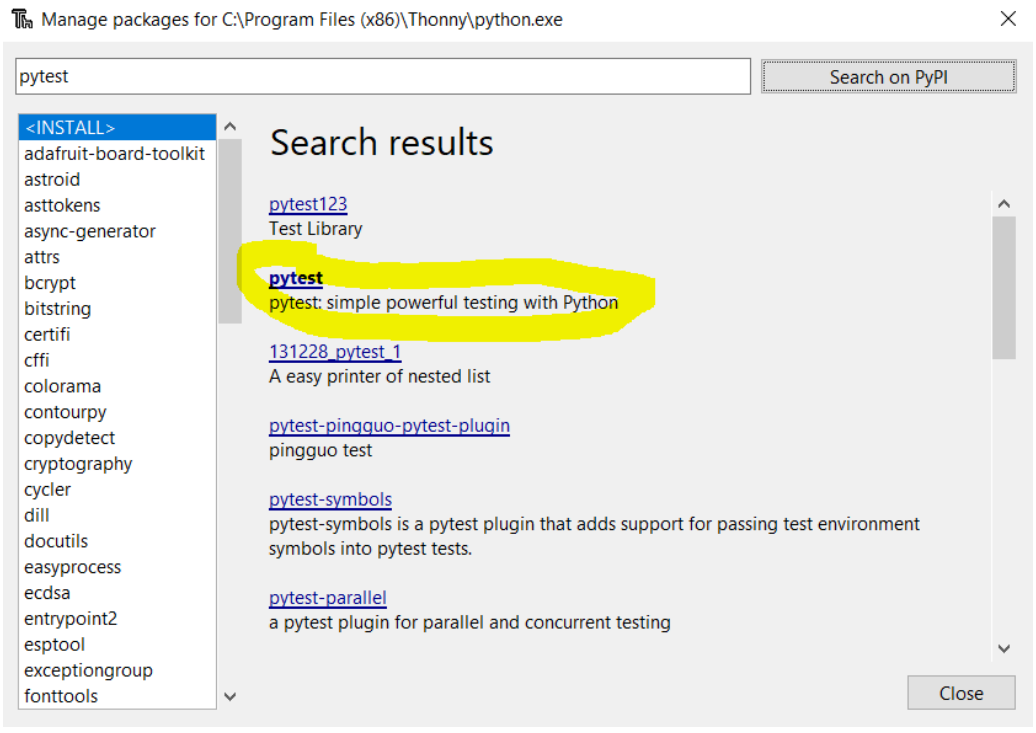

Figura 3: Buscando pytest

2.3. Primer Test

Para probar que todo funciona correctamente, crea un nuevo archivo llamado *test_sample.py*, que contenga una función y una funcion de test:

```
def func(x):
    return x + 1


def test_answer():
    assert func(4) == 5
```

La función básicamente suma uno al parámetro de entrada. El caso de test comprueba si la salida de la función func es 5 cuando se la invoca con el parámetro 4. Para ejecutar el test debes escribir en la consola de Thonny:

ejemlos de test ejecuciones

```
>>> !py.test test_sample.py

============================= test session starts ==============================
platform win32 -- Python 3.10.6, pytest-7.2.0, pluggy-1.0.0
rootdir: C:\Users\natalia\OneDrive - UPV\UPV\2022-23\ACG\NuevasPracticas\seleiumPython
collected 1 item

test_sample.py .                                                         [100%]

============================== 1 passed in 0.01s ===============================
```

El [100 %] se refiere al progreso general de ejecutar todos los casos de prueba. Una vez que finaliza, pytest muestra un informe de de que el test se ha pasado.

2.4. El comando assert

Python implementa una característica llamada aserciones (**assert**) que es bastante útil durante el proceso de depuración y test. Las aserciones permiten probar la corrección del código al verificar si algunas condiciones específicas son ciertas, lo que puede ser útil cuando se testea el código. La condición de aserción siempre debe ser verdadera a menos que tenga un error en el programa o función.

El comando **assert** tiene la la siguiente sintaxis:

```
assert expression[, assertion_message]
```

la palabra clave de **assert**, la expresión o condición para probar y un mensaje opcional. Se supone que la condición siempre es verdadera. Si la condición de aserción es verdadera, entonces no sucede nada y el programa continúa con su ejecución normal. Por otro lado, si la condición es falsa, la afirmación detiene el programa generando un AssertionError. El parámetro assertion_message es opcional pero recomendado.

Como ya sabemos el testing se reduce a comparar un valor observado con uno esperado para verificar si son iguales o no. Este tipo de verificación encaja perfectamente con los **assert**. Los **assert** deben verificar las condiciones que normalmente deberían ser ciertas, a menos que tenga un error en el código.

2.5. El framework pytest

`pytest` es un framework para Python que ofrece la recolección automática de los tests, aserciones simples, soporte para fixtures, debugeo, etc.

Para escribir las pruebas es necesario escribir funciones que comiencen con el prefijo test_. Es necesario que las llamemos así ya que al momento de ejecutar pytest debemos especificar un fichero de python, a partir de este fichero pytest buscará funciones que comiencen con test_.

Por ejemplo, imagina que tenemos que escribir una función `sin_vocales` que dado un string `s`, devuelve el mismo string `s` pero sin vocales. Por ejemplo:

ejemplo de ejecución

```
>>> sin_vocales("el agua esta mojada")
  l g st mjd
```

Ejercicio 1. Implementa la siguiente función en un fichero `sin_vocales.py`

```
def sin_vocales (s):
    """
    devuelve el argumento s sin vocales
    """
    vocales = 'aeiou'
    s_sinVocales = ''
    for ch in s:
        pos = vocales.find(ch)
        if pos == -1: #ch no es vocal
            s_sinVocales = s_sinVocales + ch
    return s_sinVocales
```

Ejercicio 2. Implementa una función de test que al menos pruebe las siguientes cadenas:

- el string del ejemplo `"el agua esta mojada"`,
- el string del ejemplo en mayúsculas,
- otro test donde la cadena empieze con una consonante

Ejercicio 3. Ejecuta los tests. Cuantos casos de test se han llegado a probar? Corrige el programa para tener en cuenta las vocales en mayúsculas y vuevle a ejecutar los tests.

2.5.1. Tests Parametricos

Cuando se prueba una función de test, generalmente se pasan varios conjuntos de parámetros a la función. Por ejemplo, para probar el inicio de sesión en una cuenta, necesitamos simular todo tipo de nombres de usuario y contraseñas. Podemos escribir estos valores dentro de la función de test. Cuando un determinado conjunto de valores hace que la aserción falle, la prueba finaliza. Mediante la captura de excepciones, podemos garantizar la ejecución completa de todos los valores de test, pero resulta complicado analizar los resultados del test.

En pytest, tenemos una mejor solución, que son las pruebas parametrizadas, es decir, cada conjunto de valores se prueba de forma independiente. Por ejemplo, para la función `sin_vocales` podemos probar con:

- el string del ejemplo `"el agua esta mojada"`,
- el ejemplo empieza con un vocal,vamos a hacer otro test que empieza con un consonante
- el ejemplo termina con un vocal,vamos a hacer otro test que termina con un consonante
- el string vacio
- un string sin espacios
- un string con solo una letra vocal,
- un string con solo una letra consonante,
- un string con mayúsculas
- un string que no tiene vocales
- un string que solo tiene vocales
- un string con dígitos
- un string con signos como interrogación

Diseñamos nuestro conjunto de tests eligiendo valores de entradas y salidas esperadas:

caso de test	entrada	salida esperada	comentario
1	`"el agua esta mojada"`	`"l g st mjd"`	ejemplo del enunciado
2	`"mojada bañando en el agua"`	`"mjd bñnd n l g"`	empieza con un consonante
3	`"ahora termina bien"`	`"hr trmn bn"`	termina con un consonante
4	`""`	`""`	el string vacio
5	`"a"`	`""`	solo una letra vocal
6	`"m"`	`"m"`	solo una letra consonante
7	`"unstringsinespacios"`	`"nstrngsnspcs"`	sin espacios
8	`"MAYUSculas FUNCIOnaN"`	`"MYScls FNCnN"`	string con mayúsculas
9	`"krt yhgf dwpq"`	`"krt yhgf dwpq"`	que no tiene vocales
10	`"aeoiuuuoiea"`	`""`	que solo tiene vocales
11	`"disco de los 80"`	`"dsc d ls 80"`	con dígitos
12	`"signos como ? y ! y ¡"`	`"sgns cm ? y ! y ¡"`	con signos como interrogación

Ahora podemos correr los tests de arriba de forma automatizado con pytest, definiendo:

```
import pytest

@pytest.mark.parametrize("num_caso, entrada, salida_esperada",[
(1, "El agua esta mojada", "l g st mjd"),           #ejemplo del enunciado
(2, "unstringsinespacios", "nstrngsnspcs"),         #sin espacios
(3, "", ""),                                        #el string vacio
(4, "MAYUSculas FUNCIOnaN", "MYScls FNCnN"),        #string con mayusculas
(5, "krt yhgf dwpq", "krt yhgf dwpq"),              #que no tiene vocales
(6, "aeoiuuuoiea", ""),                             #que solo tiene vocales
(7, "disco de los 80", "dsc d ls 80"),              #con digitos
(8, "signos como ? y !", "sgns cm ? y !")   #con signos como interrogacion
])

def test_sin_vocales(num_caso, entrada, salida_esperada):
    assert sin_vocales(entrada) == salida_esperada, "caso {0}".format(num_caso)
```

El `pytest.mark.parametrize` permite definir los casos de test (i.e. las entradas y salidas esperadas) que queremos ejecutar. El primer argumento de `pytest.mark.parametrize`, es decir: `'num_caso, entrada, salida_esperada'`, refleja los componentes de los casos de test para esta función. Esto se llama *test signature* y para esta función corresponde con las primeras 3 columnas de nuestra tabla de arriba.

- `'num_caso'`: el identificador/numero del test
- `'entrada'`: la entrada que queremos dar a la función
- `'salida_esperada'`: la salida que esperamos que sale

El resto de los argumentos de `pytest.mark.parametrize` consiste de los casos de test que hemos definidos.

Después definimos una función de prueba `test_sin_vocales` que recibe una cantidad de parámetros que coincide con el *test signature*. El unico que hace la función es:

- llamar a la función que queremos testear con las entradas `sin_vocales(entrada)`
- comparar lo que sale con la salida esperada == `salida_esperada`
- cuando la instrucción **`assert`** recibe un `True` (que significa que lo que ha salido de la función es lo que esperábamos) no hace nada.
- cuando la instrucción **`assert`** recibe un `False` (que significa que NO ha salido lo que esperábamos) entonces lanza un mensaje indicando que caso de test (`num_caso`) ha fallado.

El modulo `pytest` asegura que la función de prueba recibe todos los casos de test definidos en `pytest.mark.parametrize`. Para ejecutar los tests hacemos en la consola del Thonny:

```
>>> !py.test sin_vocales.py
============================= test session starts ==============================
platform darwin -- Python 3.7.9, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /Users/tanjavos/Python/Tema5
collected 8 items

sin_vocales.py ........                                                  [100%]
============================== 8 passed in 0.02s ===============================
```

Y vemos que los 8 casos de test que hemos definidos han pasado. Nuestro programa ha sobrevido los 8 tests! Ahora pensaras quizás "pero en castellano también tenemos acentos, á, é, etc., ta,bien son vocales". Vamos a añadir tests para estos casos: Ahora pensaras quizás "pero en castellano también tenemos acentos, á, é, etc., ta,bien son vocales". Vamos a añadir tests para estos casos:

caso de test	entrada	salida esperada	comentario
9	"ábc élla ó"	"bc ll"	acentos en minúsculas
10	"Ó tambien mayÚsculÁs"	"tmbn myscls"	acentos en mayúsculas

El `pytest.mark.parametrize` se quedará como:

```
@pytest.mark.parametrize("num_caso, entrada, salida_esperada",[
(1, "El agua esta mojada", "l g st mjd"),           #ejemplo del enunciado
(2, "unstringsinespacios", "nstrngsnspcs"),         #sin espacios
(3, "", ""),                                        #el string vacio
(4, "MAYUSculas FUNCIOnaN", "MYScls FNCnN"),        #string con mayúsculas
(5, "krt yhgf dwpq", "krt yhgf dwpq"),              #que no tiene vocales
(6, "aeoiuuuoiea", ""),                             #que solo tiene vocales
(7, "disco de los 80", "dsc d ls 80"),              #con dígitos
(8, "signos como ? y ! y ¡", "sgns cm ? y ! y ¡"),  #con signos como interrogación
(9, "ábc élla ó", "bc ll"),                         #acentos en minúsculas
(10, "Ó tambien mayÚsculÁs", "tmbn myscls")         #acentos en mayúsculas
])
```

Ejecutarmos los tests en la consola del Thonny y nos sale:

```
>>> !py.test sin_vocales.py
============================= test session starts ==============================
platform darwin -- Python 3.7.9, pytest-6.1.2, py-1.9.0, pluggy-0.13.1
rootdir: /Users/tanjavos/Python/Tema5
collected 10 items

sin_vocales.py ........FF                                                [100%]
=================================== FAILURES ===================================
_________________ test_sin_vocales[9-\xe1bc \xe9lla \xf3-bc ll] _________________

num_caso = 9, entrada = 'ábc élla ó', salida_esperada = 'bc ll'

    @pytest.mark.parametrize("num_caso, entrada, salida_esperada",[
    (1, "El agua esta mojada", "l g st mjd"),           #ejemplo del enunciado
    (2, "unstringsinespacios", "nstrngsnspcs"),         #sin espacios
    (3, "", ""),                                        #el string vacio
    (4, "MAYUSculas FUNCIOnaN", "MYScls FNCnN"),        #string con mayúsculas
    (5, "krt yhgf dwpq", "krt yhgf dwpq"),              #que no tiene vocales
    (6, "aeoiuuuoiea", ""),                             #que solo tiene vocales
    (7, "disco de los 80", "dsc d ls 80"),              #con dígitos
    (8, "signos como ? y ! y ¡", "sgns cm ? y ! y ¡"),  #con signos como interrogación
    (9, "ábc élla ó", "bc ll"),                         #acentos en minúsculas
    (10, "Ó tambien mayÚsculÁs", "tmbn myscls")         #acentos en mayúsculas
    ])
```

```
    def test_sin_vocales(num_caso, entrada, salida_esperada):
>       assert sin_vocales(entrada) == salida_esperada, "caso {0}".format(num_caso)
E       AssertionError: caso 9
E       assert 'ábc éll ó' == 'bc ll'
E         - bc ll
E         + ábc éll ó

sin_vocales.py:29: AssertionError
________ test_sin_vocales[10-\xd3 tambien may\xdascul\xc1s-tmbn myscls] ________

num_caso = 10, entrada = 'Ó tambien mayÚsculÁs', salida_esperada = 'tmbn myscls'

    @pytest.mark.parametrize("num_caso, entrada, salida_esperada",[
    (1, "El agua esta mojada", "l g st mjd"),           #ejemplo del enunciado
    (2, "unstringsinespacios", "nstrngsnspcs"),         #sin espacios
    (3, "", ""),                                        #el string vacio
    (4, "MAYUSculas FUNCIOnaN", "MYScls FNCnN"),        #string con mayúsculas
    (5, "krt yhgf dwpq", "krt yhgf dwpq"),              #que no tiene vocales
    (6, "aeoiuuuoiea", ""),                             #que solo tiene vocales
    (7, "disco de los 80", "dsc d ls 80"),              #con dígitos
    (8, "signos como ? y ! y ¡", "sgns cm ? y ! y ¡"),  #con signos como interrogación
    (9, "ábc élla ó", "bc ll"),                         #acentos en minúsculas
    (10, "Ó tambien mayÚsculÁs", "tmbn myscls")         #acentos en mayúsculas
    ])

    def test_sin_vocales(num_caso, entrada, salida_esperada):
>       assert sin_vocales(entrada) == salida_esperada, "caso {0}".format(num_caso)
E       AssertionError: caso 10
E       assert 'Ó tmbn myÚsclÁs' == 'tmbn myscls'
E         - tmbn myscls
E         + Ó tmbn myÚsclÁs
E         ? ++       +   +

sin_vocales.py:29: AssertionError
========================== short test summary info ==========================
FAILED sin_vocales.py::test_sin_vocales[9-\xe1bc \xe9lla \xf3-bc ll] - Assertio...
FAILED sin_vocales.py::test_sin_vocales[10-\xd3 tambien may\xdascul\xc1s-tmbn myscls]
======================== 2 failed, 8 passed in 0.05s ========================
```

Dos casos de test han fallado, el numero 9 y el 10. Nuestra función `sin_vocales` no funciona bien para los acentos en castellano. Podemos ver que el pytest dice:

```
E       AssertionError: caso 9
E       assert 'ábc éll ó' == 'bc ll'
E         - bc ll
E         + ábc éll ó
```

Ha ocurrido un `AssertionError` en caso 9, se esperaba `'bc ll'` pero ha salido `ábc éll ó`.

Ejercicio 4. Arregla la funcion `sin_vocales` y ejecuta los 10 casos de test para ver si ahora pasan todos los tests con tu función.

Ejercicio 5. Define las funciones de test para las siguientes funciones. Como mínimo debes hacer 10 casos de test para cada función.

```python
def is_prime(number):
    """Return True if *number* is prime."""
    for element in range(2, number):  # Don't use 0 and 1
        if number % element == 0:
            return False
    return True
```

```python
def find_next_prime(number):
    """Find the closest prime number larger than *number*."""
    index = number
    while True:
        index += 1
        if is_prime(index):
            return index
```

3. Sesión 2. Selenium con Pytest

Selenium es uno de los frameworks de automatización de pruebas de código abierto más reconocidos. Selenium permite la automatización de pruebas de aplicaciones web o sitios web en diferentes navegadores y sistemas operativos. Selenium ofrece compatibilidad con múltiples lenguajes de programación como Java, JavaScript, Python, C y más, lo que permite a los programadores automatizar las pruebas de su sitio web en cualquier lenguaje de programación con el que se sientan cómodos.

3.1. Configuracion

Para instalar selenium en Thonny debes hacerlo en el menú *Heramientas->Gestión de Paquetes*. Buscamos selenium, seleccionamos selenium (Figura 4) e instalamos.

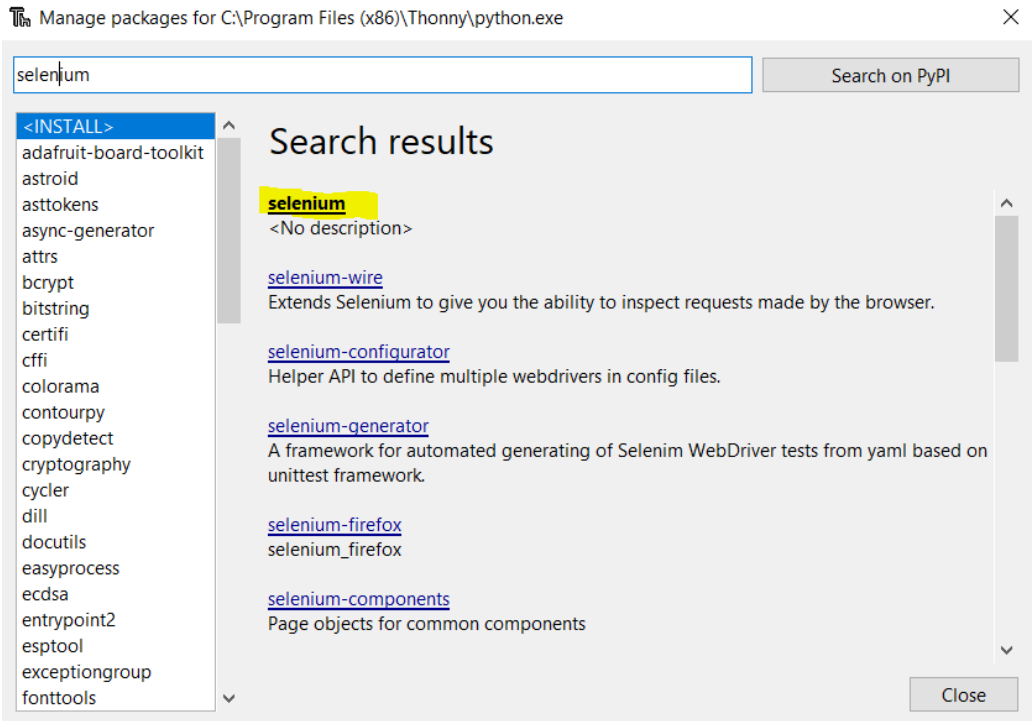

Figura 4: Instalando Selenium

A continuación debemos instalar uno de los Selenium WebDriver. Éste nos permitirá interactuar directamente con los navegadores a través de scripts de prueba escritos en python.Los navegadores compatibles con Selenium WebDriver incluyen Mozilla Firefox, Google Chrome, Internet Explorer, Safari, etc. En esta sesión nosotros vamos a instalar el Chorme Driver. En contreto bajaremos la versión correspondiente de aquí: https://chromedriver.chromium.org/downloads

Se recomienda colocar el controlador del navegador (es decir, en nuestro caso, es ChromeDriver) en la ubicación donde está presente el ejecutable del navegador.

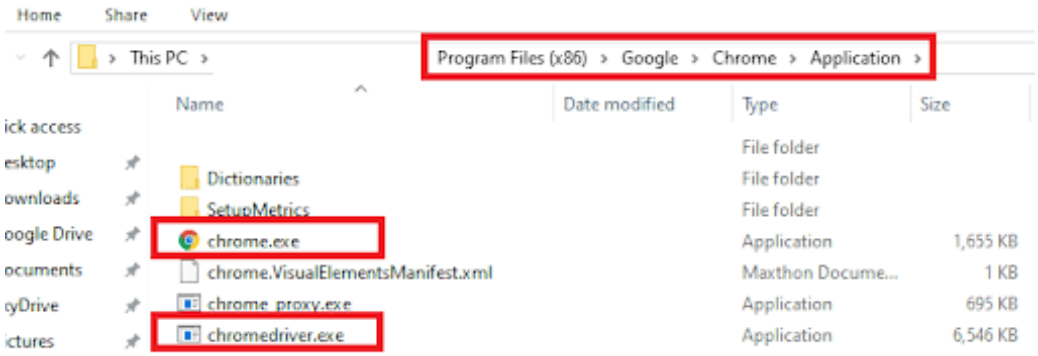

Figura 5: Instalando Selenium

Ejecutamos el driver

3.2. Testeando Una Aplicación Web

El caso de test que vamos a implementar tiene los siguientes pasos:

1. Navegar a la URL https://lambdatest.github.io/sample-todo-app/
2. Seleccionar las dos primeras casillas de verificación.
3. Enviar 'Happy Testing at LambdaTest' al cuadro de texto con id = sampletodotext.
4. Hacer clic en el botón Add y verificar si el texto se ha añadido o no.

Implementaremos el caso de test usando python y pytest.

3.2.1. Implementación

Paso 1: se importan los módulos necesarios:pytest, sys, selenium, time, etc.

```
import pytest
from selenium import webdriver import sys
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.keys import Keys
from time import sleep
```

Paso 2: el nombre de la función de test comienza con test_ (es decir, test_lambdatest_todo_app) para que pytest pueda identificar la prueba. Con el comando Selenium WebDriver, se crea una instancia de Chrome WebDriver. La URL se pasa usando el método .get en Selenium.

```
def test_lambdatest_todo_app():
    chrome_driver = webdriver.Chrome()

    chrome_driver.get('https://lambdatest.github.io/sample-todo-app/')
    chrome_driver.maximize_window()
```

Paso 3: la función de herramienta de inspección del navegador Chrome se utiliza para obtener los detalles de los elementos web requeridos, es decir, las casillas de verificación con el nombre li1 & li2 y el elemento de botón con id = addbutton.

Una vez que los elementos web se encuentran utilizando los métodos de Selenium apropiados (find_element_by_name() y find_element_by_id()), las operaciones necesarias (click()) se realizan en esos elementos.

```
    chrome_driver.find_element_by_name("li1").click()
    chrome_driver.find_element_by_name("li2").click()
```

```
    ......................
    ......................
    chrome_driver.find_element_by_id("addbutton").click()
    sleep(5)

    output_str = chrome_driver.find_element("name","li6").click()
    ......................
```

Paso 4 – El método close() de Selenium se usa para liberar los recursos del driver.

```
chrome_driver.close()
```

Ejercicio 1. Ejecuta el siguiente programa de test con pytest.

```
import pytest
from selenium import webdriver
import sys
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.common.keys import Keys
from time import sleep

def test_lambdatest_app():
    chrome_driver = webdriver.Chrome()

    chrome_driver.get('https://lambdatest.github.io/sample-todo-app/')
    chrome_driver.maximize_window()

    chrome_driver.find_element("name","li1").click()
    chrome_driver.find_element("name","li2").click()

    title = "Sample page - lambdatest.com"
    assert title == chrome_driver.title

    sample_text = "Happy Testing at LambdaTest"
    email_text_field = chrome_driver.find_element("id","sampletodotext")
    email_text_field.send_keys(sample_text)
    sleep(5)

    chrome_driver.find_element("id","addbutton").click()
    sleep(5)

    chrome_driver.find_element("name","li6").click()


    sleep(2)
    chrome_driver.close()
```

La consola debe mostrar que el caso de test ha pasado.

```
>>> !py.test SampleTest.py --verbose
============================= test session starts ==============================
platform win32 -- Python 3.10.6, pytest-7.2.0, pluggy-1.0.0 -- C:\Program Files (x86)\Thonny\python.
cachedir: .pytest_cache
rootdir: C:\Users\natalia\OneDrive - UPV\UPV\2022-23\ACG\NuevasPracticas\seleiumPython
collecting ... collected 1 item
```

```
prueba.py::test_lambdatest_app PASSED                                    [100%]

============================== 1 passed in 17.27s ==============================
```

Aquí está la instantánea de la ejecución de la prueba en curso:

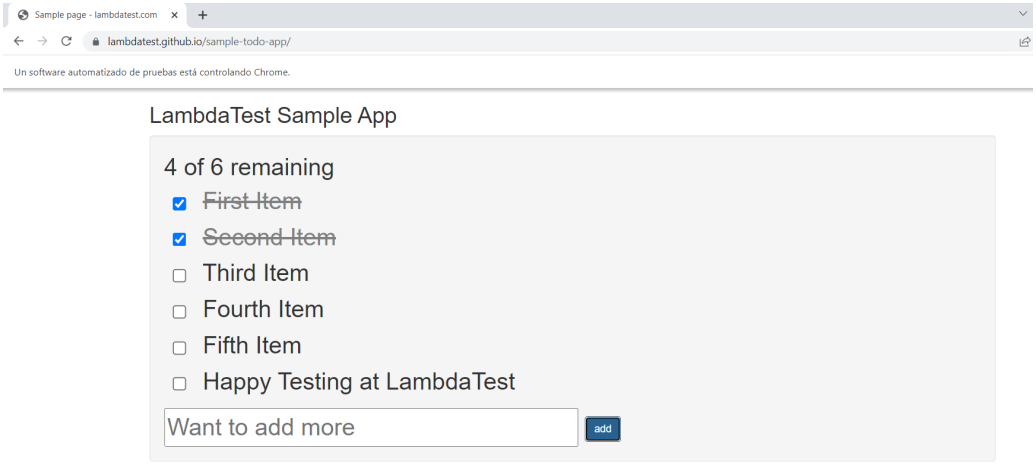

Figura 6: Ejecución de la prueba

Ejercicio 2. Haz un caso de test para la web https://www.phptravels.net. El test debe verificar la funcionalidad de buscar vuelos. Debe comprobar que después de llenar todos los campos, la página web muestra la lista de vuelos disponibles al hacer clic en el botón de buscar.

La documentación del API de Selenium esta aquí: https://www.selenium.dev/selenium/docs/api/py/api.html.

También puede ser útil usar la opción en Chrome de ver el código fuente para encontrar los differentes elementos que necesitas usar en el test (haciendo clic derecho en un espacio en blanco del sitio web y seleccionando la opción “Ver código fuente de la página”).

A continuación te damos una solución incompleta.

```python
from time import sleep
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import datetime
import pytest

def test_FlightSearch_app():
    browser = webdriver.Chrome()
    browser.get("https://www.phptravels.net")
    browser.find_element(By.LINK_TEXT,"Flights").click()
    sleep(2)

    tomorrows_date = str(datetime.date.today() + datetime.timedelta(days=1))
    departure=browser.find_element(By.ID, 'departure')
    departure.send_keys(Keys.CONTROL, 'a')
    departure.send_keys(Keys.BACKSPACE)
    departure.send_keys(tomorrows_date)

    fromInput=browser.find_element(By.ID, 'autocomplete')
    fromInput.send_keys("VLC – Valencia – Valencia")
```

```
    #Falta completar el destino del vuelo


    browser.find_element(By.ID, "flights-search").click()
    assert 'Flights' in browser.title
    browser.close()
```

Ejercicio 3. Haz un test paramétrico basandote el script anterior para probar diferentes opciones de fechas y aeropuertos.