

Testing II

Teórica



Introducción

Bienvenidos al módulo 11 - Testing II. En el mismo, los objetivos de aprendizaje estarán orientados a incorporar conceptos sobre el formato Markdown y el almacenamiento de los resultados de un test en un archivo de texto.

Una vez finalizado este módulo estarás capacitado para las siguientes acciones:

- Conocer los conceptos fundamentales del formato Markdown.
- Crear archivo con formato Markdown en Python.
- Utilizar el formato Markdown para documentar la secuencia de ejecución de un test.
- Generar un archivo de texto para documentar los resultados de la ejecución de un test suite.



Tema 1. Fundamentos de Markdown

Objetivo

Una vez finalizado este tema podrás realizar las siguientes acciones:


- Comprender la estructura general del lenguaje Markdown.
- Utilizar la librería markdown para generar archivos .md utilizando Python.

Introducción













Markdown es un lenguaje de marcado comúnmente utilizado para simplificar el proceso de escribir contenido en un formato de texto, que mediante una herramienta de software se puede convertir en formato HTML para mostrar en un navegador u otro programa de escritura.

Debido a que utiliza sintaxis de texto sin formato, Markdown es compatible con cualquier editor de texto y puede convertir encabezados, listas, enlaces y otros componentes. Los bloggers, los autores de tutoriales y los escritores de documentación usan Markdown ampliamente y los sitios web, como Github , StackOverflow y The Python Package Index (PyPI) , lo admiten.

Por ejemplo, lo podemos encontrar en Jupyter Notebook, seleccionando la opción Markdown a nivel de celda.

 **jupyter** Testing II Last Checkpoint: hace 2 minutos (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

        Run    Markdown 


```
# Título principal
## Subtítulo

* Una viñeta
* Otra viñeta













*Letra cursiva*

**Letra negrita**

1. Item de una Lista
2. Item de una lista
```

 **jupyter** Testing II Last Checkpoint: hace 4 minutos (autosaved)

File Edit View Insert Cell Kernel Widgets Help

        Run    Markdown 

Título principal

Subtítulo

- Una viñeta
- Otra viñeta

Letra cursiva

Letra negrita

1. Item de una Lista
2. Item de una lista

Sintaxis de markdown

En la siguiente [página web](#) podemos consultar las reglas principales para escribir aplicaciones utilizando markdown. También encontraremos buenas prácticas

Sintaxis básica

Todas las aplicaciones que soportan markdown admiten este tipo de elementos.

Element	Markdown Syntax
Heading	# H1 ## H2 ### H3
Bold	**bold text**
Italic	<i>*italicized text*</i>
Blockquote	> blockquote
Ordered List	1. First item 2. Second item 3. Third item
Unordered List	- First item - Second item - Third item
Code	`code`
Horizontal Rule	---
Link	[title](https://www.example.com)
Image	![alt text](image.jpg)

Sintaxis extendida

Estos elementos componen la sintaxis extendida donde agregan nuevas funcionalidades. No todas las aplicaciones de markdown admiten este tipo de elementos.

Element	Markdown Syntax
Table	<pre> Syntax Description ----- ----- Header Title Paragraph Text </pre>
Fenced Code Block	<pre>``` { "firstName": "John", "lastName": "Smith", "age": 25 } ```</pre>
Footnote	<p>Here's a sentence with a footnote. [¹]</p> <p>[¹]: This is the footnote.</p>
Heading ID	<pre>### My Great Heading {#custom-id}</pre>
Definition List	<pre>term : definition</pre>
Strikethrough	<pre>~~The world is flat,~~</pre>
Task List	<pre>- [x] Write the press release - [] Update the website - [] Contact the media</pre>
Emoji (see also Copying and Pasting Emoji)	<p>That is so funny! :joy:</p>
Highlight	<pre>I need to highlight these ==very important words==.</pre>
Subscript	<pre>H~2~O</pre>
Superscript	<pre>x^2^</pre>

Librería Markdown

La librería markdown de Python nos permite trabajar con cadenas de texto y convertirlas a código HTML.

En el siguiente ejemplo vemos cómo convertir un texto a formato HTML, para luego ser convertido a formato archivo.html

```

Librería markdown

In [7]: # Importación
import markdown as mk

In [8]: # Texto a convertir
string = """# Título principal
## Subtítulo

* Una viñeta
* Otra viñeta

*Letra cursiva*

**Letra negrita**

1. Item de una Lista
2. Item de una lista"""

In [10]: # Conversión a html
html = mk.markdown(string)
html

Out[10]: '<h1>Título principal</h1>\n<h2>Subtítulo</h2>\n<ul>\n<li>Una viñeta</li>\n<li>Otra viñeta</li>\n</ul>\n<p>\n<p><strong>Letra negrita</strong></p>\n<ol>\n<li>Item de una Lista</li>\n<li>Item de una lista</li>\n</ol>\n'

In [11]: # Guardamos el contenido de la variable html a un archivo .html
with open("test_markdown.html", "w", encoding="utf-8", errors="xmlcharrefreplace") as output_file:
    output_file.write(html)

```



También podemos escribir scripts para convertir los datos almacenados en variables a un formato HTML.

En el siguiente ejemplo vemos cómo convertir las variables almacenadas en un diccionario.

Convertir un diccionario en un archivo html

```
In [14]: country_cities = {
    'Argentina': ['Córdoba', 'San Luis', 'Gualeguay'],
    'Uruguay': ['Montevideo', 'Salto', 'Paysandú'],
    'Brasil': ['Río de Janeiro', 'Florianópolis', 'San Pablo'],
    'Chile': ['Santiago de Chile', 'Atacama', 'Puerto Montt']
}
```

```
In [15]: with open('cities.md', 'bw+') as f:
    for country, cities in country_cities.items():
        f.write('# {}\n'.format(country).encode('utf-8'))
        for city in cities:
            f.write('* {}\n'.format(city).encode('utf-8'))
    f.seek(0)
    markdown.markdownFromFile(input=f, output='cities.html')
```

←
→
↺
🔍 Archivo | C:/Users/User/Python/Notebooks/cities.html

Argentina

- Córdoba
- San Luis
- Gualeguay

Uruguay

- Montevideo
- Salto
- Paysandú

Brasil

- Río de Janeiro
- Florianópolis
- San Pablo

Chile

- Santiago de Chile
- Atacama
- Puerto Montt



Tema 2. Documentación de la secuencia de un test

Objetivo

Una vez finalizado este tema podrás:

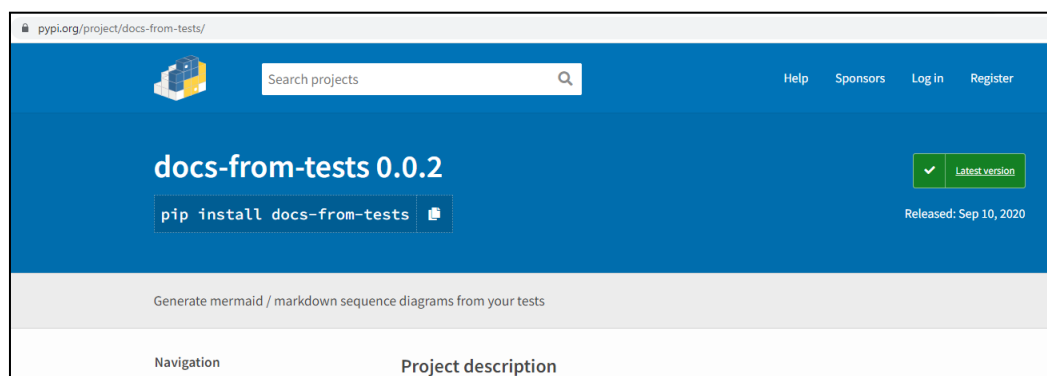
- Generar un archivo markdown con la secuencia lógica de ejecución de nuestro test.

Introducción

La documentación de un proyecto es una de las etapas que más trabajo requiere ya que una vez que nuestro código funciona correctamente, se pasa al siguiente proyecto. Las pruebas que realizamos durante el proyecto no son la excepción y requieren además que la documentación se actualice a la par del código.

Visualización de la secuencia de ejecución

En el siguiente programa vamos a utilizar la librería **docs from tests** (`pip install docs-from-tests`) para generar un diagrama en formato Markdown que nos servirá para visualizar de una manera gráfica la secuencia de ejecución que sigue nuestro código.

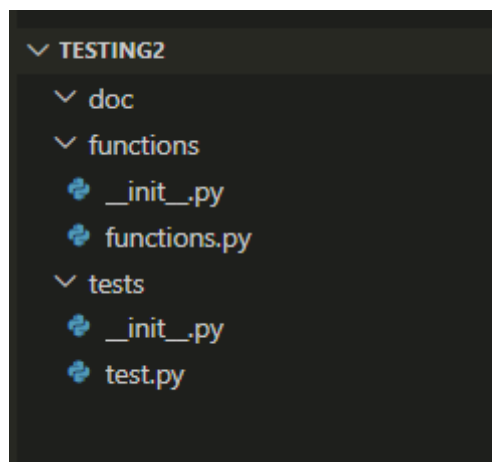


Caso de prueba:

Código fuente: [Link](#)

Se requiere diseñar un test que verifique la cadena “hello world” utilizando Unittest.

Se propone la siguiente estructura de carpetas:



Donde:

- **Doc:** Es la carpeta donde se guardará el diagrama de secuencia una vez ejecutado el test.
- **Functions:** es el módulo con las funciones necesarias para realizar el test.
- **Test:** es el módulo con el test case y la configuración de la librería docs_from_tests

El módulo **functions** contiene tres funciones:

- `hola ()`: que retorna la cadena ‘hola’
- `mundo ()`: que retorna la cadena ‘mundo’
- `get_valid_word ()`: que llama a las dos funciones anteriores y retorna la cadena ‘hola mundo’

```

functions.py X
functions > functions.py > get_valid_word
1  # Funciones.
2
3  def get_hola()-> str:
4      return 'hola'
5
6  def get_mundo ()-> str:
7      return 'mundo'
8
9  def get_valid_word()-> str:
10     valid_word = get_hola() + ' ' + get_mundo()
11     return valid_word
12

```

Para generar un diagrama de la secuencia que realiza nuestro programa, configuramos el módulo test de la siguiente manera:

```

test.py X
tests > test.py > MyTest > test_hello_world
1  import os, sys
2  import unittest
3  sys.path.append('')
4  import functions.functions as f
5
6  # Imports para configurar docs_from_tests
7
8  from pathlib import Path
9  from docs_from_tests.instrument_call_hierarchy import (
10     instrument_and_import_package,
11     initialise_call_hierarchy,
12     finalise_call_hierarchy
13 )
14
15 instrument_and_import_package(os.path.join(Path(__file__).parent.absolute(), '..',
16     'functions'), 'functions')

```

Como podemos ver la librería docs_from_tests requiere su propio setup. Asimismo, en la función *instrument_and_import_package* especificamos nuestro módulo de funciones.

En la clase `TestCase` de `Unittest` vamos a definir la configuración para registrar la secuencia completa de nuestro programa.

```
class MyTest(unittest.TestCase):
    def test_hello_world(self):

        # Inicio de la secuencia
        initialise_call_hierarchy('start')

        # Palabra a testear
        to_check = "hola mundo"

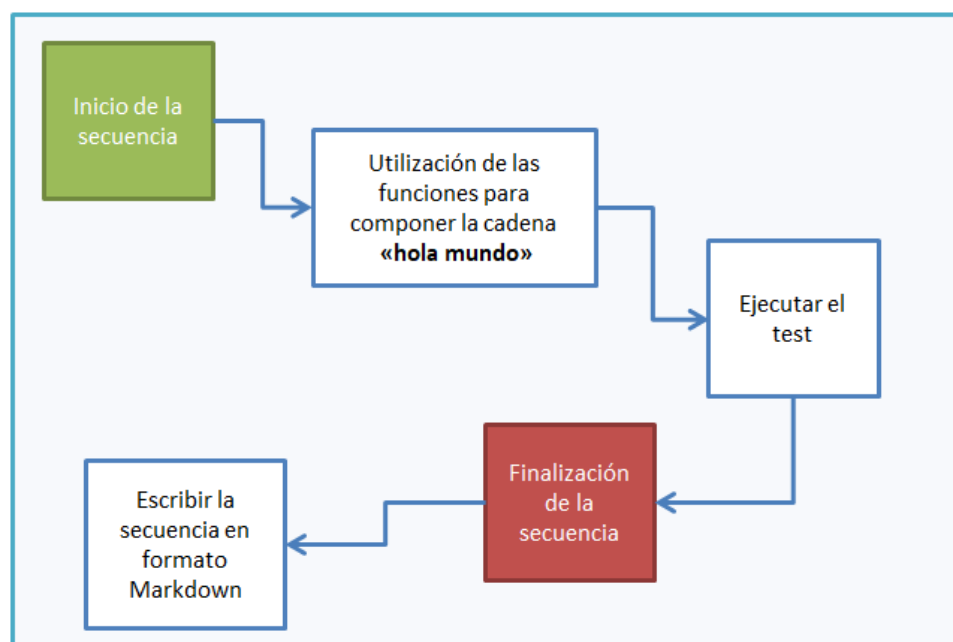
        valid_world = f.get_valid_word()
        self.assertEqual(valid_world, to_check)

        # Finalizamos la secuencia
        root_call = finalise_call_hierarchy()

        # Retornamos el diagrama de secuencia
        sequence_diagram = root_call.sequence_diagram(
            show_private_functions=False,
            excluded_functions=[]
        )

        # Escribimos el archivo de la secuencia en formato markdown
        sequence_diagram_filename = os.path.join(os.path.dirname(__file__), '..',
        'doc', 'diagrama de secuencia.md')
        Path(sequence_diagram_filename).write_text(sequence_diagram)
```

Dentro del testcase se realizan las siguientes acciones:





Tema 3. Documentación de resultados

Objetivo

Una vez finalizado este tema podrás:

- Generar un archivo de texto con el resultado de la ejecución de un test suite.

Introducción

Cuando ejecutamos un test case o un test suit, vemos el resultado en la consola del editor de código. ¿Pero, y si necesitamos almacenar el resultado de las pruebas para consultarlo en otro momento?.

Caso de prueba:

Código fuente: [Link](#)

Se requiere diseñar un test suite con los siguientes test cases.

- Verificar la cadena “Hola”
- Verificar que la palabra “Frutilla” se encuentre dentro de una lista de palabras.
- Guardar el registro de la ejecución de las pruebas en un archivo TXT.

Definimos dos tests cases:

- test_hola: que valida la palabra hola
- test_fruit_is_in: que valida que la palabra “Frutilla” esté presente en una lista de palabras.

```
class MyTests(unittest.TestCase):
    def test_hola(self):
        to_check = 'Hola'
        valid_word = 'Hola'
        self.assertEqual(valid_word, to_check)

    def test_fruit_is_in(self):
        to_check = ['Banana', 'Manzana', 'Frutilla']
        valid_word = 'Frutilla'
        self.assertIn(valid_word, to_check)
```

Definimos dos funciones:

- insert_header: que nos agrega el título, la fecha y la hora en la que se ejecutó el tests.
- main: que se ocupa de generar un test suit con los test cases, los ejecuta y guarda los resultados en el archivo testing.txt.

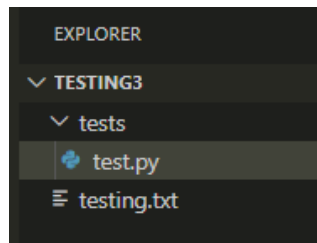
```
def insert_header(f):
    f.write('\n')
    f.write('*****TESTING*****')
    f.write('\n')
    now = datetime.datetime.now()
    date_time = now.strftime("%m/%d/%Y, %H:%M:%S")
    f.write(date_time)
    f.write('\n')
    return f

def main(out = sys.stderr, verbosity = 2):
    loader = unittest.TestLoader()

    suite = loader.loadTestsFromModule(sys.modules[__name__])
    unittest.TextTestRunner(out, verbosity = verbosity).run(suite)

if __name__ == '__main__':
    with open('testing.txt', 'a') as f:
        f = insert_header(f)
        main(f)
```

Finalmente, cuando ejecutamos el script se nos generará el archivo testing.txt.



Por ejemplo: podemos ver el archivo testing.txt, luego de ejecutar los siguientes tests:

- test_fruits_is_in: OK / test_hola: OK
- test_fruits_is_in: FAIL / test_hola: OK
- test_fruits_is_in: OK / test_hola: FAIL

```

testing.txt
1
2 *****TESTING*****
3 10/14/2022, 12:21:29
4 test_fruit_is_in (__main__.MyTests) ... ok
5 test_hola (__main__.MyTests) ... ok
6
7 -----
8 Ran 2 tests in 0.001s
9
10 OK
11
12 *****TESTING*****
13 10/14/2022, 12:21:58
14 test_fruit_is_in (__main__.MyTests) ... FAIL
15 test_hola (__main__.MyTests) ... ok
16
17 =====
18 FAIL: test_fruit_is_in (__main__.MyTests)
19 -----
20 Traceback (most recent call last):
21   File "c:\Users\User\Python\Proyectos\Testing3\tests\test.py", line 17, in test_fruit_is_in
22     self.assertIn(valid_word, to_check)
23   AssertionError: 'Frutilla' not found in ['Banana', 'Manzana', 'Pera']
24
25 -----
26 Ran 2 tests in 0.001s
27
28 FAILED (failures=1)
29
30 *****TESTING*****
31 10/14/2022, 12:22:13
32 test_fruit_is_in (__main__.MyTests) ... ok
33 test_hola (__main__.MyTests) ... FAIL
34
35 =====
36 FAIL: test_hola (__main__.MyTests)
37 -----
38 Traceback (most recent call last):
39   File "c:\Users\User\Python\Proyectos\Testing3\tests\test.py", line 11, in test_hola
40     self.assertEqual(valid_word, to_check)

```




Cierre

Durante esta unidad aprendimos a conectarnos sobre el lenguaje Markdown y también sobre cómo documentar resultados de nuestros tests.

Comenzamos viendo la sintaxis de Markdown y sus componentes principales. Luego aprendimos a utilizar la librería markdown, la cual nos permitió convertir cadenas de texto en formato Markdown, al formato HTML.

Posteriormente trabajamos con la librería docs-from-tests, donde aprendimos cómo generar un diagrama con la secuencia de ejecución de un test.

Finalmente aprendimos a generar un archivo de texto con los resultados de nuestro test, el cual nos permite mantener un registro del mismo.

Quedamos a disposición de las consultas que puedan surgir.



Referencias

<https://www.markdownguide.org/cheat-sheet/>

<https://docs.python.org/3/library/doctest.html>

tiiiiiit by 