Comentarios en Python - Docstrings

(Data Engineer)

tiimiit by

Introducción

Bienvenidos al módulo 7 - Comentarios en Python - Docstrings. En el mismo, los objetivos de aprendizaje estarán orientados a incorporar conocimientos generales acerca de la utilización de Docstrings.

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

- Comprender que es un docstring y los diferentes tipos que existen.
- Conocer buenas prácticas.
- Comprender acerca de las convenciones de docstring más utilizadas.
- Realizar una breve práctica documentando un módulo utilizando Sphinx.

Tema 1. Docstrings

Objetivos

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

- Explicar que es un docstring y describir los diferentes tipos de docstring que existen.
- Comprender qué elementos deben tener docstrings.
- Utilizar mejores prácticas para implementarlos.

Introducción

La Python Software Foundation (PSF) nos provee definiciones del lenguaje, funcionalidades y cómo deberíamos usarlo. Todo se encuentra abierto al público en forma de Propuestas de Mejora de Python, en inglés, Python Enhancement Proposal, y conocidas como PEP.

PEP-257 Convenciones de docstrings

Este PEP documenta la semántica y convenciones asociadas a los docstrings de Python.

El objetivo de este PEP es estandarizar la estructura de alto nivel de los docstrings: qué deben contener y cómo decirlo (sin tocar ninguna sintaxis de marcado dentro de los docstrings). Este PEP contiene convenciones, no leyes ni sintaxis.

Esto lo resumió muy bien el desarrollador Tim Peters:

> *"Una convención universal proporciona toda la capacidad de mantenimiento, claridad, consistencia y también una base para buenos hábitos de programación. Lo que no hace es insistir en que lo sigas en contra de tu voluntad. ¡Eso es Python!"*

—Tim Peters en comp.lang.python, 2001-06-16

Si violamos estas convenciones, lo peor que obtendremos serán algunas miradas raras. Pero algunos programas (como el sistema de procesamiento de cadenas de documentos Docutils PEP 256 , PEP 258) reconocerán las convenciones, por lo que seguirlas nos permitirá obtener los mejores resultados.

La PEP-257 nos sugiere:

- Qué componentes deben tener DocString.
- Cómo implementar DocString de una línea.
- Cómo implementar DocString de múltiples líneas.
- Indentación de DocStrings.

¿Qué es un docstring?

Un docstring es una cadena literal de una o varias líneas, delimitada por comillas triples, simples o dobles que se suele utilizar al comienzo de un módulo, clase, método o función y que describe su funcionamiento.

Si y sólo si es la primera declaración en la función, puede ser reconocida por el compilador de código de bytes de Python y accesible como atributo de objeto de tiempo de ejecución a través del método __doc__ o la función help().

Por ejemplo, en la siguiente función utilizamos el docstring como la primera declaración de la función.

```
def show_docstring():
    """Descripción de la función de impresión"""
    pass
```

Utilizando el método __doc__ podemos visualizar el docstring.

```
show_docstring.__doc__
'Descripción de la función de impresión'
```

Utilizando la función help () también podemos visualizar el docstring.

```
help(show_docstring)
```

```
Help on function show_docstring in module __main__:

show_docstring()
    Descripción de la función de impresión
```

También podemos consultar los docstrings de las funciones o módulos que utilizamos habitualmente. Por ejemplo en el módulo logging, podemos consultar el docstring de la función getLevelName().

```
import logging
```

```
print(logging.getLevelName.__doc__)
```

```
    Return the textual or numeric representation of logging level 'level'.

    If the level is one of the predefined levels (CRITICAL, ERROR, WARNING,
    INFO, DEBUG) then you get the corresponding string. If you have
    associated levels with names using addLevelName then the name you have
    associated with 'level' is returned.

    If a numeric value corresponding to one of the defined levels is passed
    in, the corresponding string representation is returned.

    If a string representation of the level is passed in, the corresponding
    numeric value is returned.

    If no matching numeric or string value is passed in, the string
    'Level %s' % level is returned.
```

```
help(logging.getLevelName)
```

```
Help on function getLevelName in module logging:

getLevelName(level)
    Return the textual or numeric representation of logging level 'level'.

    If the level is one of the predefined levels (CRITICAL, ERROR, WARNING,
    INFO, DEBUG) then you get the corresponding string. If you have
    associated levels with names using addLevelName then the name you have
    associated with 'level' is returned.

    If a numeric value corresponding to one of the defined levels is passed
    in, the corresponding string representation is returned.

    If a string representation of the level is passed in, the corresponding
    numeric value is returned.

    If no matching numeric or string value is passed in, the string
    'Level %s' % level is returned.
```

Si accedemos al código del módulo logging y nos dirigimos a la función getLevelName(level) vamos a encontrar su docstring.

```python
def getLevelName(level):
    """
    Return the textual or numeric representation of logging level 'level'.

    If the level is one of the predefined levels (CRITICAL, ERROR, WARNING,
    INFO, DEBUG) then you get the corresponding string. If you have
    associated levels with names using addLevelName then the name you have
    associated with 'level' is returned.

    If a numeric value corresponding to one of the defined levels is passed
    in, the corresponding string representation is returned.

    If a string representation of the level is passed in, the corresponding
    numeric value is returned.

    If no matching numeric or string value is passed in, the string
    'Level %s' % level is returned.
    """
    # See Issues #22386, #27937 and #29220 for why it's this way
    result = _levelToName.get(level)
    if result is not None:
        return result
    result = _nameToLevel.get(level)
    if result is not None:
        return result
    return "Level %s" % level
```

Docstrings de una sola línea

Son aquellos que se utilizan para casos realmente obvios y por lo tanto caben en una sola línea.

```
def kos_root():
    """Return the pathname of the KOS root directory."""
    global _kos_root
    if _kos_root: return _kos_root
    ...
```

Docstrings multilínea

Son aquellos que constan de una línea de resumen, como un docstring de una sola línea, seguida por una línea en blanco y de una descripción más elaborada.

La línea de resumen puede ser utilizada por herramientas de indexación automática; es importante que quepa en una línea y que esté separado del resto del docstring por una línea en blanco. La línea de resumen puede estar en la misma línea que las comillas de apertura o en la línea siguiente. Todo el docstring tiene la misma sangría que las comillas en su primera línea.

```
def complex(real=0.0, imag=0.0):
    """Form a complex number.

    Keyword arguments:
    real -- the real part (default 0.0)
    imag -- the imaginary part (default 0.0)
    """
    if imag == 0.0 and real == 0.0:
        return complex_zero
    ...
```

Mejores prácticas

A continuación revisaremos algunas de las recomendaciones que se indican en la PEP 257.

Recomendaciones a seguir en Clases

- Insertar una línea en blanco después de todos los docstring (de una línea o de varias líneas) que documentan una clase. En términos generales, los métodos de la clase están separados entre sí por una sola línea en blanco, y la cadena de documentos debe compensarse del primer método por una línea en blanco.
- El docstring de una clase debe resumir su comportamiento y enumerar los métodos públicos y las variables de instancia.
- Si la clase está destinada a ser subclase y tiene una interfaz adicional para subclases, debe enumerarse por separado (en el docstring).
- El constructor de la clase debe estar documentado en la cadena de documentación para su __init__ método. Los métodos individuales deben estar documentados por su propia cadena de documentación.
- Si una clase subclasifica a otra clase y su comportamiento se hereda principalmente de esa clase, su cadena de documentación debe mencionar esto y **resumir las diferencias**. Podemos utilizar el verbo "override" para indicar que un método de subclase reemplaza un método de superclase y no llama al método de superclase. También podemos utilizar el verbo "extend" para indicar que un método de subclase llama al método de superclase (además de su propio comportamiento).

  Ejemplo: Docstring utilizado en una clase, constructor y método.

```
class PlaceHolder(object):
    """
    PlaceHolder instances are used in the Manager logger hierarchy to take
    the place of nodes for which no loggers have been defined. This class is
    intended for internal use only and not as part of the public API.
    """
    def __init__(self, alogger):
        """
        Initialize with the specified logger being a child of this placeholder.
        """
        self.loggerMap = { alogger : None }

    def append(self, alogger):
        """
        Add the specified logger as a child of this placeholder.
        """
        if alogger not in self.loggerMap:
            self.loggerMap[alogger] = None
```

Recomendaciones a seguir scripts stand alone

El docstring de un script stand alone (independiente) debería poder usarse como su mensaje de "uso", impreso cuando se invoca el script con argumentos incorrectos o faltantes (o quizás con una opción "-h", para "ayuda").

Dicho docstring debe documentar la función del script, la sintaxis de la línea de comandos, las variables de entorno y los archivos. Los mensajes de uso pueden ser bastante elaborados (varias pantallas llenas) y deberían ser suficientes para que un usuario nuevo use el comando correctamente, así como una referencia rápida completa a todas las opciones y argumentos para el usuario sofisticado.

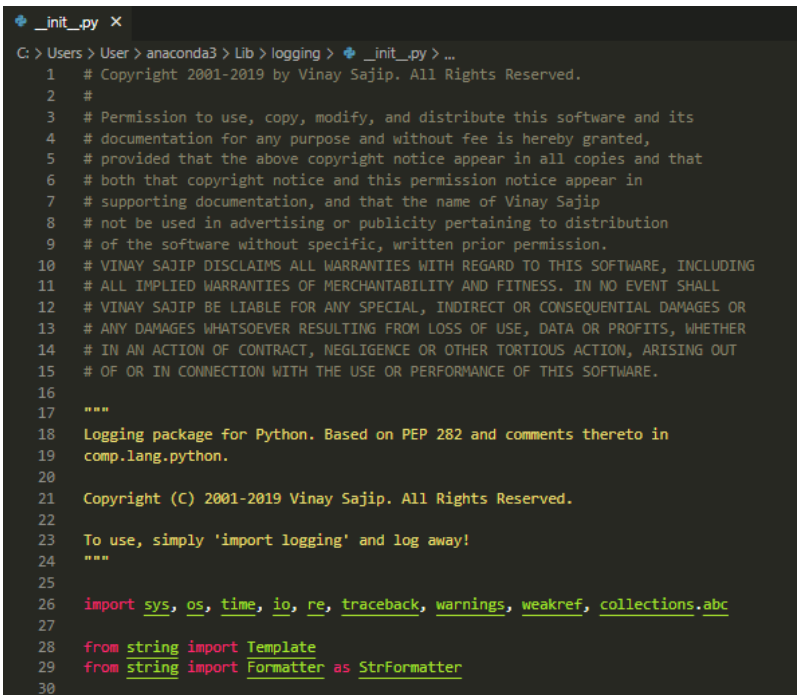
Recomendaciones a seguir en módulos

El docstring de un módulo generalmente debe enumerar las clases, excepciones y funciones (y cualquier otro objeto) que exporta el módulo, con un resumen de una línea de cada una. (Estos resúmenes generalmente brindan menos detalles que la línea de resumen en la cadena de documentación del objeto).

Recomendaciones a seguir en paquetes

El docstring de un paquete (es decir, la cadena de documentación del **__init__.py** módulo del paquete) también debe enumerar los módulos y subpaquetes exportados por el paquete.

Ejemplo: Docstring utilizado en el constructor del módulo logging.



Recomendaciones a seguir en funciones

La cadena de documentación para una función o método debe resumir su comportamiento y documentar sus argumentos, valores devueltos, efectos secundarios, excepciones planteadas y restricciones sobre cuándo se puede llamar (todo si corresponde). Deben indicarse los argumentos opcionales. Debe documentarse si los argumentos de palabras clave son parte de la interfaz.

Manejo de la sangría

Las herramientas de procesamiento de docstring eliminarán una cantidad uniforme de sangría de la segunda línea y posteriores, igual que la sangría mínima de todas las líneas que no estén en blanco después de la primera línea.

Cualquier sangría en la primera línea de la cadena de documentación (es decir, hasta la primera línea nueva) es insignificante y se elimina. Se conserva la sangría relativa de las líneas posteriores en la cadena de documentación. Las líneas en blanco deben eliminarse al principio y al final de la cadena de documentación.

Tema 2. Convenciones

Objetivos

Una vez finalizado este tema estarás capacitado para comprender las tres principales convenciones de docstring.

Introducción

La Python Software Foundation (PSF) identificó la necesidad de proveer más lineamientos, por lo que redactó un PEP que extiende al PEP-257: PEP-287, basado en el proyecto reStructuredText.

Este proyecto comprende un lenguaje de marcas ligero creado para escribir textos con formato definido de manera cómoda y rápida. Es parte del proyecto Docutils dentro de la comunidad de Python, y es formalizado por el grupo Python Doc-SIG (Documentation Special Interest Group). En el siguiente link podemos encontrar más información

Convenciones para docstrings

Existen 3 implementaciones del estándar y que son ampliamente adoptados por la industria, siendo utilizados por grandes empresas y librerías Open Source. Los IDEs más populares utilizan estas implementaciones para auto-generar los DocStrings.

Estilo Sphinx

El estilo Sphinx es el que utiliza la PSF para escribir sus documentos. También se utiliza de forma predeterminada en PyCharm de JetBrains (el IDE inmensamente popular), al escribir comillas triples después de definir su función y presionar “Enter”.

Sphinx es un generador de documentos HTML, PDF, y otros, que utiliza los docstrings del proyecto como fuente de información.

El estilo Sphinx utiliza la sintaxis de un lenguaje de markup ligero reStructuredText (reST) , diseñado para ser:

- Procesable por software de procesamiento de documentación como Docutils.
- Fácilmente legible por programadores humanos que leen y escriben código fuente de Python.

Este estilo utiliza diferentes palabras reservadas para comentar el código. Las más importantes son las siguientes:

- **:param** : valor del parámetro.
- **:type** : tipo de variable.
- **:return** : valor devuelto.
- **:rtype** : tipo de valor devuelto.
- **:raises** : describe los errores que genera el código.
- **..seealso::** : otras lecturas de interés.
- **..notes::** : agrega una nota
- **..warning::** agrega una advertencia

Aunque el orden de estas palabras clave no es fijo, es (nuevamente) una convención mantener el orden anterior a lo largo de todo el proyecto. Las entradas **seealso**, **notes** y **warning** son opcionales.

```python
class World:
    """Class to contain the properties of the World.

    :param yearly_population: World's population by year
    :type arg: dict
    :ivar yearly_population: Internal attribute to hold the world population.
    :vartype yearly_population: dict
    """

    def __init__(self, yearly_population):
        self.yearly_population = yearly_population

    def population(self, year):
        """Obtain the population in the world for a given year.
        :param year: Year of which the population wants to be known.
        :type year: int
        :returns: Population for that year. 0 if the year was not found.
        :rtype: int
        """
        return self.yearly_population.get(year, 0)
```

DocString con formato Google

El famoso buscador desarrolló su propio estándar Python, extendiendo los PEP e implementó una guía de estilos para la documentación.

La guía de estilos de Google es ampliamente adoptada en la comunidad Python. Es común encontrarla en proyectos privados y en cientos de librerías Open Source.

```python
class World:
    """Class to contain the properties of the World.

    Args:
        yearly_population (dict): World's population by year
    Attributes:
        yearly_population (dict): Internal attribute to hold the world population.
    """

    def __init__(self, yearly_population):
        self.yearly_population = yearly_population

    def population(self, year):
        """Obtain the population in the world for a given year.
        Args:
            year (int): Year of which the population wants to be known.
        Returns:
            Population for that year. 0 if the year was not found.
        """
        return self.yearly_population.get(year, 0)
```

DocString con formato Numpy

La librería de computación numérica Numpy, conocida por la comunidad científica y de datos, también desarrolló su estándar. Al generar documentación científica y con un nicho más específico, el estándar de Numpy propone verbosidad y detalle en sus DocStrings.

```python
class World:
    """Class to contain the properties of the World.

    Parameters
    ----------
    yearly_population : dict
        World's population by year
    Attributes
    ----------
    yearly_population :dict
        Internal attribute to hold the world population.
    """

    def __init__(self, yearly_population):
        self.yearly_population = yearly_population

    def population(self, year):
        """Obtain the population in the world for a given year.
        Parameters
        ----------
        year : int
            Year of which the population wants to be known.
        Returns
        -------
        Population for that year. 0 if the year was not found.
        """
        return self.yearly_population.get(year, 0)
```

Tema 3. Implementación con Sphinx

Objetivos

Realizar una breve introducción práctica a Sphinx que nos permita visualizar cómo generar documentación en formato html de nuestro código Python, a partir de comentarios realizados con docstring a nivel de módulo.

Introducción

Cuando trabajamos en un proyecto que debe completarse en un tiempo determinado, además de revisiones de código, pruebas de automatización, pruebas unitarias y muchas cosas más, rara vez nos queda tiempo para la documentación. Y no importa lo que estemos desarrollando, tarde o temprano nosotros mismos o nuestros compañeros visitarán esa pieza de código nuevamente. Y cuando llegue el día, ¡la mayoría de nosotros nos perderemos en esos dentro del código!

La documentación se deja de lado debido al tiempo que consume, pero ¿y si todo esto se puede automatizar y en un abrir y cerrar de ojos puede generar un hermoso sitio web que documente todo su código? ¡Aquí es donde entra Sphinx!

¿Cómo funciona?

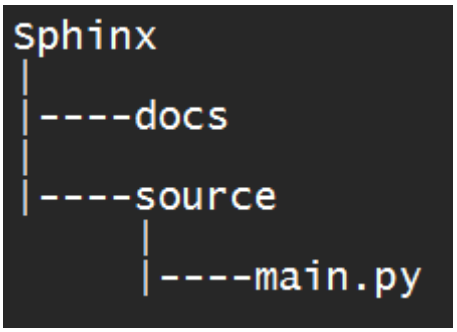
En términos más simples, Sphinx toma nuestros archivos .rst y los convierte a HTML, ¡y todo eso se hace usando solo un montón de comandos! Las principales bibliotecas de Python como Django, NumPy , SciPy , Scikit-Learn , Matplotlib y muchas más están escritas con Sphinx.

Proyecto de implementación

A continuación vamos a utilizar Sphinx para realizar un archivo html con la documentación de una clase.

Pasos a seguir

1. Crear la siguiente estructura de carpetas. Adentro de la carpeta source crear el archivo main.py con el siguiente código.




```
#main.py
```

```
"""
Esta es una clase de pruebas para implementar Sphinx


"""
class Teacher:
    """
    Clase que contiene las propiedades de un profesor


    :param first_name: Primer nombre del profesor
```

```
    :param last_name: Apellido del profesor
    :param age: Edad del profesor

    """
    def __init__(self, first_name, last_name, age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

    def get_full_name(self):
        """
        Esta función no recibe parámetros

        :return: Nombre y apellido del profesor
        :rtype: string
        """
        return f"{self.first_name} {self.last_name}"

    def introduce(self):
        """
        Esta función no recibe parámetros
```

```
            :return: Presentación del profesor. Nombre,
apellido y edad

        :rtype: string

        """

                    return f"Hi. I'm {self.first_name}
{self.last_name}. I'm {self.age} years old."
```

2. Instalar Sphinx a través de la línea de comandos o la terminal del editor de código. `pip install -U sphinx`
3. Iniciar Sphinx a través de la línea de comandos o la terminal del editor de código. `sphinx-quickstart`
4. Completamos las siguientes preguntas
    a. Separar directorios fuente y compilado (y/n) [n]: n
    b. Nombre de proyecto: Docsting - Sphinx
    c. Autor(es): Alkemy
    d. Liberación del proyecto []: 1.0.0
    e. Lenguaje del proyecto [en]: en
    f. Crear Makefile: y
    g. Crear Archivo de comandos de Windows: y
5. Agregamos lo siguiente en el archivo config.py.

```
import sys
sys.path.append('source')
```

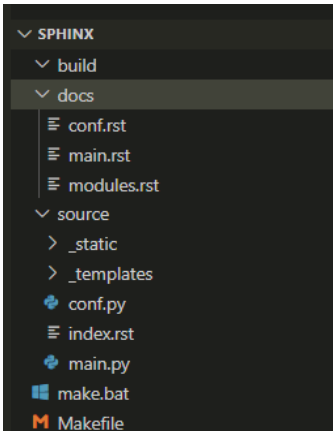
```
extensions = [
    'sphinx.ext.napoleon'
]
```

6. Agregamos docs/main al archivo index.rst

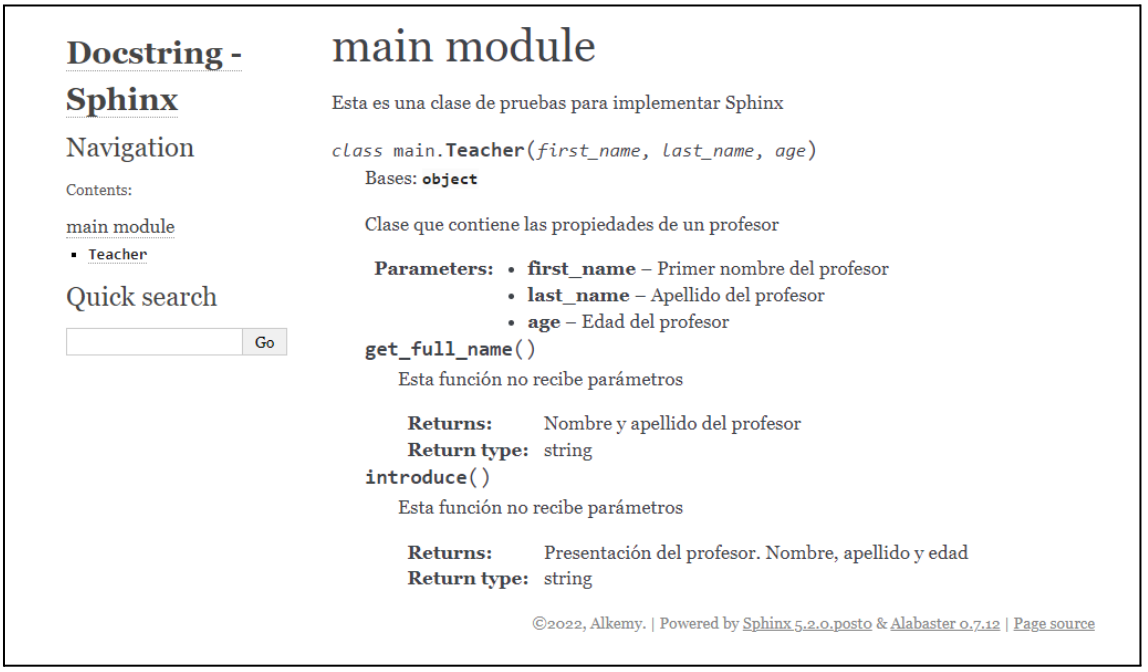
```
Welcome to Docstring - Sphinx's documentation!
==============================================

.. toctree::
   :maxdepth: 2
   :caption: Contents:

   docs/main
```

7. Ejecutar a través de la línea de comandos o la terminal del editor de código. `sphinx-apidoc -o docs source`

```
(sphinx) (base) PS C:\Users\User\Python\Proyectos\Sphinx> sphinx-apidoc -o docs source
Creando archivo docs\conf.rst.
Creando archivo docs\main.rst.
Creando archivo docs\modules.rst.
(sphinx) (base) PS C:\Users\User\Python\Proyectos\Sphinx>
```



8. Ejecutar a través de la línea de comandos o la terminal del editor de código. `make html`
9. Dirigirse con la línea de comandos o la terminal a la carpeta build/html y luego ejecutar `index.html`
10. Finalmente podremos visualizar la documentación generada.



Docstring - Sphinx

Navigation

Contents:

main module

- Teacher

Quick search

Go

main module

Esta es una clase de pruebas para implementar Sphinx

class main.**Teacher**(*first_name*, *last_name*, *age*)

Bases: **object**

Clase que contiene las propiedades de un profesor

Parameters:
- **first_name** – Primer nombre del profesor
- **last_name** – Apellido del profesor
- **age** – Edad del profesor

get_full_name()

Esta función no recibe parámetros

Returns: Nombre y apellido del profesor
Return type: string

introduce()

Esta función no recibe parámetros

Returns: Presentación del profesor. Nombre, apellido y edad
Return type: string

©2022, Alkemy. | Powered by Sphinx 5.2.0.posto & Alabaster 0.7.12 | Page source

Cierre

Durante esta unidad aprendimos sobre docstrings.

Comenzamos revisando la norma Python Enhancement Proposal (PEP) 257 donde se definen los diferentes tipos de docstring. Luego nos enfocamos en algunas buenas prácticas y las convenciones más utilizadas.

Finalmente utilizamos Sphinx para generar documentación de un archivo Python, el cual fue previamente comentado con docstrings para ese fin.

Quedamos a disposición de las consultas que puedan surgir.

Referencias

PEP 257 Docstrings Conventions | **peps.python.org.** Recuperado de: https://peps.python.org/pep-0257

Sphinx Documentation | **Sphinx.** Recuperado de: https://www.sphinx-doc.org/