2.5.-Documentar el código

2.5. Documentar el código en Python

Intentaremos explicar cómo documentar el código de Python. Ya sea un script pequeño o un proyecto grande, ya sea un principiante o un Pythonista experimentado.

La unidad se divide en cuatro secciones principales:

1. **Por qué es tan importante documentar su código:** una introducción a la documentación y su importancia
2. **Comentar vs. Documentar código:** una descripción general de las principales diferencias entre comentar y documentar, así como los momentos y formas apropiados para usar los comentarios.
3. **Documentación de la base de código de Python mediante Docstrings:** una inmersión profunda en docstrings para clases, métodos de clase, funciones, módulos, paquetes y scripts, así como lo que se debe encontrar dentro de cada uno
4. **Documentación de sus proyectos de Python:** los elementos necesarios y lo que deben contener para sus proyectos de Python

1. Por qué es tan importante documentar su código

Es posible que ya te hayas dado cuenta de la importancia de documentar el código. Pero si no, ten en cuenta esto que dijo Guido en una PyCon, creador del Python:

“Code is more often read than written.”

—Guido *van Rossum*

Cuando escribes código, lo haces dirigido principalmente a dos audiencias: los usuarios y los desarrolladores (incluido tú mismo). Ambos públicos son igualmente importantes. Con el tiempo, abrirás el código fuente que creaste en el pasado, y te preguntarás: "¿Que ... estaba intentando hacer aquí?" Si tiene problemas para tu propio código, imagínate lo que pueden experimentar los usuarios u otros desarrolladores cuando intenten usar o contribuir a tu código.

Por otra parte, con el tiempo te encontrarás en la siguiente situación, quieres hacer algo en Python y encuentras lo que parece ser una gran biblioteca que puede hacer el trabajo. Sin embargo, cuando comienzas a usar la biblioteca, buscas ejemplos, artículos o incluso documentación oficial sobre cómo hacer algo específico y te resulta difícil o imposible encontrarlo.

Después de buscar, te das cuenta de que falta algo de documentación o, lo que es peor, no hay nada de documentación. Esta situación, posiblemente te lleve a no usar la biblioteca, sin importar el trabajo que te podía haber quitado. Daniele Procida resumió mejor esta situación:

“It doesn’t matter how good your software is, because **if the documentation is not good enough, people will not use it** “

—Daniel *Procida*

2. Comentar vs Documentar Código

Antes de que podamos analizar cómo documentar su código Python, debemos distinguir la documentación de los comentarios.

En general, **comentar código** es describirlo para los desarrolladores. La audiencia principal del código fuente serán los desarrolladores que mantendrán o usarán ese código. Junto con un código bien escrito, los comentarios ayudarán a comprender mejor el código y su propósito y diseño:

"Code tells you how; Comments tell you why."

— *Jeff Atwood (también conocido como Coding Horror)*

Por otra parte, **documentar código** es describir su uso y funcionalidad a los usuarios que harán uso de este. Si bien puede ser útil en el proceso de desarrollo, la principal audiencia prevista son los usuarios. Seguidamente veremos cómo y cuándo comentar su código.

2.1. Conceptos básicos durante los comentarios en el código

En python, los comentarios se crean haciendo uso del carácter `#` al comienzo de la linea. Deben ser declaraciones breves de no más de unas pocas frases. Aquí hay un ejemplo simple:

```
def hello_world():
    # A simple comment preceding a simple print statement
    print("Hello World")
```

Según PEP 8 , los comentarios deben tener una longitud máxima de 72 caracteres. Esto es cierto, aunque tus líneas de código sean mayores que los 80 caracteres recomendados. Si un comentario va a ser mayor que el límite remendado, es apropiado usar varias líneas para el comentario:

```
def hello_long_world():
    # A very long statement that just goes on and on and on and on and
    # never ends until after it's reached the 80 char limit
    print("Hellooooooooooooooooooooooooooooooooooooooooooooooooooooooooo
World")
```

Los comentarios pueden tener múltiples propósitos, que incluyen :

- **Planificación y revisión:** cuando esté desarrollando nuevas partes de tu código, puede ser apropiado usar primero los comentarios como una forma de planificar o delinear esa sección de código. Recuerde eliminar estos comentarios una vez que se haya implementado y revisado/probado la funcionalidad real:

```
# First step
# Second step
# Third step
```

- **Descripción del código:** los comentarios se pueden usar para explicar la intención de secciones específicas del código:

```
# Attempt a connection based on previous settings. If unsuccessful,
# prompt user for new settings.
```

- **Descripción algorítmica:** cuando se usan algoritmos, especialmente los complicados, puede ser útil explicar cómo funciona el algoritmo o cómo se implementa dentro de su código. También puede ser apropiado describir por qué se seleccionó un algoritmo específico sobre otro.

```
# Using quick sort for performance gains
```

- **Etiquetado:** para etiquetar secciones específicas de código donde se encuentran problemas conocidos o áreas de mejora. Algunos ejemplos son: `BUG`, `FIXME`y `TODO`.

```
# TODO: Add condition for when val is None
```

Los comentarios a su código deben ser breves y clarificadores. Evite el uso de comentarios largos cuando sea posible. Además, debe utilizar las siguientes cuatro reglas esenciales sugeridas por Jeff Atwood :

1. Mantenga los comentarios lo más cerca posible del código que se describe. Los comentarios que no están cerca del código al que se refieren, son frustrantes para el lector y se pasan por alto fácilmente cuando se realizan actualizaciones.
2. No utilice formatos complejos (como tablas o cifras ASCII). ya que pueden distraer y pueden ser difíciles de mantener con el tiempo.
3. No incluyas información redundante. Suponga que el lector del código tiene una comprensión básica de los principios de programación y la sintaxis del lenguaje.
4. Diseña el código para que se comente a sí mismo. La forma más fácil de entender el código es leyéndolo. Cuando diseñas el código utilizando conceptos claros (variables y métodos con nombres clarificadores) y fáciles de entender, el lector entenderá la intención del código que está leyendo, sin necesidad de comentarios.

Recuerda que los comentarios están diseñados para el lector, incluido tú mismo, para ayudarlo a comprender el propósito y el diseño del software.

2.2. Comentarios a través de las sugerencias de tipo (Python 3.5+)

La sugerencia de tipo se agregó a Python 3.5 y es una forma adicional para ayudar a los lectores. Aplica la cuarta sugerencia de Jeff, ya que permite al desarrollador diseñar y explicar partes del código sin comentar. He aquí un ejemplo rápido:

```
def hello_name(name: str) -> str:
    return(f"Hello {name}")
```

Al examinar la sugerencia de tipo, inmediatamente entiendes que la función espera que la entrada `name` sea de tipo `str`. También entiendes que la salida esperada de la función será de tipo `str`. Si bien las sugerencias de tipo ayudan a reducir los comentarios, tenga en cuenta que hacerlo también puede generar trabajo adicional al crear o actualizar la documentación de su proyecto.

3. Documentar su base de código de Python usando Docstrings

Ahora que hemos aprendido a comentar, profundicemos en la documentación del código de Python. Veremos cómo usar las cadenas de documentación `docstring` y cómo usarlas para la documentación:

3.1. Cadenas de documentación `docstring`

La documentación de código Python se centra en cadenas de documentación. La propiedad `docstring` viene predefinida en los objetos y , cuando se configuran correctamente, pueden ayudar a los usuarios de este código a entender los objetos, y al desarrollador a tener documentado el proyecto. Junto con las cadenas de documentación, Python también tiene la función `help()` que imprime la cadena de documentación de los objetos en la consola. He aquí un ejemplo rápido:

```
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors are specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 # Truncated for readability
```

¿Cómo se genera esta salida? Como en Python todo es un objeto, puede examinar el directorio del objeto usando el comando `dir()`, es decir, listar los métodos y propiedades del objeto indicado. Hagamos eso y veamos qué encontramos:

```
>>> dir(str)
['__add__', ..., '__doc__', ..., 'zfill'] # Truncated for readability
```

En ese volcado, hay una propiedad interesante, `__doc__`. Si profundizamos en esta propiedad, veremos lo siguiente:

```
>>> print(str.__doc__)
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors are specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
```

Como podemos observar, en la propiedad `__doc__` se almacena la documentación del objeto. Esto significa que puedes manipular directamente esa propiedad. Sin embargo, existen restricciones que no nos permiten modificarlo los objetos predeterminados:

```
>>> str.__doc__ = "I'm a little string doc! Short and stout; here is
my input and print me for my out"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'str'
```

Cualquier otro objeto personalizado puede ser manipulado:

```
def say_hello(name):
    print(f"Hello {name}, is it me you're looking for?")

say_hello.__doc__ = "A simple function that says hello... Richie
style"
>>> help(say_hello)
Help on function say_hello in module __main__:

say_hello(name)
    A simple function that says hello... Richie style
```

Python tiene una característica más que simplifica la asignación de contenido a las docstrings. En lugar de manipular directamente la propiedad `__doc__`, la ubicación estratégica del literal debajo de la definición del objeto establecerá automáticamente el valor de `__doc__`. Esto es lo que sucede con el mismo ejemplo que el anterior:

```
def say_hello(name):
    """A simple function that says hello... Richie style"""
    print(f"Hello {name}, is it me you're looking for?")
>>> help(say_hello)
Help on function say_hello in module __main__:

say_hello(name)
    A simple function that says hello... Richie style
```

Ya que conoces el trasfondo de las docstrings. Ahora es el momento de conocer los diferentes tipos de docstrings y qué información deben contener.

3.2. Tipos de cadenas de documentos

Las convenciones de docstring se describen en PEP 257. Su propósito es proporcionar a sus usuarios una breve descripción general del objeto. Deben mantenerse lo suficientemente concisos para que sean fáciles de mantener, pero aun así ser lo suficientemente elaborados para que los nuevos usuarios entiendan su propósito y cómo usar el objeto documentado.

En todos los casos, las cadenas de documentación deben usar el formato de cadena de triples comillas dobles `"""`, ya sea con Docstrings que tengan varias líneas o no. Como mínimo, una cadena de documentación debe ser un resumen rápido de lo que sea que estés describiendo y debe estar contenida en una sola línea:

```
"""This is a quick summary line used as a description of the
object."""
```

Las cadenas de documentos de varias líneas se utilizan para realizar una descripción más elaborada del objeto más allá de un mero resumen. Todas las Docstrings compuestas por varias líneas tendrían que tener las siguientes partes:

- Una línea de resumen de una línea
- Una línea en blanco antes del resumen
- Cualquier elaboración adicional para la cadena de documentación
- Otra línea en blanco

```
"""This is the summary line

This is the further elaboration of the docstring. Within this section,
you can elaborate further on details as appropriate for the situation.
Notice that the summary and the elaboration is separated by a blank
new
line.
"""

# Notice the blank line above. Code should continue on this line.
```

Las Docstrings deben tener la misma longitud que la recomendad para los comentarios (72 caracteres). Además, se pueden dividir en tres categorías principales:

- **Class Docstrings:** clase y métodos de clase
- **Docstrings de paquetes y módulos:** paquetes, módulos y funciones
- **Script Docstrings:** Script y funciones

3.2.1. Docstrings de clase

Las Docstrings de clase se crean para la clase en sí, así como para cualquier método de clase. Se colocan inmediatamente después de la clase o el método de clase con una sangría de un nivel:

```
class SimpleClass:
    """Class docstrings go here."""

    def say_hello(self, name: str):
        """Class method docstrings go here."""

        print(f'Hello {name}')
```

Las Docstrings de clase deben contener la siguiente información:

- Un breve resumen de su propósito y comportamiento.
- Cualquier método público, junto con una breve descripción.
- Cualquier propiedad de clase (atributos)
- Cualquier cosa relacionada con la interfaz para subclases.

Los parámetros del constructor de clase deben documentarse dentro de la Docstring del método de clase `__init__` . Los métodos individuales deben documentarse utilizando sus Docstrings individuales, y deben contener lo siguiente:

- Una breve descripción de qué es el método y para qué se utiliza.
- Todos los argumentos (tanto obligatorios como opcionales) que se pasan.
- Etiquete cualquier argumento que se considere opcional o que tenga un valor predeterminado
- Cualquier efecto secundario que ocurra al ejecutar el método.
- Cualquier excepción que se plantee
- Cualquier restricción sobre cuándo se puede llamar al método

Tomemos un ejemplo simple de una clase de datos que representa un Animal. Esta clase contendrá algunas propiedades de clase, propiedades de instancia, un método `__init__`, y un método de instancia:

```
class Animal:
    """
    A class used to represent an Animal

    ...

    Attributes
    ----------
    says_str : str
        a formatted string to print out what the animal says
    name : str
        the name of the animal
    sound : str
        the sound that the animal makes
    num_legs : int
        the number of legs the animal has (default 4)

    Methods
    -------
    says(sound=None)
        Prints the animals name and what sound it makes
    """

    says_str = "A {name} says {sound}"

    def __init__(self, name, sound, num_legs=4):
        """
        Parameters
        ----------
        name : str
            The name of the animal
        sound : str
            The sound the animal makes
        num_legs : int, optional
            The number of legs the animal (default is 4)
        """

        self.name = name
        self.sound = sound
        self.num_legs = num_legs

    def says(self, sound=None):
        """
        Prints what the animals name is and what sound it makes.

        If the argument `sound` isn't passed in, the default Animal
        sound is used.
```

```
        Parameters
        ----------
        sound : str, optional
            The sound the animal makes (default is None)

        Raises
        ------
        NotImplementedError
            If no sound is set for the animal or passed in as a
            parameter.
        """

        if self.sound is None and sound is None:
            raise NotImplementedError("Silent Animals are not
supported!")

        out_sound = self.sound if sound is None else sound
        print(self.says_str.format(name=self.name, sound=out_sound))
```

3.2.2. Docstrings de paquetes y módulos

Las **docstring del paquete** deben colocarse en la parte superior del archivo `__init__.py` del paquete. Esta cadena de documentación debe enumerar los módulos y subpaquetes que exporta el paquete.

Las **docstring del módulo** son similares a las Docstrings de la clase, salvo que en lugar de que se documenten las clases y los métodos de clase, ahora es el módulo y las funciones que se encuentran dentro. Las Docstrings del módulo se colocan en la parte superior del archivo incluso antes de cualquier importación, y deben incluir lo siguiente:

- Una breve descripción del módulo y su propósito.
- Una lista de cualquier clase, excepción, función y cualquier otro objeto exportado por el módulo

La **docstring para una función de módulo** debe incluir los mismos elementos que un método de clase:

- Una breve descripción de qué es la función y para qué se utiliza.
- Todos los argumentos (tanto obligatorios como opcionales) que se pasan.
- Etiquete cualquier argumento que se considere opcional
- Cualquier efecto secundario que ocurra al ejecutar la función
- Cualquier excepción que se plantee
- Cualquier restricción sobre cuándo se puede llamar a la función

3.2.3. Docstrings de scripts

Los scripts se consideran ejecutables de un solo archivo que se ejecutan desde la consola. Las docstrings para los scripts se colocan en la parte superior del archivo y deben documentarse lo suficientemente bien como para que los usuarios puedan tener una comprensión suficiente de cómo usar el script. Debería poder usarse para obtener un mensaje de como "usar el script", cuando el usuario pasa incorrectamente un parámetro o usa la opción `-h` al ejecutar el script.

Si usa `argparse`, puede omitir la documentación específica, suponiendo que se haya documentado correctamente dentro del parámetro `help` de la función `argparser.parser.add_argument`. Se recomienda usar la propiedad `__doc__`para el parámetro `description` del constructor `argparse.ArgumentParser`. Consulte este tutorial sobre bibliotecas de análisis de línea de comandos para obtener más detalles sobre cómo usar `argparse`y otros analizadores de línea de comandos comunes.

Finalmente, cualquier importación personalizada o de terceros debe incluirse en las docstring para permitir a los usuarios saber qué paquetes pueden ser necesarios para ejecutar el script. Aquí hay un ejemplo de un script que se usa para imprimir los encabezados de las columnas de una hoja de cálculo:

```
"""Spreadsheet Column Printer

This script allows the user to print to the console all columns in the
spreadsheet. It is assumed that the first row of the spreadsheet is
the
location of the columns.

This tool accepts comma separated value files (.csv) as well as excel
(.xls, .xlsx) files.

This script requires that `pandas` be installed within the Python
environment you are running this script in.

This file can also be imported as a module and contains the following
functions:

    * get_spreadsheet_cols - returns the column headers of the file
    * main - the main function of the script
"""

import argparse

import pandas as pd


def get_spreadsheet_cols(file_loc, print_cols=False):
    """Gets and prints the spreadsheet's header columns

    Parameters
    ----------
    file_loc : str
        The file location of the spreadsheet
    print_cols : bool, optional
        A flag used to print the columns to the console (default is
        False)

    Returns
    -------
    list
        a list of strings used that are the header columns
    """

    file_data = pd.read_excel(file_loc)
    col_headers = list(file_data.columns.values)

    if print_cols:
```

```
        print("\n".join(col_headers))

    return col_headers


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument(
        'input_file',
        type=str,
        help="The spreadsheet file to pring the columns of"
    )
    args = parser.parse_args()
    get_spreadsheet_cols(args.input_file, print_cols=True)


if __name__ == "__main__":
    main()
```

3.3. Formatos de docstrings

Es posible que haya notado en los ejemplos vistos hasta ahora de docstrings que existían elementos comunes: `Arguments`, `Returns`y `Attributes`. Hay formatos específicos de docstrings que se pueden usar para ayudar a los analizadores de docstrings y a los usuarios a tener un formato familiar y conocido. El formato utilizado para los docstrings sigue el estilo NumPy/SciPy. Algunos de los formatos más comunes son los siguientes:

Tipo de formato	Descripción	Con el apoyo de Sphynx	especificación formal
docstrings de Google	Forma de documentación recomendada por Google	Sí	No
Texto reestructurado	Estándar de documentación oficial de Python; No es amigable para principiantes pero tiene muchas funciones	Sí	Sí
docstrings NumPy/SciPy	La combinación de NumPy de reStructuredText y Google Docstrings	Sí	Sí
epitexto	Una adaptación Python de Epydoc; Ideal para desarrolladores de Java	no oficialmente	Sí

La selección del formato a seguir en la creación de las docstrings es decisión personal, pero una vez eligida una, hay que ceñirse al mismo formato en todo el documento/proyecto. Los siguientes son ejemplos de cada tipo para darle una idea de cómo se ve cada formato de documentación.

3.3.1. Ejemplo de docstrings de Google

```
"""Gets and prints the spreadsheet's header columns

Args:
    file_loc (str): The file location of the spreadsheet
    print_cols (bool): A flag used to print the columns to the console
        (default is False)
```

```
Returns:
    list: a list of strings representing the header columns
"""
```

3.3.2. Ejemplo de texto reestructurado

```
"""Gets and prints the spreadsheet's header columns

:param file_loc: The file location of the spreadsheet
:type file_loc: str
:param print_cols: A flag used to print the columns to the console
    (default is False)
:type print_cols: bool
:returns: a list of strings representing the header columns
:rtype: list
"""
```

3.3.3. Ejemplo de cadenas de documentación NumPy/SciPy

```
"""Gets and prints the spreadsheet's header columns

Parameters
----------
file_loc : str
    The file location of the spreadsheet
print_cols : bool, optional
    A flag used to print the columns to the console (default is False)

Returns
-------
list
    a list of strings representing the header columns
"""
```

3.3.4. Ejemplo de epitexto

```
"""Gets and prints the spreadsheet's header columns

@type file_loc: str
@param file_loc: The file location of the spreadsheet
@type print_cols: bool
@param print_cols: A flag used to print the columns to the console
    (default is False)
@rtype: list
@returns: a list of strings representing the header columns
"""
```

4. Documentación de sus proyectos de Python

Los proyectos de Python vienen en todo tipo de formas, tamaños y propósitos. La forma en que documente su proyecto debe adaptarse a su situación específica. Ten en cuenta quiénes van a ser los usuarios de tu proyecto y adáptate a sus necesidades. Dependiendo del tipo de proyecto, se recomiendan ciertos aspectos de la documentación. El diseño general del proyecto y su documentación debe ser el siguiente:

```
project_root/
│
├── project/  # Project source code
├── docs/
├── README
├── HOW_TO_CONTRIBUTE
├── CODE_OF_CONDUCT
├── examples.py
```

Los proyectos se pueden subdividir generalmente en tres tipos principales: Privado, Compartido y Público/Código Abierto.

4.1. Proyectos Privados

Los proyectos privados son proyectos destinados solo para uso personal y, por lo general, no se comparten con otros usuarios o desarrolladores. La documentación puede ser bastante ligera en este tipo de proyectos. La documentación recomendada para este tipo de proyectos, según sea necesario:

- **Readme.md:** un breve resumen del proyecto y su propósito. Incluya cualquier requisito especial para la instalación u operación del proyecto.
- **`examples.py`:** un archivo de secuencia de comandos de Python que brinda ejemplos simples de cómo usar el proyecto.

Recuerda, aunque los proyectos privados están destinados a ti, también eres considerado un usuario. Piense en cualquier cosa que pueda resultarle confusa en el futuro y asegúrese de capturarla en comentarios, docstring o el archivo Readme.md.

4.2. Proyectos Compartidos

Los proyectos compartidos son proyectos en los que colaboras con otras personas en el desarrollo y/o uso del proyecto. El "cliente" o usuario del proyecto sigue siendo usted mismo y otros desarrolladores que utilizan el proyecto.

La documentación debe ser un poco más rigurosa de lo que debe ser para un proyecto privado, principalmente para ayudar a incorporar nuevos miembros al proyecto o alertar a los contribuyentes/usuarios de nuevos cambios en el proyecto. La documentación recomendada para estos proyectos es la siguiente:

- **Readme.md:** un breve resumen del proyecto y su propósito. Incluya cualquier requisito especial para instalar u operar el proyecto. Además, agregue cualquier cambio importante desde la versión anterior.
- **`examples.py`:** un archivo de secuencia de comandos de Python que brinda ejemplos simples de cómo usar los proyectos.
- **Cómo contribuir:** esto debe incluir cómo los nuevos contribuyentes al proyecto pueden comenzar a contribuir.

4.3. Proyectos públicos y de código abierto

Los proyectos públicos y de código abierto son proyectos que están destinados a compartirse con un gran grupo de usuarios y pueden involucrar a grandes equipos de desarrollo. Estos proyectos deben otorgar una prioridad tan alta a la documentación del proyecto como al desarrollo real del proyecto en sí. La documentación recomendada para estos proyectos es las siguiente:

- **Readme.md:** un breve resumen del proyecto y su propósito. Incluya cualquier requisito especial para instalar u operar los proyectos. Además, agregue cualquier cambio importante desde la versión anterior. Finalmente, agregue enlaces a documentación adicional, informes de errores y cualquier otra información importante

para el proyecto. Dan Bader ha elaborado un excelente tutorial sobre todo lo que debe incluirse en su archivo Léame.
- **Cómo contribuir:** esto debe incluir cómo pueden ayudar los nuevos contribuyentes al proyecto. Esto incluye el desarrollo de nuevas funciones, la solución de problemas conocidos, la adición de documentación, la adición de nuevas pruebas o la notificación de problemas.
- **Código de conducta:** define cómo deben comportarse los demás colaboradores al desarrollar o utilizar su software. Esto también establece lo que sucederá si este código no es correcto. Si está utilizando Github, se puede generar una plantilla de Código de conducta con la redacción recomendada. Especialmente para proyectos de código abierto, considere agregar esto.
- **Licencia:** un archivo de texto sin formato que describe la licencia que utiliza su proyecto. Especialmente para proyectos de código abierto, considere agregar esto.
- **docs:** una carpeta que contiene más documentación. La siguiente sección describe con más detalle qué debe incluirse y cómo organizar el contenido de esta carpeta.

4.3.1. Las cuatro secciones principales de la carpeta `docs`

Daniele Procida dio una maravillosa charla sobre PyCon 2017 y una publicación de blog posterior sobre la documentación de proyectos de Python. Menciona que todos los proyectos deben tener las siguientes cuatro secciones principales para ayudarlo a enfocar su trabajo:

- **Tutoriales**: Lecciones que llevan al lector de la mano a través de una serie de pasos para completar un proyecto (o ejercicio significativo). Orientado al aprendizaje del usuario.
- **Guías prácticas**: guías que llevan al lector a través de los pasos necesarios para resolver un problema común (Recetas orientadas a resolver problemas).
- **Referencias**: Explicaciones que aclaran e iluminan un tema en particular. Orientado a la comprensión.
- **Explicaciones**: descripciones técnicas de la maquinaria y cómo operarla (clases clave, funciones, API, etc.). Artículo de Think Encyclopedia.

La siguiente tabla muestra cómo todas estas secciones se relacionan entre sí, así como su propósito general:

	Más útil cuando estamos estudiando	**Más útil cuando estamos programando**
Paso práctico	*Tutoriales*	*Guías prácticas*
Conocimientos teóricos	*Explicación*	*Referencia*

Al final, deseas asegurarte de que los usuarios tienen acceso a las respuestas a cualquier pregunta que puedan tener. Al organizar el proyecto de esta manera, podrás responder esas preguntas fácilmente y en un formato que podrán navegar rápidamente.

4.4. Herramientas y recursos de documentación

Documentar su código, especialmente proyectos grandes, puede ser desalentador. Afortunadamente, existen algunas herramientas y referencias para comenzar:

Herramienta	Descripción
Esfinge	Una colección de herramientas para autogenerar documentación en múltiples formatos
Epydoc	Una herramienta para generar documentación de API para módulos de Python basada en sus docstrings
Leer los documentos	Creación, control de versiones y alojamiento automáticos de sus documentos.
doxígeno	Una herramienta para generar documentación compatible con Python, así como con muchos otros lenguajes.
MkDocs	Un generador de sitios estáticos para ayudar a construir la documentación del proyecto utilizando el lenguaje Markdown. Consulta Crea tu documentación de proyecto de Python con MkDocs para obtener más información.
pycco	Un generador de documentación "rápido y sucio" que muestra el código y la documentación uno al lado del otro. Consulta Tutorial sobre cómo usar pycco .
pydoc	El módulo pydoc genera automáticamente documentación a partir de módulos de Python. La documentación puede presentarse como páginas de texto en la consola, enviarse a un navegador web o guardarse en archivos HTML. .

Junto con estas herramientas, hay algunos tutoriales, videos y artículos adicionales que pueden ser útiles cuando esté documentando su proyecto:

1. Carol Willing - Práctica Esfinge - PyCon 2018
2. Daniele Procida - Desarrollo basado en documentación - Lecciones del Proyecto Django - PyCon 2016
3. Eric Holscher - Documentando su proyecto con Sphinx & Read the Docs - PyCon 2016
4. Titus Brown, Luiz Irber - Crear, construir, probar y documentar un proyecto de Python: un CÓMO práctico - PyCon 2016
5. reStructuredText Documentación Oficial
6. Manual de texto reestructurado de Sphinx
7. El generador de documentación y sistema de ayuda en línea pydoc

A veces, la mejor forma de aprender es imitando a los demás. Aquí hay algunos excelentes ejemplos de proyectos que usan bien la documentación:

- **Django:** Documentos (Fuente)
- **Solicitudes:** Documentos (Fuente)

- **Haga clic en:** Documentos (Fuente)
- **Pandas:** Documentos (Fuente)

5. ¿Dónde empiezo?

La documentación de proyectos tiene una progresión sencilla:

1. Sin documentación
2. Algo de documentación
3. Documentación completa
4. buena documentación
5. Gran documentación

Si no sabes por dónde empezar con la documentación, identifica dónde se encuentra tu proyecto ahora en relación con la progresión anterior. ¿Tienes alguna documentación? Si no, entonces comience allí. Si tiene alguna documentación, pero le faltan algunos de los archivos clave del proyecto, comience agregándolos.

Al final, no te desanimes ni te sientas abrumad@ por la cantidad de trabajo que se requiere para documentar el código. Una vez que comienzas a documentar el código, será más fácil continuar.

Fuente

- Documentando código python

2.5.1-¿Deberías documentar tu código?

2.5.1. ¿Deberías documentar tu código?

Hay programadores que dice que los comentarios son un mal que se debería evitar al máximo. Sin embargo, aquí proponemos lo contrario: **usa los comentarios correctamente para crear código mantenible, basados en las ideas de 'A Philosophy of Software Design'**

Cuando hablamos de los comentarios en el código, hay dos escuelas. La primera dice que debes usar los comentarios para **clarificar lo que quisiste expresar con tu código**, mientras que la segunda dice que deberías **evitarlos al máximo** y que comentar tu código es un mal necesario que sólo denota tu falta de habilidad para no hacer código lo suficientemente claro.

¿A cuál de los dos consejos deberías hacerle caso? pienso que deberías ver los comentarios como una **herramienta necesaria**, valiosa y muy útil, y cómo usarlos para no caer en el extremo que ha llevado a algunas personas a tener una mala actitud hacia ellos.

1. Un sistema sin documentación está incompleto

Como futuro desarrollador estarás de acuerdo en que un sistema **no** tiene la *calidad suficiente* si no cuenta con documentación, es decir, información acerca del sistema que comunique cosas como la razón de existir de ciertos módulos, valores y funciones y cómo usarlos.

Si, además, tienes que modificar este sistema, será una pesadilla entender todo lo que los programadores anteriores hicieron o *intentaron* hacer. Si tienes que *usar* algo sin documentación, es el mismo caso: **tienes estudiar el código para saber cómo funciona.**

Por cosas como las anteriores, la documentación es completamente necesaria para crear programas útiles. Ahora bien, ¿dónde ponemos esa documentación? Muchos desarrolladores y equipos no tienen idea de dónde ponerla y crean documentos que dejan después olvidados en una carpeta en la nube y que nadie encuentra después. Pero, ¿no sería más lógico mantener la documentación lo **más cerca posible del código**? Eso es precisamente lo que los comentarios te permiten hacer.

Puedes usar los comentarios documentar:

- Decisiones de diseño
- Explicaciones sobre la existencia, funcionamiento o razón de ser de cierta parte del código
- Las interfaces y su ejemplo de uso
- Efectos de usar cierto código
- Partes inconclusas o que se pueden mejorar (TODO's)

Tener esta información muy cerca del código sobre el que está proporcionando información ayudará a que sea fácil de encontrar y además, si se establecen reglas como tratar los comentarios como ciudadanos de primer rango, se mantendrá actualizado y útil.

También es buena idea tener un documento o sitio web especializado en documentación que te ayude a encontrar rápido lo que buscas como Docusaurus o un sitio generado por Sophinx. Puedes utilizar esta misma documentación que escribiste junto al código si usaste el estilo definido por el lenguaje de programación o por las herramientas de generación de documentos.

1.1. Los comentarios te pueden ayudar en el futuro

Incluso aunque no los uses formalmente como documentación, los comentarios estarán ahí para darte información y recordarte lo que hiciste, pero sobre todo **por qué** lo hiciste.

Recuerda que la mente humana busca la eficiencia máxima de recursos, por lo que es probable que elimine información que no ocupe inmediatamente y que no recuerdas a menudo, como por qué esa variable tenía el valor 730 y no otro.

Tu yo futuro y tu equipo te agradecerán haber escrito esos comentarios que te informan sobre lo que estabas pensando en el momento que escribiste ese código.

1.2. Los comentarios son una buena herramienta de diseño

John Ousterhout, en "A Philosophy of Software Design" recomienda **empezar** con los comentarios antes de programar (de esto hablaremos más adelante). Pero, ¿por qué lo recomienda?

Escribir en un lenguaje humano cómo funciona algo antes de implementarlo realmente, te da la capacidad de ver si es lógico y suficiente, además te permite ponerte en los zapatos del usuario para notar deficiencias sobre todo en **la interfaz**. Los comentarios de interfaz es lo primero que deberías crear porque te servirán de guía para avanzar con tu diseño y, sobre todo, que sea lógico y fácil de usar.

Una buena guía: si no eres capaz de crear un comentario concreto y corto sobre cómo funciona o por qué existe algo, **lo más probable es que tengas que re-pensar tu diseño**.

1.3. El lenguaje de programación no es suficiente para expresar todo lo necesario

Todos los lenguajes de programación están pensados para ser un **subconjunto del lenguaje humano** que elimine las ambigüedades, manteniendo el mayor poder expresivo posible. Esto nos lleva a sus limitantes: es imposible, o por lo menos impráctico, intentar expresar todas las ideas con el código.

En la práctica, el tiempo y los recursos para lograr algo son limitados, por lo que a veces es más conveniente y fácil para todos explicar lenguaje humano algo que intentar expresarlo con código, como los puristas afirman.

No te sientas mal si tienes que recurrir de vez en cuando a explicar la forma en que funciona algo, siempre y cuando no sea la práctica común.

2. ¿Cómo usar los comentarios para que sean valiosos?

No todos los comentarios son valiosos, hay algunos que pueden estorbar más de lo que ayudan, por ejemplo, los que no aportan información a lo que es obvio en el código.

Hablemos de algunas formas de aprovecharlos lo mejor posible para que contribuyan positivamente a aumentar la calidad del proyecto.

2.1. Escribe los comentarios primero

Una de las partes más importantes de los comentarios como documentación es que deben ser concretos, cercanos a la realidad y que proporcionen la mayor cantidad de información útil posible.

Para lograr esto, se tienen que crear lo más cerca que puedas a la *creación del código*. Pero como todos sabemos que después de escribir y probar (básicamente) el código vamos a sentir que ya está terminado, es buena práctica obligarte a escribirlos antes, justo como propone TDD con las pruebas.

De esta manera te asegurarás que tu código esté documentado incluso antes de escribirlo y te servirán como una **herramienta de diseño** que te ayudará a pensar mejor en la usabilidad de tus módulos y piezas de software.

2.2. Crea comentarios acerca de la interfaz

La interfaz es el **medio de uso** que tus módulos o funciones presentan para que las demás partes de tu sistema lo usen. Lo primero que deberías documentar y explicar es **esta interfaz**, para que más personas a parte de ti puedan usar este pedazo de código.

Debes escribir comentarios claros sobre:

- **Cómo usar esa pieza de código**
- **Por qué existe esa parte del sistema**
- **Qué efectos tiene usarla**

Este tipo de comentarios son los que aportan mayor valor al sistema y si están lo suficientemente completos, con ejemplos y explicaciones claras, son una documentación válida que está en un muy buen lugar: es fácil de encontrar y no se va a perder enterrada entre otros documentes que después nadie va a consultar.

Evita los comentarios sobre la implementación

Los comentarios sobre la implementación son aquello que describen *qué* estás haciendo, como por ejemplo, sumar número, abrir un archivo, etc. Estos comentarios normalmente son innecesarios, ya que lo que se está haciendo es obvio si el código es lo suficientemente expresivo y *siempre deberíamos buscar que sea así*.

De hecho, estos son los comentarios que hacen que la gente odie a los comentarios en general, pues en vez de proporcionar información extra son una carga que hay que mantener y pueden confundir si no son actualizados.

Si realmente sientes que tienes que explicar *qué* estás haciendo con cierta pieza de código, primero pregúntate si no hay una manera de reescribirlo para que **sea obvio**. Si no existe o no es práctica esta solución, entonces escribe el comentario de la manera más concisa posible, incluyendo la razón de la existencia de ese código.

Para hacer esto debes tomar muy en cuenta los recursos del proyecto: no te puedes tardar el triple del tiempo implementando la pieza de código perfecta porque no quieres escribir un comentario que explique cómo funciona.

3. Conclusión

Escribir comentarios es una de las grandes tareas que los programadores debemos dominar. Los lenguajes de programación y los entornos de programación cada vez le dan más poder a esta parte de los programas y permiten incluso escribir pruebas en ellos, generar documentación automática y listar tareas a partir de ellos.

Si pones el suficiente esmero en aprender a escribir buenos comentarios y mantenerlos, serán una gran herramienta de diseño y documentación de tu software.

*Este artículo está basado en las ideas del "A Philosophy of Software Design de John Ousterhout", en el que se le dedican **4 capítulos** al buen uso de los comentarios*.

Fuente

- ¿Deberías documentar tu código? - Héctor Patricio