Loguear Eventos II

(Data Engineer)

tiimiit by

Introducción

Bienvenidos al módulo 4 - Loguear Eventos II. En el mismo, los objetivos de aprendizaje estarán orientados a incorporar conocimientos teóricos y prácticos el logging de eventos utilizando Python.

Una vez finalizado este módulo serás capaz de:

- Configurar un logging utilizando un archivo de configuración.
- Utilizar buenas prácticas para tus próximos loggins
- Comprender sobre la utilización de logging utilizando diferentes módulos.

Tema 1. Configuración de logging mediante un archivo

Configuración mediante un archivo

En el documento “Loguear Eventos I” aprendimos a configurar nuestro log utilizando el método basic config y también a crear un logger personalizado. Ahora aprenderemos a realizar esta tarea mediante un archivo de configuración que podemos almacenar de forma separada.

El archivo de configuración debe tener una estructura predeterminada, donde se incluyen las secciones **[loggers]**, **[handlers]** y **[formatters]** que identifican por nombre las entidades de cada tipo que se definen en el archivo.

Para cada una de esas entidades, hay una sección separada que identifica cómo se configura esa entidad. Por lo tanto, para un logger llamado **root** en la sección [loggers], los detalles de configuración relevantes se encuentran en una sección **[logger_root]**

```
[loggers]
keys=root,log02,log03
```

```
[logger_root]
level=NOTSET
handlers=hand01
```

Del mismo modo, un handler llamado **hand01** en la sección **[handlers]** tendrá su configuración en una sección llamada **[handler_hand01]**, mientras que un formatter llamado **form01** en la sección **[formatters]** tendrá su configuración especificada en una sección llamada **[formatter_form01]**.

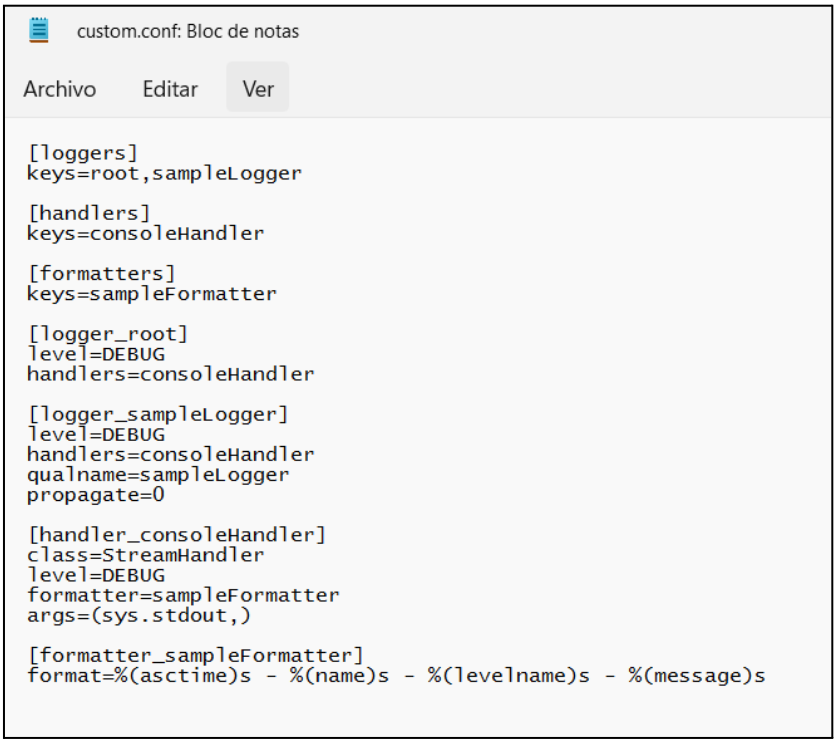
En el siguiente ejemplo podemos visualizar la implementación de las secciones loggers, handlers y formatters.

```
[loggers]
keys=root,log02,log03,log04,log05,log06,log07

[handlers]
keys=hand01,hand02,hand03,hand04,hand05,hand06,hand07,hand08,hand09

[formatters]
keys=form01,form02,form03,form04,form05,form06,form07,form08,form09
```

y como se configura el handler **hand01**

```
[handler_hand01]
class=StreamHandler
level=NOTSET
formatter=form01
args=(sys.stdout,)
```

En el siguiente archivo, estamos implementando la configuración del logger Root y también de un logger personalizado llamado **sampleLogger**. Ambos, utilizan un handler con un formato “name /level/message” que imprime el registro por pantalla.



Para implementar la configuración utilizamos el método fileConfig.

```
import logging
import logging.config

logging.config.fileConfig(fname='custom.conf', disable_existing_loggers=False)

# Get the logger specified in the file
logger = logging.getLogger(__name__)

logger.debug('Este es un mensaje de debug')
```

```
2022-06-07 20:24:39,079 - __main__ - DEBUG - Este es un mensaje de debug
```

Buenas prácticas para utilizar Logging

Cuando desarrollamos scripts, nos puede resultar útil visualizar alguna variable en particular o un estado para comprender qué es lo que está sucediendo. En ese sentido, generalmente utilizamos impresiones en la consola o a veces escribimos estos valores en un archivo. Este tipo de solución tiene su precio, y es que no estamos utilizando los beneficios del logging.

El enfoque primitivo, de utilizar impresiones o guardar resultados en archivos, puede ser suficiente para una aplicación trivial, pero nos dará dolores de cabeza cuando esa aplicación comience a crecer. Estos registros que vamos a estar generando, estarán llenos de ruido y no nos permitirán administrar la severidad.

Además, ¿qué hay de la flexibilidad para cambiar fácilmente los destinos de los registros? Por ejemplo, si en algún momento deseamos que un mensaje en vez de imprimirlo por la consola, ahora se guarde en un archivo, o si queremos ajustar el formato de estos registros. En ese escenario es cuando comenzamos a utilizar el logging.

A continuación veremos algunas buenas prácticas para tener en cuenta.

1. Utilizar el módulo estándar de logging

Este módulo fue diseñado para ser flexible y fácil de utilizar. Podemos definir fácilmente los handlers y ajustar su formato, lo cual nos

permite lograr combinaciones muy valiosas para mejorar nuestros programas.

2. Utilizar los niveles de severidad correctos

Python nos ofrece 5 niveles de severidad que permiten clasificar nuestros registros. Acá detallamos algunos casos típicos para utilizar cada uno.

- **DEBUG**: Se utiliza únicamente para realizar análisis en la etapa de desarrollo del proyecto.
- **INFO**: Se utiliza cuando algo interesante sucede y necesitamos visualizarlo. Por ejemplo: Cuando se inicia una tarea en un programa que compila N tareas.
- **WARNING**: Se utiliza cuando sucede algo inesperado, pero que aún así no es un error. Podemos utilizarlo cuando una variable toma un valor que se encuentra fuera de los parámetros esperados. Por ejemplo, si estamos midiendo la temperatura en una cámara frigorífica, tendremos un warning cuando ésta se encuentra por fuera del umbral requerido para conservar los alimentos..
- **ERROR**: En este nivel podemos clasificar aquellas acciones que salieron mal, pero que aún así pueden recuperarse. Aplica a excepciones que podemos manejar. Por ejemplo, en el caso de que estemos consultando una API y tengamos un error.
- **CRITICAL**: Este es el peor escenario posible y nuestro programa ya no podrá continuar funcionando.

3. Incluir el timestamp en cada entrada del registro

Saber que algo sucedió sin saber cuándo sucedió es solo marginalmente mejor que no saber nada sobre el evento.

Debemos asegurarnos de agregar un timestamp o marca de tiempo en cada entrada del registro para facilitar la vida de las personas que se ocupan de solucionar los problemas. Si lo hacemos, también permitiremos a los desarrolladores analizar las entradas de registro para obtener información/análisis sobre el comportamiento del usuario.

4. Adoptar el formato ISO-8601 para el timestamp

Si bien podemos optar por utilizar un formato de timestamp que se ajuste a las convenciones de nuestro país, como por ejemplo un formato DD/MM/YYYY, esta opción quizás no sea válida en el caso de que tengamos que compartir nuestro programa con alguna persona de otro país.

Además, puede ser el caso de que estemos importando en nuestro programa, librerías de terceros que ya tengan su logging configurado y que estén utilizando otro formato de timestamp

Por esas razones, es conveniente familiarizarse y utilizar un estándar que posiblemente sea común en la comunidad de programadores. Ese formato estándar existe y se llama ISO-8601. Es un estándar internacional para el intercambio de datos relacionados con la fecha y la hora.

Un ejemplo de un timestamp expresado en formato ISO-8601 es el siguiente:

```
2020-03-14T15:00-03:00
```

Para configurarlo tendremos que agregar el modificador de formato

```
datefmt="%Y-%m-%dT%H:%M:%S%z"
```

5. Utilizar la clase FileHandler

Si bien al principio puede parecer más fácil guardar todos los registros en un solo archivo, se considera una buena práctica distribuir los registros en varios archivos, especialmente si tenemos registros extensos.

Tener un solo archivo de registro masivo puede generar un rendimiento deficiente, ya que el sistema necesita abrir y cerrar el archivo cada vez que registra un nuevo mensaje de registro. Un archivo de registro de 500 MB tardará mucho más en abrirse y cerrarse que uno limitado a 2 MB.

Afortunadamente, no tenemos que implementar esto a mano en Python. En su lugar, podemos utilizar la clase **RotatingFileHandler** en lugar de la clase **FileHandler** normal.

Propagación de Logs

Si este atributo se evalúa como verdadero, los eventos registrados en este logger se pasarán a los handlers de loggers de nivel superior (ancestros), además de cualquier handler adjunto a este logger.

Si esto se evalúa como falso, los mensajes del logger no se pasan a los handlers de los loggers de nivel superior. El constructor del objeto setea este parámetro como verdadero por defecto.

Ejercicio Práctico 1

En este ejercicio vamos a implementar un script que cuente la cantidad de palabras que se encuentran en un archivo txt.

Comenzaremos creando un módulo llamado **lowermodule.py** donde vamos a procesar el archivo txt y vamos a calcular la cantidad de palabras. Este módulo tendrá un logger personalizado que nos informará el estado de la función.

lowermodule.py > word_count

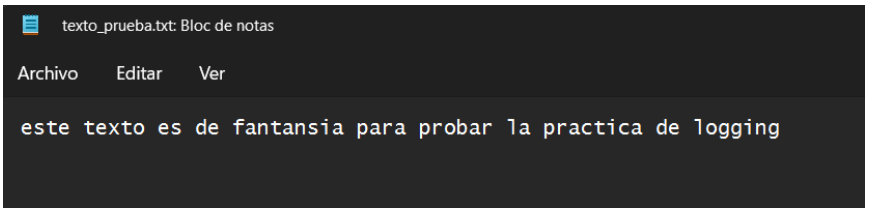
```python
import logging

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(name)s %(levelname)s:%(message)s', datefmt="%Y-%m-%dT%H:%M:%S%z")
logger = logging.getLogger(__name__)

def word_count(myfile):
    try:
        with open(myfile, 'r') as f:
            file_data = f.read()
            words = file_data.split(" ")
            final_word_count = len(words)
            logger.info("this file has %d words", final_word_count)
            return final_word_count
    except OSError as e:
        logger.error("error reading the file")
```

Luego tendremos el módulo **uppermodule.py** donde vamos a importar el módulo anterior y también vamos a guardar los resultados de la operación en un archivo csv.

El módulo **uppermodule.py** también tiene su propio logger configurado con un nivel de severidad del tipo DEBUG.

```python
import logging
import lowermodule

logging.basicConfig(level=logging.DEBUG, format='%(asctime)s %(name)s %(levelname)s:%(message)s', datefmt="%Y-%m-%dT%H:%M:%S%z")
logger = logging.getLogger(__name__)

def record_word_count(myfile):
    logger.info("starting the function")
    try:
        word_count = lowermodule.word_count(myfile)
        with open('wordcountarchive.csv', 'a') as file:
            row = str(myfile) + ',' + str(word_count)
            file.write(row + '\n')
    except:
        logger.warning("could not write file %s to destination", myfile)
    finally:
        logger.debug("the function is done for the file %s", myfile)

def main():
    record_word_count('texto_prueba.txt')

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

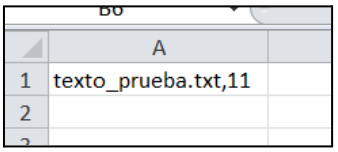
Finalmente al procesar el siguiente archivo txt



Obtenemos los siguientes logs

```
(base) PS C:\Users\User\Python\Notebooks> & C:/Users/User/anaconda3/python.exe c:/Users/User/Python/Notebooks/uppermodule.py
2022-09-17T16:26:05-0300 __main__ INFO:starting the function
2022-09-17T16:26:05-0300 lowermodule INFO:this file has 11 words
2022-09-17T16:26:05-0300 __main__ DEBUG:the function is done for the file texto_prueba.txt
(base) PS C:\Users\User\Python\Notebooks>
```

y al abrir el archivo wordcountarchive.csv podemos ver el resultado de la operación.



el siguiente ejemplo práctico vamos a configurar dos handlers, el prim

Ejercicio Práctico 2

En este ejercicio vamos a implementar la clase RotatingFileHandler, que nos va a permitir separar nuestros logs en diferentes archivos de texto.

```
import logging
from logging.handlers import RotatingFileHandler

# Creamos el logger
logger = logging.getLogger('simple_logger')

# Establecemos el nivel de severidad
logger.setLevel(logging.DEBUG)

# Creamos un rotating file handler y seteamos el nivel de severidad en DEBUG
handler = RotatingFileHandler('my_log.log', maxBytes=2000, backupCount=50)
handler.setLevel(logging.DEBUG)

# Creamos un objeto de formato
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# Agregamos el objeto formato al rotating file handler
handler.setFormatter(formatter)

# Agregamos el handler al logger
logger.addHandler(handler)

#Generamos los logs
for i in range(10000):

    logger.debug('debug message {}'.format(i))
    logger.info('info message {}'.format(i))
    logger.warning('warn message {}'.format(i))
    logger.error('error message {}'.format(i))
    logger.critical('critical message {}'.format(i))
```

Cómo funciona

Utilizamos un for loop para generar 10.000 ciclos donde cada uno de ellos genera 5 registros (uno por cada nivel de severidad). El controlador **files.RotatingFileHander** rotará los archivos de registro en función de un tamaño máximo configurado.

Esta configuración la establecemos con los siguientes parámetros.

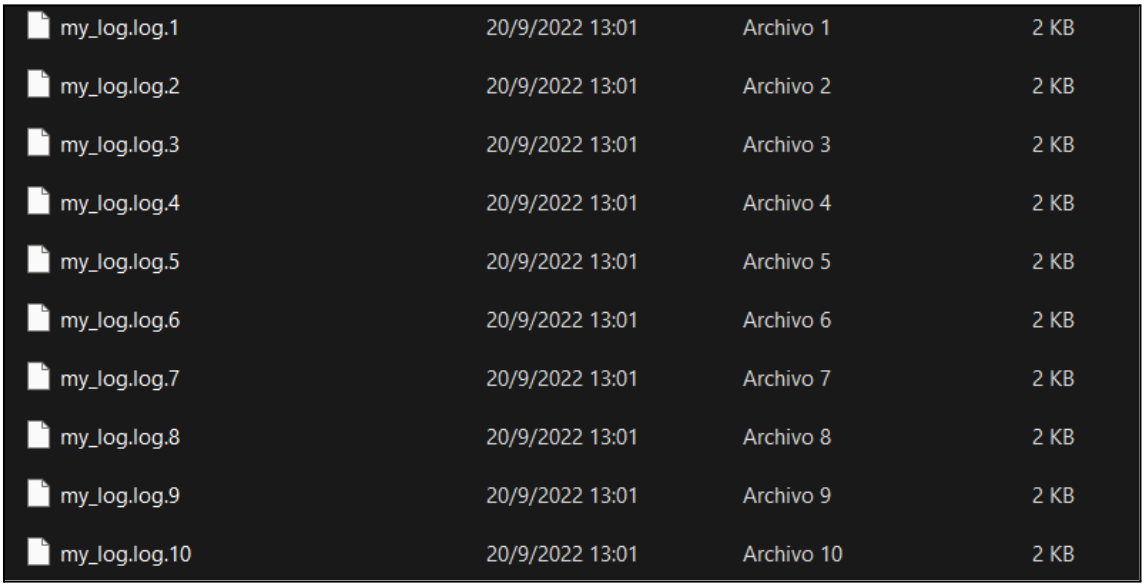
- maxBytes = 2000
- backupCount = 10

El valor **maxBytes** es el tamaño máximo de un archivo de registro. Cuando el archivo de registro está a punto de alcanzar maxBytes, ese

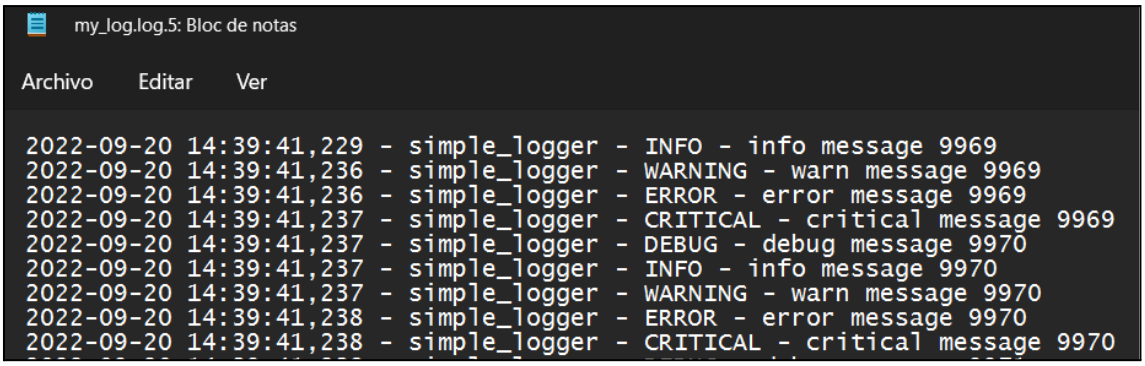
archivo se cierra y un nuevo archivo se abre silenciosamente para recibir nuevos mensajes de registro.

El valor **backupCount** establece nombres para los nuevos archivos de registro al rotar. Por ejemplo, cuando establecemos backupCount en cinco y un nombre de archivo de registro base de logging_file.log, obtendremos los archivos: logging_file.log, logging_file.log.1, y así sucesivamente, hasta logging_file.log.5.

Finalmente podemos visualizar los archivos de registro generados con la configuración establecida previamente.



Es importante destacar que cuando el script termina de escribir en el archivo número 10 (que es el valor backupCount establecido), comenzará a sobreescribir a partir del archivo 1 hasta que el for loop haya terminado.



Ejercicio Práctico 3

En este ejercicio vamos a utilizar la clase TimedRotatingFileHandler, que nos va a permitir separar nuestros logs en diferentes archivos de texto cada un tiempo determinado.

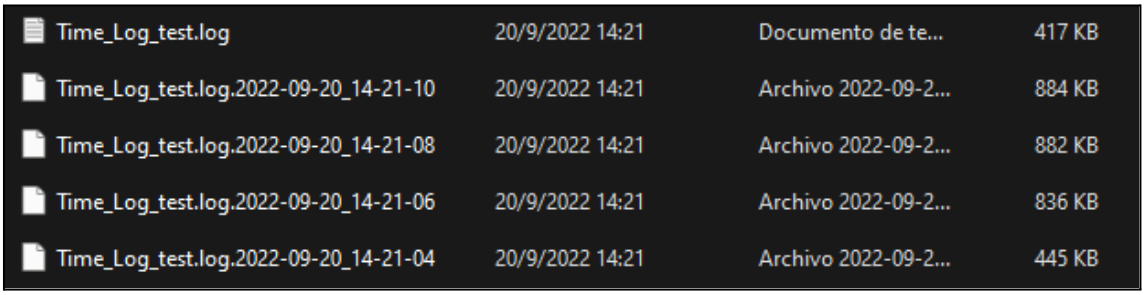
De forma análoga al ejercicio anterior, vamos a generar un for loop para crear los registros.

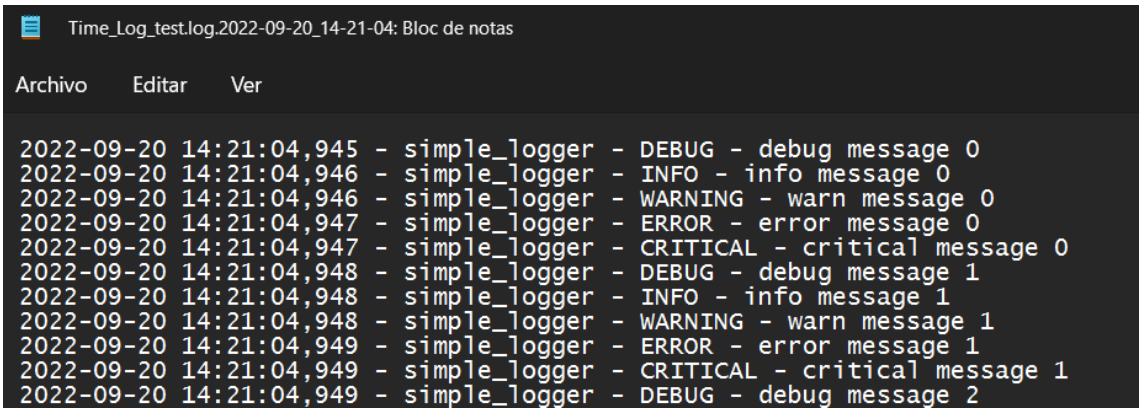
Configurando el handler con el parámetro When = S establecemos que la unidad para realizar el corte será del tipo segundos.

Interval = 2 significa dos segundos, y backupCount = 50 representa la cantidad máxima de archivos a generar.

```
import logging
from logging.handlers import TimedRotatingFileHandler

# Creamos el logger
logger = logging.getLogger('simple_logger')

# Establecemos el nivel de severidad
logger.setLevel(logging.DEBUG)

# Creamos un Time rotating file handler y seteamos el nivel de severidad en DEBUG
handler = TimedRotatingFileHandler('Time_Log_test.log', when='S', interval=2, backupCount=50)
handler.setLevel(logging.DEBUG)

# Creamos un objeto de formato
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# Agregamos el objeto formato al rotating file handler
handler.setFormatter(formatter)

# Agregamos el handler al logger
logger.addHandler(handler)

#Generamos los logs
for i in range(10000):

    logger.debug('debug message {}'.format(i))
    logger.info('info message {}'.format(i))
    logger.warning('warn message {}'.format(i))
    logger.error('error message {}'.format(i))
    logger.critical('critical message {}'.format(i))
```

Finalmente podemos visualizar los archivos de registro generados con la configuración establecida previamente.


Time_Log_test.log 20/9/2022 14:21 Documento de te... 417 KB
Time_Log_test.log.2022-09-20_14-21-10 20/9/2022 14:21 Archivo 2022-09-2... 884 KB
Time_Log_test.log.2022-09-20_14-21-08 20/9/2022 14:21 Archivo 2022-09-2... 882 KB
Time_Log_test.log.2022-09-20_14-21-06 20/9/2022 14:21 Archivo 2022-09-2... 836 KB
Time_Log_test.log.2022-09-20_14-21-04 20/9/2022 14:21 Archivo 2022-09-2... 445 KB


Time_Log_test.log.2022-09-20_14-21-04: Bloc de notas
Archivo Editar Ver
2022-09-20 14:21:04,945 - simple_logger - DEBUG - debug message 0
2022-09-20 14:21:04,946 - simple_logger - INFO - info message 0
2022-09-20 14:21:04,946 - simple_logger - WARNING - warn message 0
2022-09-20 14:21:04,947 - simple_logger - ERROR - error message 0
2022-09-20 14:21:04,947 - simple_logger - CRITICAL - critical message 0
2022-09-20 14:21:04,948 - simple_logger - DEBUG - debug message 1
2022-09-20 14:21:04,948 - simple_logger - INFO - info message 1
2022-09-20 14:21:04,948 - simple_logger - WARNING - warn message 1
2022-09-20 14:21:04,949 - simple_logger - ERROR - error message 1
2022-09-20 14:21:04,949 - simple_logger - CRITICAL - critical message 1
2022-09-20 14:21:04,949 - simple_logger - DEBUG - debug message 2

Cierre

A lo largo de este módulo continuamos trabajando con el módulo logging que viene por defecto en Python.

Aprendimos cómo configurarlo a través de un archivo y también incorporamos algunas buenas prácticas para tener en cuenta.

Finalmente revisamos diferentes casos prácticos donde continuamos aprendiendo sobre esta herramienta.

Referencias

Instalación de Logging para Python | Documentación de Python - 3.10.7. Recuperado de: https://docs.python.org/es/3/library/logging.html#logging.info

Logging in Python | Real Python. Recuperado de: https://realpython.com/python-logging/#using-handlers

https://www.datadoghq.com/blog/python-logging-best-practices/

tiimiit by