



📌 Guía: Configuración avanzada de Logging en la capa de datos

✨ Introducción

En este ejercicio vamos a extender la configuración del archivo `logger_base.py` para que el logging no solo envíe información a la consola, sino también a un archivo de log. Definiremos un formato de mensaje personalizado incluyendo la hora, nivel, archivo y línea.

🔥 Paso 2: Editar el archivo `logger_base.py`

Editamos el archivo `logger_base.py` con el siguiente contenido:

```
import logging as log

log.basicConfig(level=log.DEBUG,
                format='%(asctime)s: %(levelname)s [%(filename)s:%(lineno)s] %(message)s',
                datefmt='%I:%M:%S %p',
                handlers=[
                    log.FileHandler('capa_datos.log'),
```

```
        log.StreamHandler()
    ])

if __name__ == '__main__':
    log.debug('Mensaje a nivel debug')
    log.info('Mensaje a nivel info')
    log.warning('Mensaje a nivel de warning')
    log.error('Mensaje a nivel de error')
    log.critical('Mensaje a nivel critico')
```

✨ Explicación parte por parte:

◆ `import logging as log`

➡ Estamos **importando el módulo logging** de Python, pero lo renombramos como **log** para usarlo más corto en el resto del código.

Ejemplo: en vez de `logging.debug(...)` ahora solo escribimos `log.debug(...)`.

◆ `log.basicConfig(...)`

Esta función **configura la forma en que funciona el logging en toda la aplicación**. Aquí es donde hicimos la **configuración avanzada**.

Veamos cada parámetro:

🔑 `level=log.DEBUG`

✓ Define el nivel mínimo de los mensajes que se van a mostrar.

👉 Como usamos `DEBUG`, significa que **se mostrarán todos los mensajes**: `debug`, `info`, `warning`, `error` y `critical`.

🔍 Si hubiéramos puesto `WARNING`, solo mostraría `warning`, `error` y `critical` (ignorarías `debug` y `info`).

 `format='...'`

✓ Define cómo se verá el mensaje de log (el texto que aparece en consola o archivo).

La cadena:


```
'%(asctime)s: %(levelname)s [%(filename)s:%(lineno)s] %(message)s'
```

significa:

- `%(asctime)s`: La hora en que ocurrió el evento.
- `%(levelname)s`: El **nivel** del mensaje (DEBUG, INFO, etc.).
- `%(filename)s`: El **nombre del archivo Python** donde ocurrió el mensaje.
- `%(lineno)d`: El **número de línea** donde ocurrió el mensaje.
- `%(message)s`: El **mensaje de texto** que escribimos (ejemplo: "Mensaje a nivel debug").

✓ El formato resultará en algo como:

```
12:45:03 PM (DEBUG) [logger_base.py:15] Mensaje a nivel debug
```

 `datefmt='%I:%M:%S %p'`

✓ Define cómo mostrar la hora.

Esto:

```
'%I:%M:%S %p'
```

significa:

- `%I`: Hora (1-12)
- `%M`: Minutos
- `%S`: Segundos
- `%p`: AM o PM

✓ Así se mostrará la hora tipo "03:45:10 PM".

 `handlers=[...]`

✓ Aquí decimos a dónde queremos enviar los mensajes de log.

Los **handlers** son "destinos", es decir, hacia donde se va a mandar o guardar la información.

En este caso hay **dos destinos**:

➡ `log.FileHandler('capa_datos.log')`

👉 **Guarda los mensajes en un archivo llamado `capa_datos.log`.**

Cada vez que ejecutas el programa, los nuevos mensajes se agregan al final del archivo.

✅ Este archivo guarda **todo lo que aparece en consola, pero en texto plano**.

➡ `log.StreamHandler()`

👉 **Muestra los mensajes en la consola de Python o terminal.**

Este es el **comportamiento por defecto**, pero aquí lo estamos declarando explícitamente para que trabaje junto al archivo.

✅ **Ambos destinos reciben el mismo formato y niveles.**

◆ `if __name__ == '__main__':`

✔ Esta línea **evita que los mensajes se impriman automáticamente si importas este archivo desde otro módulo**.

✅ Solo si **ejecutas este archivo directamente**, se mostrarán los mensajes.

◆ Las llamadas a `log.debug(...)`, `log.info(...)`, etc.

Cada una de estas líneas:

```
log.debug('Mensaje a nivel debug')
log.info('Mensaje a nivel info')
log.warning('Mensaje a nivel de warning')
log.error('Mensaje a nivel de error')
log.critical('Mensaje a nivel critico')
```

- ✓ Genera un mensaje de log en el nivel correspondiente.
- ✓ Se mostrará en la consola y también se guardará en el archivo `capa_datos.log`.

Con el formato configurado, por ejemplo:

```
12:47:32 PM (WARNING) [logger_base.py:17] Mensaje a nivel warning
```



¿Qué hemos logrado con todo esto?

👉 Ahora los logs:

- ✓ Se muestran en consola y se guardan en archivo al mismo tiempo.
 - ✓ Tienen formato legible y detallado (hora, nivel, archivo, línea, mensaje).
 - ✓ El archivo `capa_datos.log` se actualiza cada vez que se ejecuta el programa.
-



Paso 3: Ejecutar el archivo

Ejecutamos `logger_base.py`.

- ✓ Al ejecutar, los mensajes se mostrarán en la consola y también se guardarán en el archivo `capa_datos.log`.
-



Paso 4: Verificar el archivo `capa_datos.log`

Abrimos el archivo `capa_datos.log` generado y confirmamos que contiene las mismas líneas que se imprimieron en la consola.



Conclusión

La configuración de logging ahora envía mensajes tanto a la consola como al archivo `capa_datos.log`. Los mensajes incluyen la hora, el nivel, el archivo y la línea de código donde se generó el mensaje.



¿Por qué es útil?

- ✓ Permite seguir errores o mensajes en producción sin tener la consola abierta.
 - ✓ Permite rastrear qué parte del código generó el mensaje (archivo y línea).
 - ✓ Útil para proyectos grandes donde necesitas auditar errores o registrar eventos.
-

✨ Sigue adelante con tu aprendizaje 🚀, ¡el esfuerzo vale la pena!

¡Saludos! 🙌

Ing. Marcela Gamiño e Ing. Ubaldo Acosta

Fundadores de [GlobalMentoring.com.mx](https://www.globalmentoring.com.mx)