Python Cheatsheet – `logging` Module

> **Curso:** Google IT Automation with Python **Tema:** Uso profesional de `logging`, handlers, niveles y buenas prácticas

¿Qué es `logging`?

`logging` es el módulo estándar de Python para **registrar información** durante la ejecución de un programa. Permite guardar mensajes de distintos niveles (DEBUG, INFO, WARNING, ERROR, CRITICAL) y enviarlos a distintos destinos: consola, archivos, email, syslog, etc.

Es una alternativa profesional a `print()`.

Niveles de Log

Nivel	Uso
`DEBUG`	Información detallada para debugging.
`INFO`	Confirma que las cosas funcionan como se espera.
`WARNING`	Algo no esperado ocurrió, pero el programa sigue.
`ERROR`	Error serio: una parte del programa falló.
`CRITICAL`	Error grave: el programa puede no seguir.

Configuración básica

```
import logging

logging.basicConfig(
    level=logging.DEBUG,                # Nivel mínimo a registrar
    format='%(asctime)s - %(levelname)s - %(message)s'
)

logging.debug("Mensaje de debug")
logging.info("Información normal")
logging.warning("Advertencia")
logging.error("Ocurrió un error")
logging.critical("Fallo crítico")
```

Estructura de un logger

`logging` se compone de 3 capas:

- **Logger** → genera mensajes
- **Handler** → define dónde van (archivo, consola, syslog, email...)
- **Formatter** → formato del mensaje

🗁 Logger personalizado

```
import logging

logger = logging.getLogger("mi_aplicacion")
logger.setLevel(logging.DEBUG)
```

📄 Handler: mandar logs a un archivo

```
file_handler = logging.FileHandler("app.log")
file_handler.setLevel(logging.INFO)

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %
(message)s')
file_handler.setFormatter(formatter)

logger.addHandler(file_handler)

logger.info("Inicio de la aplicación")
```

🖥 Handler: enviar logs a la consola

```
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.DEBUG)

console_handler.setFormatter(formatter)
logger.addHandler(console_handler)

logger.debug("Esto se imprime en consola")
```

🔁 Rotating logs (muy usado en producción)

```
from logging.handlers import RotatingFileHandler

rotating_handler = RotatingFileHandler(
    "app.log",
    maxBytes=2000000,    # 2 MB
```

```
    backupCount=5        # Mantener 5 archivos viejos
)
rotating_handler.setFormatter(formatter)

logger.addHandler(rotating_handler)
```

Syslog Handler (como en servidores Linux)

```
from logging.handlers import SysLogHandler

syslog_handler = SysLogHandler(address="/dev/log")
syslog_handler.setLevel(logging.WARNING)
syslog_handler.setFormatter(formatter)

logger.addHandler(syslog_handler)
```

Logging dentro de excepciones

```
try:
    1 / 0
except ZeroDivisionError:
    logging.exception("División por cero detectada")
```

`logging.exception()` captura automáticamente el traceback.

Buenas prácticas

✔ Usar `logging` en lugar de `print()` ✔ Configurar handlers en un módulo separado (`logging_config.py`) ✔ Usar diferentes niveles según la gravedad ✔ En libs: crear loggers con `__name__` ✔ En apps grandes: usar rotating logs ✔ Nunca mezclar `print()` y `logging` para registrar errores

---

Ejemplo profesional completo

```
import logging
from logging.handlers import RotatingFileHandler

logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
```

```
# Consola
console = logging.StreamHandler()
console.setLevel(logging.INFO)
console.setFormatter(formatter)

# Archivo con rotación
file = RotatingFileHandler("app.log", maxBytes=1000000, backupCount=3)
file.setLevel(logging.DEBUG)
file.setFormatter(formatter)

logger.addHandler(console)
logger.addHandler(file)

logger.info("Aplicación iniciada")
logger.debug("Debug interno")
```