Clase 012 — Logging

Parte: 0 — Prerrequisitos · Fuente: Python Tutorial logging HOWTO · The Pragmatic Programmer — "Programming by Coincidence". · Duración estimada: 60 min.

Clase 012 — Logging

> *Parte: 0 — Prerrequisitos · Fuente: Python Tutorial logging HOWTO · The Pragmatic Programmer — "Programming by Coincidence". · Duración estimada: 60 min.*

Objetivo

Que el alumno deje de usar print para debug y aprenda el módulo logging estándar: niveles (DEBUG/INFO/WARNING/ERROR/CRITICAL), handlers (consola, archivo), formatters, y configuración por módulo. Es la diferencia entre código que se debuggea reiniciando el notebook y código que se debuggea leyendo logs.

Resultados de aprendizaje

Al finalizar la clase, el alumno podrá:

1. Diferenciar los 5 niveles de logging y cuándo usar cada uno.
2. Configurar un logger con logging.basicConfig y entender por qué basicConfig solo funciona una vez.
3. Crear loggers por módulo con logging.getLogger(__name__).
4. Agregar handlers: uno a consola (INFO+), otro a archivo (DEBUG+).
5. Formatear logs con timestamp, módulo y nivel.

Temas

#	Tema	Por qué importa
1	print vs logging	print() es output; logging es observabilid
2	Niveles: DEBUG/INFO/WARNING/ERROR/C	Filtran qué se ve sin tocar código.
3	Logger jerárquico por módulo	getLogger(__name__) para herencia natural.
4	Handlers: consola, archivo, rotating	Mismo log → múltiples destinos.
5	Formatters	Timestamp + nivel + módulo + mensaje.
6	logging.basicConfig y sus límites	Solo afecta el primero; preferir config ex

Definiciones y características

Logger

: Punto de entrada para emitir logs. Se obtiene con logging.getLogger(__name__) — esto crea un logger nombrado por el módulo. Característica clave: los loggers son jerárquicos (separados por .); la config de root propaga a hijos.

Handler

: Define a dónde van los logs (consola, archivo, syslog, sentry…). Un logger puede tener N handlers. Cada handler tiene su propio nivel y formatter.

Formatter

: Define cómo se renderiza el log: '%(asctime)s [%(levelname)s] %(name)s: %(message)s'. Campos

comunes: asctime, levelname, name (logger), message, funcName, lineno, pathname.

Nivel (DEBUG/INFO/WARNING/ERROR/CRITICAL)

: Severidad ascendente. Filtran qué se emite. DEBUG: detalle interno; INFO: progreso normal; WARNING: algo raro pero no fatal; ERROR: operación falló; CRITICAL: sistema no puede continuar.

dictConfig

: Forma declarativa de configurar logging desde un dict (o YAML/JSON). Más mantenible que llamadas basicConfig/addHandler dispersas. Convención: una sola llamada en el entrypoint.

Dataset / recursos

Genera un log file de demo. Sin descarga.

Ejercicios

1. Reemplaza prints. Toma una función con 5 prints y conviértelos a logger con niveles apropiados.

2. Logger por módulo. Crea 2 archivos .py que cada uno usa getLogger(__name__). Configura el root logger una vez; verifica que ambos heredan.

3. Handler doble. Configura: consola = INFO+, archivo app.log = DEBUG+. Genera 5 logs de niveles distintos y verifica qué aparece en cada destino.

4. Formato con timestamp. Cambia el formato a '%(asctime)s [%(levelname)s] %(name)s: %(message)s'. Inspecciona output.

5. Logger en notebook. Pelea con basicConfig no recordando estado entre reinicios — usa dictConfig o force=True.

Homework verificable

Notebook + 2 módulos .py que importan y loguean. Un logging_config.py con dictConfig que define: consola (INFO+, formato corto) y app.log (DEBUG+, formato verbose con timestamp). El notebook ejecuta funciones que generan logs de distintos niveles desde ambos módulos. Adjunta el app.log resultante.

Criterio de aceptación: app.log contiene timestamp y módulo correcto en cada línea; consola filtra DEBUG.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
logging.basicConfig(...) no tiene efecto	basicConfig solo aplica si root no tiene h
Logs duplicados (cada mensaje aparece 2 ve	Configuraste el mismo handler dos veces (c
log.debug(f'valor: {expensive_call()}') si	El f-string se construye antes de pasar a
Mi logger emite en INFO pero quiero ver DE	El nivel está en handler o logger raíz. Fi
logging rompe en multiprocessing	Handlers no son fork-safe. Fix: en cada pr

Preguntas frecuentes

¿print o logging?

logging siempre en código que vivirá >1 día. print solo para REPL/scripts one-shot. Logging te da niveles, timestamps, módulo origen, múltiples destinos, integración con observabilidad.

¿Dónde configuro logging?

Una sola vez en el entrypoint (__main__, app.py, cli.py). Cada módulo solo hace log = logging.getLogger(__name__); nunca llama a basicConfig/addHandler desde un módulo importable.

¿Por qué getLogger(__name__)?

Crea un logger jerárquico nombrado por el módulo. Permite silenciar uno específico (logging.getLogger('mi_app.db').setLevel(WARNING)) sin tocar el resto. Pattern estándar.

¿Cómo formato un dict/object en el mensaje?

log.info('user=%s data=%s', user_id, data) (mejor que f-string por el lazy). Para JSON estructurado, usa python-json-logger o stdlib con custom formatter.

¿logging propaga al root logger?

Por default, sí — cada logger propaga al padre hasta root. Si configuras handlers en root y en hijos, verás el mensaje dos veces. Fix: logger.propagate = False en el hijo, o solo configura root.

Referencias

- Logging HOWTO
- Logging Cookbook
- logging.config — dictConfig

Siguiente clase

Clase 013 — Type hints y mypy

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
import logging
import tempfile
from pathlib import Path
from logging.config import dictConfig
```

Archivos complementarios

- notebook.ipynb