PROYECTO: Monitoreo acústico de niveles de ruido submarino y telemetría satelital para detección de eventos acústicos intensos en aguas de la Plataforma Continental Argentina

Dirección del proyecto: *Igor Prario*
Co-Dirección del proyecto: *Patricio Bos*

Rutinas de configuración y adquisición para placa de audio Behringer

Patricio Bos

División Acústica Submarina
Informe Técnico AS 3/25
Marzo 2025

Rutinas de configuración y adquisición para placa de audio Behringer.

Patricio Bos

RESUMEN

El presente informe técnico documenta el diseño e implementación de un módulo de software para configurar y controlar una placa de audio Behringer UMC204HD. Este instrumento forma parte del equipamiento de la Estación Autónoma Marítima para el Monitoreo de Ruido Ambiente (EAMMRA), desarrollado por la División Acústica Submarina de la Dirección de Investigación de la Armada (DIIV). El software desarrollado permite la adquisición y registro de señales acústicas analógicas provenientes de hidrófonos de manera automática o manual. Este trabajo es parte del Proyecto "Monitoreo acústico de niveles de ruido submarino y telemetría satelital para detección de eventos acústicos intensos en aguas de la Plataforma Continental Argentina", del programa PIDDEF del Ministerio de Defensa, vinculado al Proyecto "Localización e identificación de fuentes de ruido" de la Armada Argentina, llevado a cabo en la División Acústica Submarina de la Dirección de Investigación de la Armada (DIIV) y la Unidad Ejecutora de Investigación y Desarrollo Estratégicos para la Defensa (UNIDEF), dependiente de CONICET/MinDef.

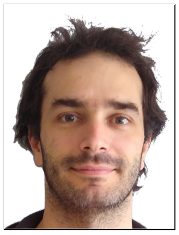
ABSTRACT

This technical report documents the design and implementation of a software module to configure and control a Behringer UMC204HD audio interface. This instrument is part of the equipment of the Autonomous Maritime Station for Ambient Noise Monitoring (EAMMRA), developed by the Submarine Acoustics Division of the Navy Research Directorate (DIIV). The developed software allows for the automatic or manual acquisition and recording of analog acoustic signals from hydrophones. This work is part of the Project "Acoustic monitoring of underwater noise levels and satellite telemetry for detecting intense acoustic events in waters of the Argentine Continental Shelf", from the PIDDEF program of the Ministry of Defense, linked to the Project "Localization and identification of noise sources" of the Argentine Navy, carried out in the Underwater Sound Division of the Argentinian Navy Research Office (DIIV) and the Strategic Research and Development Unit for Defense (UNIDEF), dependent on CONICET/MinDef.

Este trabajo es parte del Proyecto
“Monitoreo acústico de niveles de ruido submarino y telemetría satelital
para detección de eventos acústicos intensos en aguas de la
Plataforma Continental Argentina”,
del programa PIDDEF del Ministerio de Defensa,
vinculado al Proyecto “Localización e identificación de fuentes de ruido”
de la Armada Argentina,
que se lleva a cabo en la División Acústica Submarina
de la Dirección de Investigación de la Armada (DIIV) y
la Unidad Ejecutora de Investigación y Desarrollo Estratégicos
para la Defensa (UNIDEF), dependiente de CONICET/MinDef.

AGRADECIMIENTOS

Los autores desean expresar su agradecimiento a:



Ing. Rui Marques Rojo, investigador RPIDFA que forma parte del equipo de trabajo del Departamento de Propagación Acústica de la Dirección de Investigación de la Armada, por el montaje y puesta a punto de la plataforma Gitea con servicios para el desarrollo de código, que fuera extensamente utilizada en la realización del trabajo que aquí se documenta. Asimismo, se deja constancia de sus valiosas sugerencias para la implementación de código en lenguaje Python.

ÍNDICE

GLOSARIO DE SIGLAS

API	Application Program Interface.
ARA	Armada Argentina.
ASCII	American Standard Code for Information Interchange.
AWG	American Wire Gauge.
CD	Continuous Delivery.
CI	Continuous Integration.
CSV	Comma Separated Values.
DDR	Double Data Rate.
EAMMRA	Estación Autónoma Marítima para el Monitoreo de Ruido Ambiente.
ETX	End of TeXt.
GB	Giga Byte.
GNU	Gnu is Not Unix.
IDE	Integrated Development Environment.
LTS	Long Term Support.
LAN	Local Area Network.
MIL-DTL	Military Detail Specification.
NMEA	National Marine Electronics Association.
PCM	Pulse Code Modulation.
PEP	Python Enhancement Proposal.
RIFF	Resource Interchange File Format.
SSH	Secure SHell.
URL	Uniform Resource Locators.
VPN	Virtual Private Network.
WAV	WAVeform audio file.
YAML	YAML Ain't Markup Language.

LISTA DE FIGURAS

LISTA DE TABLAS

I. INTRODUCCIÓN

El objetivo de este informe técnico es documentar el diseño e implementación de un módulo de software para configurar y controlar una placa de audio Behringer UMC204HD. Este dispositivo forma parte del equipamiento e instrumental de la Estación Autónoma Marítima para el Monitoreo de Ruido Ambiente (EAMMRA) que se desarrolla en la División Acústica Submarina de la Dirección de Investigación de la Armada (DIIV).

La estación EAMMRA es una boya de superficie de diseño específico para la medición de Ruido Ambiente submarino cuya concepción, diseño y fabricación se encuentran documentados en respectivos informes técnicos [Bos *et al.*, 2016], [Ezcurra *et al.*, 2019] y [Cinquini *et al.*, 2019].

El control de la estación en general, y la interacción con la placa en particular, se realizan con una computadora de grado industrial Kaise KBOX-S J6412. Esta computadora pertenece a la categoría de *fanless embedded systems*, que son sistemas de misión específica sin partes móviles y es especialmente adecuada para aplicaciones de funcionamiento autónomo. Cuenta con un procesador Intel Celeron J6412 de cuatro núcleos y memoria DDR4 de 16 GB [Kaise, 2025], lo que le otorga al sistema una razonable capacidad de cómputo para el bajo consumo de energía que requiere la aplicación. En la figura 1 se muestra una vista frontal y trasera de la computadora y se pueden observar las interfaces y conectores disponibles.

FIG. 1. Vista frontal y trasera de la computadora Kaise KBOX-S J6412.

La computadora de EAMMRA corre un sistema operativo de propósitos generales, Ubuntu Server 22.04 LTS. Ubuntu Server es una distribución libre y gratuita de GNU/Linux que no requiere la compra de licencias para su uso. Este sistema operativo es reconocido por su estabilidad y seguridad, atributos esenciales para aplicaciones críticas en sistemas desatendidos como EAMMRA. Por diseño, Ubuntu Server permite una extensa personalización, con la posibilidad de realizar una instalación con los mínimos componentes necesarios, lo que permite optimizar el rendimiento general del sistema.

El módulo de software para configurar y controlar la placa de audio se implementa en Python, un lenguaje de programación interpretado y multiparadigma de alto nivel que viene integrado por defecto en las distribuciones de Ubuntu. En particular, para este módulo se utiliza Python3 que es la versión del lenguaje recomendada para proyectos nuevos, que no requieren retrocompatibilidad con componentes previos que hayan sido desarrollados en Python2.

I.1. Herramientas de desarrollo

Las herramientas de desarrollo de software que se utilizaron para la implementación del módulo de configuración y control permiten mejorar la calidad del código. Se entiende como código de calidad, aquel que hace lo que se supone que debe hacer, no contienen defectos ni problemas y es fácil de extender con nuevas características.

Las guías de estilo en general se utilizan para definir una forma consistente de escribir código. Si bien el estilo de codificación puede parecer una cuestión de forma, que no afecta el funcionamiento lógico del código, algunas decisiones de estilo pueden evitar errores lógicos frecuentes. Las guías definen convenciones que ayudan a mantener el código fácil de leer, mantener y extender.

Se emplearon herramientas específicas para lenguaje Python, que tienen gran difusión y aceptación en la comunidad de desarrolladores conocidas como *linters*. Estas herramientas permiten analizar el código y verificar el cumplimiento de un conjunto de reglas de mejores prácticas y buscan problemas, tanto de no conformidad con un estilo como errores de lógica. Los defectos de lógica que se buscan son errores de código, código con resultados potencialmente no deseado y patrones de código peligroso. El uso de estas herramientas mejora la legibilidad, la mantenibilidad y la escalabilidad y hace que el código sea menos propenso a errores.

I.1.a. Guías de estilo para código Python - PEP-8 y PEP-257

Para la escritura del código Python del módulo de configuración y control se adopta la guía de estilo *Python Enhancement Proposal* 8 (PEP-8). Su objetivo principal es mejorar la legibilidad y coherencia del código Python y tiene amplia aceptación dentro de la comunidad de desarrolladores de este lenguaje. La guía fue escrita en 2001 por Guido van Rossum, Barry Warsaw y Nick Coghlan, basándose en las mejores prácticas existentes y las recomendaciones de la comunidad [van Rossum *et al.*, 2001]. Esta guía fomenta buenas prácticas y técnicas que pueden ayudar a evitar errores comunes.

La guía cubre varios aspectos de la codificación, que incluyen:

- Formato del código: indentación, uso de espacios en blanco, longitud de las líneas, etc.
- Convenciones de nomenclatura: cómo nombrar variables, funciones, clases, módulos, etc.
- Principios de codificación: recomendaciones sobre cómo escribir expresiones y declaraciones de manera clara.
- Comentarios: cuándo y cómo usar comentarios para mejorar la legibilidad del código.

Asimismo, se utiliza la convención *Python Enhancement Proposal* 257, (PEP-257). Esta convención define el formato y estilo para la documentación, también llamada *Python's docstrings* que aplica a módulos, clases, funciones y métodos [Goodger y van Rossum, 2001]. Adicionalmente, si los *docstring* se escriben en forma consistente, existen herramientas capaces de generar documentación directamente del código. Esto permite mantener más fácilmente la documentación actualizada dentro del mismo código.

Existen herramientas de desarrollo como pycodestyle y pydocstyle que verifican el estilo y ayudan a asegurar que el código cumpla con las convenciones de PEP-8 y PEP-257, respectivamente. Estas herramientas pertenecen a la categoría de linters de estilo y en algunos casos vienen integrados dentro de otros linters como Flake8 o Pylama.

I.1.b. Style guide enforcement - Flake8

Flake8 es una herramienta que permite imponer y hacer cumplir las directivas de la guía de estilo PEP-8 y pertenece a la familia de programas tipo *lint* que trabajan analizando el código fuente para verificar el cumplimiento de un conjunto definido de reglas. Esta herramienta puede señalar errores de programación, *bugs*, errores de estilo y construcciones sospechosas. Es altamente configurable, y permite a los desarrolladores ajustar las reglas y la severidad de las advertencias según las necesidades específicas de su proyecto [Cordasco, 2016].

Para instalar flake8 para la versión por defecto de Python:

```
python -m pip install flake8
```

Para utilizar flake8, se deben ejecutar los siguientes comando desde una terminal interactiva:

```
flake8 path/to/code/to/check.py
# or
flake8 path/to/code/
```

A su vez, se puede seleccionar una regla específica para ejecutar o ignorar:

```
flake8 --select E123,W503 path/to/code/
# or
flake8 --extend-ignore E203,W234 path/to/code/
```

El listado completo de códigos de error y su significado puede encontrarse en el sitio web oficial de la herramienta, `http://flake8.pycqa.org/en/latest/user/error-codes.html`.

I.1.c. Analizador estático - Pylint

Pylint es una herramienta de análisis estático de código para Python que busca errores de programación, ayuda a hacer cumplir un estándar de codificación y busca "malos olores" en el código (*code smells*). Utiliza diferentes técnicas para analizar el código fuente y puede identificar problemas o patrones de codificación problemáticos que podrían llevar a errores o a un código difícil de mantener o leer [Logilab y contributors to Pylint, 2024].

Las características principales de Pylint incluyen:

- Chequeo de errores: puede detectar errores que podrían hacer que el código falle en tiempo de ejecución, como llamadas a funciones no definidas, uso de variables antes de su definición, etc.
- Estándares de codificación: Pylint puede asegurar de que el código siga un estándar de codificación particular, como PEP-8.
- Refactorización de código: sugiere lugares donde el código podría ser refactorizado para mejorar la legibilidad o la eficiencia.
- Detección de código duplicado: puede identificar bloques de código duplicados que podrían ser simplificados o extraídos en una función común.
- Chequeo de tipos: aunque Python es un lenguaje dinámicamente tipado, Pylint puede realizar algunas comprobaciones de tipos para identificar posibles problemas.

Para instalar pylint para la versión por defecto de Python:

```
python –m pip install pylint
```

Para utilizar pylint, se debe ejecutar el siguiente comando:

```
pylint [options] modules_or_packages
```

Pylint agrega un prefijo a cada una de las áreas problemáticas con una R, C, W, E o F, que significan:

- **R**efactorizar por una violación de la métrica de "buena práctica".
- **C**onvención por violación del estándar de codificación.
- **W**arning (Advertencia) por problemas estilísticos o problemas de programación menores.
- **E**rror por problemas importantes de programación (es decir, muy probablemente un bug).
- **F**atal por errores que impidieron el procesamiento adicional.

La lista completa de mensajes y su significado puede encontrarse en el sitio web oficial de la herramienta, donde se encuentran agrupados por categoría o área problemática. La url es: `https://pylint.pycqa.org/en/latest/user_guide/messages/messages_overview.html`.

I.1.d. Formato automático - Black

Black es una herramienta de formateo de código para Python conocida por su enfoque en la simplicidad y la uniformidad. A menudo se le llama "el formateador de código sin compromisos" debido a su filosofía de tener una sola forma estandarizada y automatizada de formatear el código Python. Esto contrasta con otras herramientas de formateo que pueden permitir una mayor configuración o variaciones en el estilo de codificación [Łukasz Langa y contributors to Black, 2018].

Algunas características clave de Black incluyen:

- Automatización: Black reformatea todo el archivo de código con solo un comando, sin necesidad de ajustes manuales.
- Consistencia: aplica un estilo consistente en todos los proyectos de Python al seguir un conjunto de reglas predefinido, lo que ayuda a mejorar la legibilidad y reducir el tiempo dedicado a discutir sobre estilos de codificación en revisiones de código.
- Integración fácil: puede integrarse fácilmente con editores de texto y entornos de desarrollo integrados (IDEs), así como con sistemas de integración continua/entrega continua (CI/CD).
- Seguridad: está diseñado para realizar cambios en el código que no afecten su comportamiento, lo que lo hace seguro para usar en proyectos grandes y complejos.

Al tener el código un formato unificado entre los distintos desarrolladores, las revisiones de código, especialmente bajo control de versiones, se pueden hacer más rápidamente debido a que los *diffs* entre versiones son lo más pequeños posible. Black es un formateador de código PEP-8 compatible.

Para instalar Black, se debe ejecutar el siguiente comando en una terminal:

```
python pip install black
```

Para utilizar Black, se debe ejecutar el siguiente comando:

```
black modules_or_packages
```

I.1.e. Entorno de Desarrollo Integrado - Visual Studio Code

Visual Studio Code, disponible en `https://code.visualstudio.com/`, es un editor de código fuente de código abierto que soporta múltiples lenguajes de programación. Destaca por su flexibilidad y capacidad de personalización. Permite integrar extensiones que amplían sus funcionalidades. Entre sus características se encuentran el soporte para depuración integrada, control de versiones con Git y herramientas de autocompletado. También ofrece una interfaz de usuario intuitiva y un sistema de gestión de proyectos eficiente. VSCode es compatible con Windows, macOS y GNU/Linux.

Existen múltiples extensiones disponibles para Python. Entre las más destacadas se encuentran el *plugin* `Python`, que proporciona soporte para depuración, ejecución de código, y autocompletado de código Python, y `Pylance`, que ofrece características adicionales como análisis estático de código y autocompletado inteligente.

La extensión Remote - SSH de Visual Studio Code permite utilizar cualquier máquina remota con un servidor SSH como entorno de desarrollo. Esto facilita el desarrollo y la resolución de problemas en una amplia variedad de situaciones. Mediante esta extensión, es posible desarrollar en el mismo sistema operativo al que se desplegará el software o emplear hardware más grande, rápido o especializado que el de la máquina local.

Una de las ventajas de esta extensión es la capacidad de cambiar rápidamente entre diferentes entornos de desarrollo remotos, lo que permite realizar actualizaciones de manera segura sin riesgo de afectar la máquina local. Además, proporciona acceso a un entorno de desarrollo ya existente desde varias máquinas o ubicaciones, lo que aumenta la flexibilidad del proceso de desarrollo.

Esta extensión resulta útil también para depurar aplicaciones que se ejecutan en otros lugares. No es necesario tener el código fuente en la máquina local para aprovechar estas ventajas, ya que la extensión ejecuta comandos y otras extensiones directamente en la máquina remota. Es posible abrir cualquier carpeta en la máquina remota y trabajar con ella de la misma forma que si estuviera en la máquina local.

I.1.f. Pytest

Pytest [Krekel *et al.*, 2025b] es un marco de pruebas o *framework* para Python que facilita la creación, ejecución y gestión de pruebas automatizadas. Su diseño simple y flexible lo convierte en una herramienta popular tanto para desarrolladores novatos como para expertos. A través de una sintaxis intuitiva, pytest permite escribir pruebas concisas y efectivas, y ofrece potentes herramientas para la depuración de errores, la introspección de aserciones y la ejecución de pruebas en múltiples configuraciones. Es utilizado principalmente para pruebas unitarias, aunque también puede aplicarse a pruebas de integración y funcionales. pytest sigue un enfoque basado en convenciones, lo que significa que por defecto, descubre y ejecuta las pruebas sin necesidad de una configuración extensa.

Entre las características más destacadas de pytest se incluyen su sistema de *fixtures*, que permite preparar datos y recursos reutilizables para las pruebas, y su potente mecanismo de aserciones, que ofrece introspección detallada de los errores durante los fallos de prueba. Además, pytest soporta pruebas parametrizadas, la ejecución de pruebas en paralelo mediante *plugins*, y una fácil integración con herramientas de cobertura de código y análisis estático. Los resultados de las pruebas se presentan de forma clara y legible, y permiten que los desarrolladores comprendan rápidamente los problemas detectados. También es compatible con otras bibliotecas de pruebas como `unittest` y `nose`, lo que facilita la transición entre diferentes marcos de pruebas.

En cuanto a los requerimientos, pytest necesita Python 3.8 o superior para su funcionamiento, y puede instalarse fácilmente utilizando el administrador de paquetes `pip`. Para instalarlo, basta con ejecutar el siguiente comando:

```
pip install –U pytest
```

Una vez instalado, pytest se ejecuta desde la línea de comandos con el comando `pytest`, lo que hará que busque y ejecute todas las pruebas definidas en los archivos que sigan la convención de nombres `test_*.py` o `*_test.py`. Además, `pytest` permite la configuración avanzada mediante archivos de configuración y opciones de línea de comandos, lo que otorga una gran flexibilidad en proyectos de gran escala. La documentación oficial está disponible en [Krekel *et al.*, 2025a].

II. INTERFAZ DE AUDIO BEHRINGER UMC204HD

La Behringer UMC204HD es una interfaz de audio USB que ofrece una conversión de señal precisa y confiable, [Behringer, 2019]. Se utiliza en la grabación y reproducción de señales de audio analógicas y digitales. Es adecuada para la adquisición de señales de hidrófonos en aplicaciones científicas como EAMMRA. En la figura 2 se puede observar una vista frontal y trasera de la interfaz.

FIG. 2. Vista frontal y trasera de la interface de audio Behringer UMC204HD.

La UMC204HD tiene dos entradas XLR/TRS combo para micrófonos o instrumentos. También cuenta con dos salidas balanceadas TRS 1/4” para una señal limpia. La interfaz soporta dos canales de entrada y salida simultáneos. Cada canal tiene control de ganancia independiente. Incluye salidas de auriculares con control de volumen y un monitor de mezcla directa para la monitorización sin latencia.

La interfaz tiene una resolución de 24 bits y soporta hasta 192 kHz. La relación señal/ruido supera los 100 dB, lo que reduce el ruido en la señal. Estos valores permiten una captura precisa de señales, como las de hidrófonos. La interfaz es compatible con varias plataformas de software, lo que la hace versátil.

Con Python y PyAudio, se puede controlar la interfaz para la adquisición de datos en tiempo real. PyAudio permite configurar la tasa de muestreo, los canales y otros parámetros. Esto facilita la integración de la interfaz en sistemas de adquisición y análisis de señales.

III. DISEÑO E IMPLEMENTACIÓN

III.1. Descripción general del módulo

El módulo `BehringerHandler` fue desarrollado en Python con el objetivo de controlar una interfaz de audio USB marca Behringer de forma programática. Permite su inicialización, apertura de flujos de entrada, grabación asincrónica y gestión completa de recursos. Está pensado para ser utilizado en sistemas de adquisición acústica en tiempo real, y su diseño modular facilita su integración en arquitecturas mayores, como estaciones de monitoreo o sistemas embebidos.

El funcionamiento interno del módulo se apoya en varias bibliotecas estándar y externas. Utiliza `pyaudio` para el acceso de bajo nivel al hardware de audio, `wave` para la escritura de los datos capturados en formato WAV, `threading` para realizar la grabación en paralelo al hilo principal y `queue` para sincronizar los datos entre procesos. Además, se emplean `logging` para la trazabilidad de eventos, `datetime` para la generación de nombres de archivo basados en marca temporal y `os` para la gestión del entorno de ejecución y rutas de archivo. Esta combinación de módulos permite una operación robusta, escalable y fácilmente depurable.

La biblioteca `logging` de Python se utiliza para generar registros estructurados durante la ejecución del código, y permite monitorear el estado del sistema, detectar errores y realizar trazabilidad de eventos. En el módulo `BehringerHandler`, se implementa un sistema de log dual que escribe tanto en consola como en un archivo de texto (`behringer_handler.log`). Este sistema utiliza formatos personalizados con marcas de tiempo y niveles de severidad (INFO, WARNING, ERROR), lo que facilita el diagnóstico en tiempo real y la revisión posterior del funcionamiento del sistema. La inclusión de `logging` mejora la mantenibilidad y confiabilidad del software, especialmente en entornos donde no hay interacción directa con el usuario.

La biblioteca `PyAudio` proporciona una interfaz en Python para interactuar con PortAudio, una biblioteca de código abierto para la captura y reproducción de audio en tiempo real. Este paquete permite abrir flujos de entrada o salida de audio con parámetros configurables como tasa de muestreo, formato, número de canales y tamaño de buffer. En el caso de `BehringerHandler`, `PyAudio` es utilizado para detectar dispositivos de audio disponibles, abrir un flujo de entrada en formato `paInt24` y capturar señales a alta resolución (192 kHz, 2 canales). Gracias a su capacidad para operar en modo no bloqueante mediante callbacks, `PyAudio` permite implementar grabaciones asincrónicas eficientes, fundamentales para aplicaciones de adquisición continua o en tiempo real.

La biblioteca estándar `wave` de Python permite la manipulación de archivos de audio en formato WAV (RIFF). Proporciona una interfaz sencilla para la escritura y lectura de audio en formato PCM (Pulse Code Modulation), y permite definir parámetros como la cantidad de canales, la tasa de muestreo y la resolución en bits. En el módulo `BehringerHandler`, `wave` se utiliza para almacenar las grabaciones capturadas por el stream de PyAudio en archivos de alta calidad de 24 bits. La escritura se realiza en tiempo real desde un hilo separado, para asegurar la integridad de los datos y su disponibilidad inmediata en disco para posteriores análisis.

La biblioteca `threading` proporciona soporte para la ejecución concurrente de código mediante la creación y gestión de hilos de ejecución. En `BehringerHandler`, se utiliza para realizar la escritura de audio en disco de forma asincrónica, separando la captura en tiempo real del procesamiento de los datos. Esto permite que el flujo principal de adquisición no se vea interrumpido por operaciones de entrada/salida, lo que mejora el rendimiento y evita pérdidas de datos. El uso de `threading` resulta esencial para garantizar grabaciones continuas y confiables, especialmente en sistemas que requieren alta tasa de muestreo o baja latencia.

La biblioteca `queue` proporciona una estructura de datos segura para comunicación entre hilos en entornos multithreading. En el módulo `BehringerHandler`, se utiliza una instancia de `Queue` para almacenar de forma temporal los bloques de audio recibidos por el callback del stream de PyAudio. Esta cola permite desacoplar la adquisición de datos (en tiempo real) de la escritura en disco, evitando pérdidas o bloqueos por operaciones lentas de entrada/salida. El uso de `queue` garantiza una sincronización segura y eficiente entre el hilo de captura y el hilo de grabación.

La biblioteca estándar `datetime` permite manipular fechas y horas de manera precisa y flexible. En el módulo `BehringerHandler`, se utiliza para generar marcas de tiempo en el momento de iniciar una grabación, lo que permite construir nombres de archivo únicos con el formato `YYYYMMDD_HHMMSS`. Esta práctica facilita la organización de las grabaciones, evita sobreescrituras accidentales y proporciona una trazabilidad temporal inmediata. El uso de `datetime` también permite registrar eventos en los logs con exactitud temporal.

La biblioteca estándar `os` proporciona funciones para interactuar con el sistema operativo de forma portable. En el módulo `BehringerHandler`, se utiliza para obtener la ruta absoluta del script en ejecución, crear subdirectorios (como `recordings` para guardar archivos de audio) y construir rutas de forma segura e independiente del sistema operativo. El uso de `os` es esencial para garantizar la correcta gestión de archivos y configuraciones en distintos entornos de ejecución.

III.2. Dependencias y gestión del entorno

La lista completa de dependencias necesarias para ejecutar el módulo `BehringerHandler` se encuentra documentada en el archivo `requirements.txt`. Este archivo fue generado utilizando el comando `pip freeze >requirements.txt` desde un entorno virtual de Python configurado específicamente para este proyecto. La utilización de entornos virtuales permite aislar las versiones de los paquetes, asegurando la reproducibilidad del entorno de ejecución.

Para reconstruir el entorno en una nueva máquina o entorno limpio, se recomienda crear un entorno virtual (por ejemplo, mediante `python -m venv venv`) y luego instalar todas las dependencias mediante el comando `pip install -r requirements.txt`. Esto garantiza que las versiones exactas utilizadas durante el desarrollo serán replicadas, y permite reducir los errores por incompatibilidades entre bibliotecas.

A continuación, se presenta la lista de dependencias incluidas en `requirements.txt`:

```
contourpy==1.3.1
cycler==0.12.1
fonttools==4.56.0
iniconfig==2.1.0
kiwisolver==1.4.8
matplotlib==3.10.1
numpy==2.2.3
packaging==24.2
pillow==11.1.0
pluggy==1.5.0
PyAudio==0.2.14
pyparsing==3.2.1
pytest==8.3.5
python-dateutil==2.9.0.post0
schedule==1.2.2
six==1.17.0
```

III.3. Arquitectura del módulo

El diseño del módulo `BehringerHandler` está centrado en encapsular toda la funcionalidad de adquisición de audio a través de una interfaz USB Behringer, utilizando una arquitectura modular y extensible. La clase actúa como fachada para todas las operaciones, ofreciendo una interfaz clara basada en un ciclo de vida típico: inicialización, grabación y liberación de recursos.

Este ciclo se articula mediante un conjunto de métodos públicos que definen el flujo de uso:

1. `init()` inicializa la interfaz PyAudio y detecta automáticamente el dispositivo Behringer conectado por USB.
2. `record(duration)` lanza una grabación asincrónica por el tiempo especificado. Internamente, abre el stream de entrada (`open()`), inicia un hilo de escritura en disco (`_write_audio()`) y activa un callback (`_callback()`) que llena una cola de datos.
3. `stop_recording()` sincroniza el final de la grabación, deteniendo el hilo y limpiando los buffers.
4. `deinit()` libera todos los recursos y deja el módulo en estado reutilizable.

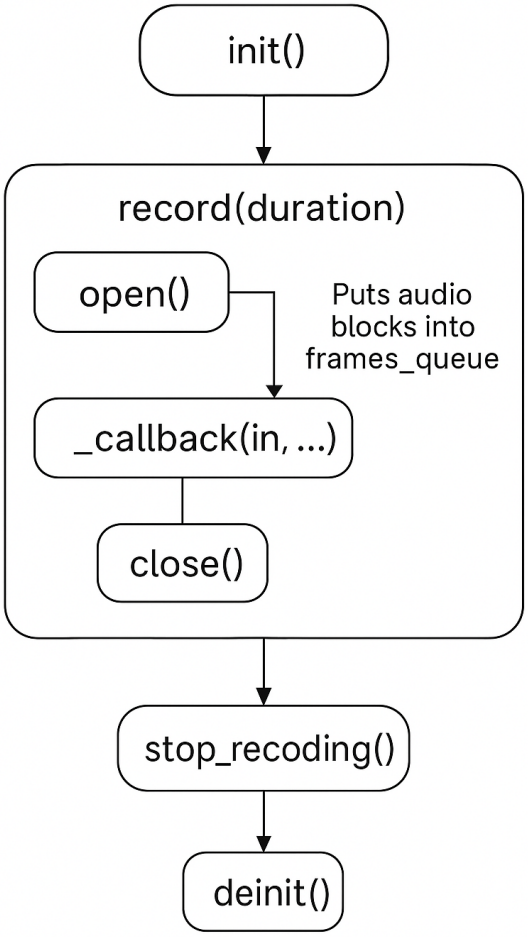
Desde el punto de vista del usuario, el uso del módulo sigue una secuencia lógica simple:

```
from behringer_handler import BehringerHandler

handler = BehringerHandler()
handler.init()
handler.record(duration=10)  # graba 10 segundos de audio
handler.stop_recording()
handler.deinit()
```

Durante la ejecución, los datos son adquiridos por el stream de PyAudio y colocados en una cola mediante el callback. En paralelo, un hilo escribe esos datos en un archivo WAV. Todos los eventos relevantes —como el inicio o cierre del stream, errores o advertencias— se registran mediante el sistema de `logging`, tanto en consola como en un archivo de log.

Este diseño permite desacoplar la adquisición en tiempo real del almacenamiento en disco, reduciendo riesgos de pérdida de datos y manteniendo la modularidad del sistema. Además, garantiza una interfaz coherente con otros módulos de adquisición definidos dentro del sistema EAMRRA.



III.4. Descripción de la clase y sus métodos

`BehringerHandler` extiende de `BaseHandler` e implementa los métodos clave para el manejo del dispositivo. La utilización de una clase base común responde a una estrategia de diseño orientada a la modularidad y escalabilidad del sistema EAMRRA, donde múltiples módulos de adquisición o control comparten una estructura funcional similar.

La clase `BaseHandler` actúa como plantilla abstracta, y define una interfaz mínima que obliga a los módulos derivados a implementar ciertos métodos esenciales, como `init`, `deinit`, `open` y `close`. Esta decisión asegura una simetría en la arquitectura del sistema, facilita el mantenimiento y permite la integración homogénea de nuevos módulos en el futuro. Además, permite aplicar patrones de programación como la inversión de dependencias, delegando a cada módulo el detalle de implementación mientras se conserva una estructura común y predecible.

A continuación se describen los métodos principales de la clase `BehringerHandler` con un detalle sobre los parámetros que reciben, los atributos que modifican, una descripción sintética de las acciones que realizan y finalmente, qué errores controlan, en cada caso.

Atributos principales definidos en `__init__`

En el constructor se inicializan los siguientes atributos:

- `audio_interface`: objeto de PyAudio encargado de la comunicación con el hardware.
- `device_index`: índice del dispositivo de audio identificado como Behringer.
- `stream`: flujo de entrada de audio (inicialmente `None`).
- `is_recording`: bandera booleana que indica si una grabación está en curso.
- `recording_thread`: hilo que ejecuta la escritura del audio en segundo plano.
- `start_time`, `duration`: variables para controlar el tiempo de grabación.
- `frames_queue`: cola de datos compartida entre el stream y el hilo de escritura.
- `logger`: objeto de `logging` configurado para registrar eventos en consola y archivo.
- `log_file`, `output_path`: rutas de archivo utilizadas durante la ejecución.

`init()`

- **Parámetros:** ninguno.
- **Atributos modificados:** `audio_interface`, `device_index`.
- **Acciones:** escanea los dispositivos de audio disponibles mediante `PyAudio` y selecciona automáticamente el que contiene “Behringer” o “USB” en su nombre, y tiene canales de entrada.
- **Control de errores:** si ya fue inicializado, emite un log y no repite el proceso. Si no encuentra el dispositivo, registra una advertencia. Si ocurre una excepción, se registra el error y se garantiza la liberación del objeto `PyAudio`.

`open()`

- **Parámetros:** ninguno.
- **Atributos modificados:** `stream`.
- **Acciones:** abre un flujo de entrada utilizando el índice del dispositivo detectado. Establece el formato en 24 bits, dos canales, 192 kHz y un tamaño de buffer de 8192 muestras.
- **Control de errores:** si el dispositivo no fue inicializado, se emite una advertencia. Si ocurre una excepción al abrir el flujo, se registra el error y el atributo `stream` se deja en `None`.

`record(duration)`

- **Parámetros:** `duration` (float), duración de la grabación en segundos.
- **Atributos modificados:** `is_recording`, `start_time`, `duration`, `recording_thread`, `output_path`.
- **Acciones:** crea el directorio `recordings` si no existe, genera un nombre de archivo único basado en fecha y hora, limpia la cola de audio, llama a `open()` y lanza un hilo que ejecuta `_write_audio()`.
- **Control de errores:** si el dispositivo no está inicializado, emite una advertencia y no comienza la grabación.

`_write_audio()`

- **Parámetros:** ninguno (método privado llamado por un hilo).
- **Atributos modificados:** ninguno directamente; opera sobre `frames_queue`.
- **Acciones:** crea el archivo WAV, configura los parámetros del encabezado y escribe bloques de audio tomados desde `frames_queue` hasta que finalice la grabación o la cola se vacíe.
- **Control de errores:** utiliza tiempo de espera en la cola para evitar bloqueos, registra la finalización y llama a `close()` al terminar.

`stop_recording()`

- **Parámetros:** ninguno.
- **Atributos modificados:** `is_recording`, `recording_thread`.
- **Acciones:** cambia el estado de grabación, espera a que el hilo finalice (join) y limpia referencias.
- **Control de errores:** verifica que el hilo exista y esté activo antes de intentar finalizarlo. Registra eventos relevantes.

`close()`

- **Parámetros:** ninguno.
- **Atributos modificados:** `stream`.
- **Acciones:** detiene y cierra el flujo de audio si está activo, y libera la referencia.
- **Control de errores:** si no hay stream activo, emite una advertencia; si ocurre una excepción, se registra.

`deinit()`

- **Parámetros:** ninguno.
- **Atributos modificados:** `audio_interface`, `device_index`.
- **Acciones:** cierra el objeto `PyAudio` y limpia los atributos relacionados al dispositivo.
- **Control de errores:** si ya está desinicializado, se registra y se evita repetir el proceso.

IV. PRUEBAS Y ENSAYOS

Para validar el correcto funcionamiento del módulo `BehringerHandler`, se desarrolló una batería de pruebas unitarias y funcionales empleando los marcos de prueba `unittest` y `pytest`, junto con técnicas de *mocking* a través de `unittest.mock`. El objetivo principal de los ensayos fue asegurar la correcta inicialización del dispositivo, la grabación de audio, el manejo de errores y la robustez del sistema ante condiciones adversas.

El framework `unittest` forma parte de la biblioteca estándar de Python. Este framework sigue el modelo xUnit, ampliamente adoptado en distintos lenguajes de programación. Las pruebas se organizan en clases que heredan de `unittest.TestCase`. Cada método representa un caso de prueba individual. El framework proporciona métodos de aserción para validar el comportamiento del código bajo prueba. También permite configurar y limpiar el entorno de prueba mediante los métodos `setUp()` y `tearDown()`. En este proyecto, `unittest` sirvió como base para estructurar los casos que evalúan la inicialización del dispositivo, la grabación de audio, el manejo de errores y los mecanismos de cierre.

El framework `pytest` es una herramienta externa que amplía las capacidades de prueba en Python. Su sintaxis es más concisa y flexible que la de `unittest`. Las pruebas se pueden definir como funciones independientes, sin necesidad de clases. `pytest` también ofrece mecanismos potentes como las *fixtures*, la captura de logs con `caplog` y una integración natural con `assert`. En este proyecto, `pytest` se utilizó para verificar el sistema de logging, simular errores y complementar los ensayos definidos con `unittest`.

Ambos frameworks resultan compatibles y pueden coexistir dentro del mismo entorno de pruebas. `unittest` aportó una estructura clásica y controlada para los casos más detallados. Por su parte, `pytest` ofreció herramientas modernas que facilitaron la validación de condiciones específicas, como el registro de advertencias y errores.

En el contexto de pruebas de software, un *mock* es un objeto simulado. Este objeto reemplaza a una dependencia real y permite controlar su comportamiento durante los ensayos. El uso de *mocks* facilita el aislamiento del componente que se desea evaluar. También permite verificar respuestas específicas ante distintos estímulos. En los ensayos del módulo `BehringerHandler`, se reemplazaron objetos como `pyaudio.PyAudio`, `wave.open`, `datetime.datetime` y `time.time`. Esta técnica hizo posible simular la presencia de dispositivos de audio. También permitió interceptar la creación de archivos sin escribir en el disco. Además, facilitó la generación de marcas de tiempo controladas y la simulación de condiciones temporales específicas. Gracias a este enfoque, se pudieron reproducir distintos escenarios. Por ejemplo, se evaluaron fallas de inicialización, errores de grabación y ausencia de datos en la cola, sin necesidad de depender de hardware real.

En los ensayos del módulo `BehringerHandler`, se utilizó una *fixture* para reiniciar los manejadores del sistema de logging antes de cada prueba. Una *fixture* es una función o recurso que prepara el entorno antes de ejecutar una prueba. También puede encargarse de restaurar el estado original una vez finalizada la ejecución. En el marco de `pytest`, las *fixtures* permiten definir comportamientos repetitivos de forma declarativa y reutilizable. Esta estrategia evitó interferencias entre casos de prueba consecutivos. También permitió verificar con precisión los mensajes generados por el sistema, en combinación con la herramienta `caplog`.

IV.1. Cobertura de pruebas

Las pruebas se estructuraron para cubrir los siguientes aspectos del módulo:

- Inicialización del dispositivo de audio Behringer.
- Manejo de errores cuando el dispositivo no está disponible o es inválido.
- Grabación de audio y generación del archivo de salida en formato `.wav`.
- Validación del flujo de ejecución en presencia o ausencia de datos en la cola de audio.
- Verificación del correcto registro de mensajes en el sistema de logging.
- Evaluación del comportamiento de los métodos auxiliares, como `open()`, `close()` y `deinit()`.

IV.2. Estrategia de prueba

Se utilizaron técnicas de simulación (*mocking*) para reemplazar el comportamiento del hardware de audio y del sistema de archivos, permitiendo un entorno controlado para la validación lógica del código. Particularmente, se realizaron los siguientes reemplazos:

- `pyaudio.PyAudio`: Simulado para emular la presencia (o ausencia) del dispositivo Behringer.
- `wave.open`: Interceptado para evitar la creación real de archivos de audio.
- `datetime.datetime.now()`: Reemplazado para asegurar consistencia en los nombres de archivo generados.
- `time.time()`: Controlado para validar condiciones temporales durante la grabación.

La ejecución de los ensayos se realiza desde la línea de comandos. Para ello, se utiliza el ejecutable de `pytest`, que detecta automáticamente todos los archivos de prueba cuyo nombre comienza con `test_` o termina en `_test.py`. El comando que se utiliza en el módulo `BehringerHandler` para lanzar todas las pruebas, que incluyen tanto las escrdefinidas para elitas con `unittest` como las específicas de `pytest`, es:

```
pytest tests/modules/behringer/test_behringer_handler.py -v
```

El modificador `-v` activa el modo detallado (*verbose*), que muestra el nombre de cada prueba y su resultado. La salida por consola incluye también los logs generados durante la ejecución, lo que permite verificar que los mensajes esperados fueron correctamente emitidos.

IV.3. Casos de prueba destacados

- **Inicialización exitosa**: Se verifica que el dispositivo Behringer es detectado y que se almacena correctamente el índice del dispositivo.
- **Inicialización fallida**: Se simula la ausencia de dispositivos y se valida que el índice del dispositivo permanece `None`.
- **Grabación de audio**: Se simula una sesión de grabación, incluyendo escritura de datos en la cola, y se verifica la creación lógica del archivo de salida.
- **Manejo de cola vacía**: Se evalúa la robustez del sistema ante la ausencia de datos durante la grabación (control de excepciones por `queue.Empty`).
- **Log de advertencias**: Se valida que se emiten advertencias apropiadas cuando se intenta abrir o grabar sin haber inicializado el sistema.
- **Callback del flujo de audio**: Se prueba el comportamiento del método de `callback` tanto en estado de grabación como en reposo.
- **Cierre con errores**: Se simula una excepción al intentar cerrar un flujo de audio defectuoso y se verifica que se registre el error correspondiente.

IV.4. Integración con Pytest

Además de los tests basados en `unittest`, se incorporaron pruebas con `pytest` utilizando `fixtures` para configurar adecuadamente los manejadores de logging. Se utilizó `caplog` para capturar mensajes de log y confirmar que los errores y advertencias se registran correctamente.

V. RESULTADOS

Todas las pruebas fueron ejecutadas de forma automatizada y arrojaron resultados satisfactorios. No se detectaron fallos ni excepciones no controladas. El módulo demostró comportarse de forma robusta frente a condiciones esperadas y adversas, lo cual valida su uso en entornos complejos de adquisición de señales de audio para aplicaciones tanto de laboratorio como de campo.

En la figura 3 se pueden observar los resultados obtenidos al ejecutar veinte pruebas unitarias al código. Se puede apreciar que todas las pruebas fueron exitosas.

```
(.venv) boya@kaise:~/eammra$ pytest modules/behringer/tests/test_behringer_handler.py -v
============================================= test session starts ==============================================
platform linux -- Python 3.12.3, pytest-8.3.5, pluggy-1.5.0 -- /home/boya/eammra/.venv/bin/python3
cachedir: .pytest_cache
rootdir: /home/boya/eammra
configfile: pytest.ini
collected 20 items

modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_callback_not_recording PASSED [  5%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_callback_recording PASSED    [ 10%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_close_with_error PASSED      [ 15%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_close_without_stream PASSED  [ 20%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_deinit_without_init PASSED   [ 25%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_init_device_found PASSED     [ 30%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_init_device_not_found PASSED [ 35%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_init_with_invalid_device PASSED [ 40
%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_logging PASSED               [ 45%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_open_stream_error PASSED     [ 50%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_open_without_init PASSED     [ 55%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_record PASSED                [ 60%]
modules/behringer/tests/test_behringer_handler.py::TestBehringerHandler::test_stop_recording PASSED        [ 65%]
modules/behringer/tests/test_behringer_handler.py::test_open_logs_warning_if_not_initialized PASSED        [ 70%]
modules/behringer/tests/test_behringer_handler.py::test_record_logs_warning_if_not_initialized PASSED      [ 75%]
modules/behringer/tests/test_behringer_handler.py::test_record_timeout_handling PASSED                     [ 80%]
modules/behringer/tests/test_behringer_handler.py::test_logger_formatter_set PASSED                        [ 85%]
modules/behringer/tests/test_behringer_handler.py::test_init_device_not_found_logs_warning PASSED          [ 90%]
modules/behringer/tests/test_behringer_handler.py::test_deinit_without_init_logs PASSED                    [ 95%]
modules/behringer/tests/test_behringer_handler.py::test_close_with_error_logs PASSED                       [100%]

============================================== 20 passed in 1.68s ==============================================
(.venv) boya@kaise:~/eammra$
```

FIG. 3. Resultados de test unitarios al código.

Un test de cobertura permite medir qué proporción del código fuente es ejecutada durante la realización de las pruebas. Esta métrica es útil para identificar partes del código que no fueron evaluadas y, por lo tanto, podrían contener errores no detectados. Para calcular la cobertura en este proyecto, se utilizó el plugin `pytest-cov`, que extiende `pytest` con opciones específicas. La cobertura se aplicó al módulo `behringer_handler.py` y se ejecutó con el siguiente comando:

```
pytest --cov=modules.behringer.behringer_handler modules/behringer/tests/
 --cov-report=term-missing -v
```

La salida del análisis mostró una cobertura del 96 %. De un total de 129 líneas ejecutables, 5 no fueron alcanzadas por las pruebas. Las líneas omitidas se encuentran en las posiciones 49–50, 72–73 y 101 del archivo fuente. Esta información resulta valiosa para planificar nuevas pruebas que completen la validación del módulo.

```
---------- coverage: platform linux, python 3.12.3-final-0 -----------
Name                                      Stmts   Miss  Cover   Missing
-----------------------------------------------------------------------
modules/behringer/behringer_handler.py      129      5    96%   49-50, 72-73, 101
-----------------------------------------------------------------------
TOTAL                                       129      5    96%
```

FIG. 4. Ejecución de pruebas con medición de cobertura.

VI. CONCLUSIONES

El desarrollo del módulo `BehringerHandler` permitió incorporar una solución robusta y extensible para la configuración y el control de la interfaz de audio Behringer UMC204HD en el sistema EAMMRA. Su diseño modular y su implementación en Python favorecen la integración con otros componentes del sistema y facilitan el mantenimiento y la evolución futura del software.

Durante el proceso de desarrollo, se aplicaron buenas prácticas de codificación basadas en las guías PEP-8 y PEP-257. Se emplearon herramientas de análisis estático como Flake8, Pylint y Black, que contribuyeron a mejorar la legibilidad, la coherencia y la calidad general del código. Estas herramientas resultaron fundamentales para reducir defectos y facilitar el trabajo colaborativo en el proyecto.

La validación funcional del módulo se realizó mediante una batería de pruebas automatizadas. Se utilizaron los marcos `unittest` y `pytest`, junto con técnicas de *mocking* y *fixtures*, que permitieron verificar el comportamiento del sistema bajo distintas condiciones, sin necesidad de depender del hardware real. Las pruebas contemplaron tanto situaciones normales como escenarios de falla y manejo de errores.

Los resultados de las pruebas indicaron un funcionamiento correcto y estable del módulo. No se detectaron errores durante la ejecución, y la cobertura alcanzó el 96 % de las líneas ejecutables. Este nivel de cobertura refleja un alto grado de confianza en el comportamiento del módulo ante condiciones esperadas y adversas.

Como línea futura de trabajo, se propone implementar pruebas de integración en conjunto con otros módulos del sistema EAMMRA, así como ensayos funcionales en condiciones reales de operación. También se sugiere completar la cobertura restante, documentar formalmente la API del módulo, y evaluar mecanismos de tolerancia a fallos para mejorar su desempeño en entornos hostiles o de difícil acceso.

REFERENCIAS

Behringer (2019). Umc204hd audio interface. https://www.behringer.com/product.html?modelCode=P0AUX. Accessed: 2025-03-29.

Bos, P., Cinquini, M., Prario, I., y Blanc, S. (2016). EAMMRA: Ingeniería conceptual. INF. TEC. AS 03/16, DAS, DIIV.

Cinquini, M., Bos, P., Prario, I., y Rojo, R. M. (2019). Integración de subsistemas para el prototipo de eammra. INF. TEC. AS 02/19, DAS, DIIV.

Cordasco, I. S. (2016). Flake8: Your tool for style guide enforcement. https://flake8.pycqa.org/en/latest/#. Accedido: 05/03/2024.

Ezcurra, H., Prario, I., Bos, P., Cinquini, M., y Blanc, S. (2019). Diseño de una boya prototipo para medición de nivel de ruido submarino en zona costera. INF. TEC. AS 01/19, DAS, DIIV.

Goodger, D., y van Rossum, G. (2001). Python enhancement proposals 257 – docstring conventions. https://peps.python.org/pep-0257/. Accedido: 04/03/2024.

Kaise (2025). Kbox-s j6412. https://www.tempelgrouplatam.com. Folleto publicado por TEMPEL GROUP.

Krekel, H., *et al.* (2025a). *pytest Documentation, Release 8.4*. pytest development team. Accessed: 2025-03-29. URL https://docs.pytest.org/en/latest/

Krekel, H., *et al.* (2025b). pytest: simple and powerful testing with python. Accessed: 2025-03-29. URL https://pytest.org

Logilab, y contributors to Pylint (2024). Pylint. https://pylint.readthedocs.io/en/stable/. Accedido: 05/03/2024.

van Rossum, G., Warsaw, B., y Coghlan, A. (2001). Pep 8 – style guide for python code. https://peps.python.org/pep-0008/. Accedido: 04/03/2024.

Łukasz Langa, y contributors to Black (2018). The uncompromising code formatter. https://black.readthedocs.io/en/stable/. Accedido: 05/03/2024.