



Indice Modulo 0

- ¿Por qué Python? Estadísticas
- ¿Qué es el Mundo IT? Información de Python en el mercadoIT
- ¿Qué es un algoritmo? ¿Qué es un programa?
- Instalación de Python intérpreter (junto al docente para personalizarla)
 - Organización Python.org (intérprete) y comunidad Python
- ¿Qué es un IDE?
- Selección e Instalación de uno o más IDEs
 - Instalación del IDE en tu propio pc
 - Uso de un IDE en línea
- Desarrollo en consola "CLI", "Command Line Interface" (Interfaz de Línea de Comandos).
- Python en los sistemas operativos (Linux, MacOS)
- Algunos ejemplos de Python en el mercado IT
- Lenguajes complicados e interpretados.
 - Ventajas y desventajas de cada una.
 - Python y su posibilidad de trabajar como interpretado y poder ser compilador



Modulo 0

- **¿Porqué Python?**

Python es un lenguaje de programación versátil que se utiliza en una amplia variedad de campos, desde desarrollo web y científico (aun pobre) hasta inteligencia artificial, Internet of Things, análisis de datos, la ciencia de la computación etc. Esto permite a los estudiantes aplicar sus conocimientos en diversos proyectos y áreas de interés.

Python es un lenguaje de programación potente y flexible que puede utilizarse para crear aplicaciones complejas.

Es gratuito y de código abierto. Python es un lenguaje de programación gratuito y de código abierto, lo que significa que es accesible para todos.

Facilidad de Aprendizaje: Python es conocido por su sintaxis simple y legible, lo que facilita su aprendizaje para principiantes. Esto lo convierte en una excelente opción para aquellos que están comenzando a programar.

Amplio Apoyo en Bibliotecas: Python tiene bibliotecas poderosas y extensas, como NumPy, Pandas, TensorFlow y PyTorch, que facilitan tareas específicas, como el análisis de datos y el aprendizaje automático.

V

Demandia en la Industria: Python es ampliamente utilizado en la industria, lo que significa que los conocimientos adquiridos pueden ser fácilmente aplicables en el ámbito laboral.

e

Enfoque en la Productividad: Python se centra en la legibilidad y la productividad, lo que permite a los programadores escribir código de manera más eficiente y rápida.

n

Sintaxis sencilla y legible: Python se destaca por su sintaxis simple y legible, lo que facilita su aprendizaje y comprensión. Su estructura de indentación obligatoria ayuda a mantener un código limpio y organizado aun sin experiencia en programación.

t

Amplia comunidad y recursos: Python cuenta con una gran comunidad de desarrolladores que brindan soporte, comparten conocimientos y crean bibliotecas y frameworks útiles. Además, existen numerosos recursos en línea, como documentación, tutoriales, foro y cursos, que facilitan el aprendizaje y el desarrollo en Python.

j

Comunidad y cultura de colaboración: La comunidad de Python es conocida por su actitud de apertura, colaboración y ayuda mutua. Los desarrolladores de Python están dispuestos a compartir conocimientos, colaborar en proyectos y ayudar a resolver problemas. Ser parte de esta comunidad puede ser enriquecedor y brindarte oportunidades de crecimiento personal y profesional.

a

Versatilidad y aplicaciones prácticas: Python se utiliza en una amplia variedad de campos, como desarrollo web, análisis de datos, inteligencia artificial, aprendizaje automático, automatización de tareas, scripting y más. Esto significa que aprender Python puede abrirte puertas en diferentes industrias y roles laborales.



S

Amplia biblioteca estándar: Python cuenta con una biblioteca estándar muy completa, que incluye módulos para realizar tareas comunes, como manipulación de archivos, acceso a bases de datos, networking, procesamiento de texto, entre otros. Esto te permite ahorrar tiempo y esfuerzo al aprovechar estas funcionalidades listas para usar.

Fácil integración: Python es conocido por su capacidad de integración con otros lenguajes y sistemas. Puede interactuar con código escrito en C, C++, Java y otros lenguajes, lo que facilita la combinación de diferentes tecnologías en un proyecto.

Portabilidad: Python es un lenguaje multiplataforma, lo que significa que puedes escribir un código en Python y ejecutarlo en diferentes sistemas operativos, como Windows, macOS y Linux, sin realizar modificaciones significativas.

Enfoque en la legibilidad del código: La filosofía de diseño de Python se basa en la legibilidad del código, lo que fomenta buenas prácticas de programación y facilita la colaboración en proyectos de equipo. Esto también ayuda a reducir errores y a mantener el código a largo plazo.

Amplia adopción y demanda laboral: Python ha experimentado un crecimiento significativo en los últimos años y se ha convertido en uno de los lenguajes de programación más populares. Esto ha llevado a una alta demanda de profesionales con habilidades en Python en el mercado laboral. Aprender Python puede aumentar tus oportunidades de empleo y mejorar tus perspectivas.

Aprendizaje gradual y progresivo: Python es un lenguaje amigable para principiantes, lo que lo hace ideal para aquellos que están comenzando en la programación. Su sintaxis clara y legible, junto con su enfoque en la simplicidad, permite un aprendizaje gradual y progresivo, lo que ayuda a construir una base sólida en conceptos de programación.

Desarrollo rápido de prototipos y productividad: Python es conocido por su enfoque en la productividad y el desarrollo rápido de prototipos. La combinación de su sintaxis sencilla y sus bibliotecas de alto nivel permite a los desarrolladores escribir código de manera eficiente y rápida, lo que acelera el proceso de desarrollo de software.

Aprendizaje de conceptos de programación fundamentales: Al aprender Python, adquirirás conocimientos y habilidades fundamentales de programación que son transferibles a otros lenguajes. Aprenderás sobre estructuras de datos, control de flujo, funciones, objetos y módulos, lo que te brinda una base sólida para abordar otros lenguajes y paradigmas de programación.

D
e
s
v

Rendimiento Relativo: Comparado con algunos lenguajes de programación de bajo nivel como C++ o Java, Python puede ser más lento en términos de ejecución. Sin embargo, en muchos casos, este impacto en el rendimiento es insignificante.

Menos Control sobre Hardware: Python no es ideal para programación de bajo nivel o situaciones donde se necesita un control extremo sobre el hardware. Esto puede ser una desventaja en ciertos contextos.

No es la Mejor Opción para Todos los Escenarios: Aunque Python es versátil, no siempre es la mejor



<p>e n t a j a s</p>	<p>opción para todos los escenarios. Algunas aplicaciones específicas pueden beneficiarse más de otros lenguajes de programación.</p> <p>Dependencia de Versiones: Puede haber problemas de compatibilidad entre versiones de Python, lo que puede ser un inconveniente cuando se trabaja en proyectos a lo largo del tiempo (para mí es una ventaja no llevar la mochila llena de cosas inservibles).</p> <p>No es para nada el preferido en Frontend, en celulares, juegos, si bien se está haciendo un lugar poco a poco.</p> <p>Hay que estudiar.....</p>
--	---

A nivel personal, estudiar Python puede ayudarte a desarrollar tus habilidades de pensamiento lógico y resolución de problemas. También te ayudará a aprender a pensar de forma creativa y a resolver problemas de manera eficiente.

A nivel profesional, estudiar Python puede ayudarte a mejorar tus oportunidades laborales. El conocimiento de Python es muy demandado por las empresas, y puede ayudarte a conseguir un mejor trabajo o a mejorar tu salario.

• ¿Qué es el Mundo IT?

El mundo IT, "**Information Technology**", engloba todas las actividades y disciplinas relacionadas con el manejo, procesamiento, almacenamiento y transmisión de información mediante el uso de tecnologías. Esto incluye tanto hardware como software, así como la gestión y administración de sistemas de información.

Desarrollo de Software: Creación, diseño y mantenimiento de programas y aplicaciones informáticas.

Redes y Comunicaciones: Diseño, implementación y mantenimiento de redes de comunicación que permiten la transferencia de datos entre dispositivos.

Seguridad Informática: Protección de sistemas, redes y datos contra amenazas y ataques cibernéticos.

Gestión de Datos: Almacenamiento, organización y análisis de grandes cantidades de datos para obtener información valiosa.

Soporte Técnico: Asistencia y resolución de problemas relacionados con hardware y software.

Inteligencia Artificial y Aprendizaje Automático: Desarrollo de sistemas y algoritmos que pueden realizar tareas que normalmente requieren inteligencia humana.

Cloud Computing: Uso de servicios y recursos informáticos a través de internet en lugar de en dispositivos locales.

Ciberseguridad: Con la creciente preocupación por la ciberseguridad, las habilidades en Python son cada vez más valiosas para tareas como análisis de seguridad, automatización de scripts y respuesta a incidentes.

Desarrollo Web: Creación y mantenimiento de sitios web y aplicaciones web.

El mundo IT es dinámico y está en constante evolución debido a los avances tecnológicos. Profesionales de IT, también conocidos como tecnólogos de la información o informáticos, desempeñan roles clave en la mayoría de



las organizaciones, ya que contribuyen al funcionamiento eficiente de los sistemas de información y tecnología.

El mercado IT es un mercado en constante crecimiento y demanda, y Python es uno de los lenguajes de programación más populares y demandados en este mercado.

Según un estudio de Stack Overflow, Python es el lenguaje de programación más popular entre los desarrolladores de software, y el segundo lenguaje de programación más popular entre los científicos de datos.

- Información de Python en el mercadoIT

La demanda de Python se debe a una serie de factores, entre los que se incluyen:

- **Su facilidad de aprendizaje.** Python es un lenguaje de programación relativamente fácil de aprender, incluso para personas sin experiencia en programación.
- **Su versatilidad.** Python se puede utilizar para una amplia gama de tareas, desde el desarrollo web y móvil hasta el análisis de datos y la ciencia de la computación.
- **Su potencia.** Python es un lenguaje de programación potente y flexible que puede utilizarse para crear aplicaciones complejas.
- **Mejorar sus habilidades de programación.** Python es un lenguaje de programación moderno y potente que puede ayudar a los profesionales a desarrollar sus habilidades de programación y a mantenerse actualizados con las últimas tendencias tecnológicas.
- **Expandir sus oportunidades laborales.** El conocimiento de Python es muy demandado por las empresas, y puede ayudar a los profesionales a conseguir un mejor trabajo o a mejorar su salario.

En conclusión, el mercado IT es un mercado en constante crecimiento y demanda, y Python es uno de los lenguajes de programación más populares y demandados en este mercado. El conocimiento de Python es una valiosa herramienta para cualquier profesional del sector IT.

- Según un estudio de LinkedIn, el salario medio de un desarrollador de software con conocimientos de Python es de 110.000 dólares.
- Según un estudio de Indeed, la demanda de desarrolladores de Python ha crecido un 28% en los últimos cinco años.
- Según un estudio de Glassdoor, el 93% de las empresas encuestadas están buscando candidatos con conocimientos de Python.

Estos datos muestran que la demanda de Python en el mercado IT es alta y está en constante crecimiento.

- ¿Qué es un algoritmo? ¿Qué es un programa?

Algoritmo: Un algoritmo es una secuencia finita y ordenada de pasos o instrucciones que describe el método para realizar una tarea o resolver un problema específico. Los algoritmos son esenciales en la informática y la programación, ya que proporcionan un plan sistemático para la resolución de problemas. Un buen algoritmo debe ser claro, preciso y capaz de producir resultados correctos.

Programa: Un programa es un algoritmo generado en un lenguaje de programación. Es un conjunto de instrucciones escritas en este caso en lenguaje Python que le indica a una computadora cómo realizar una tarea específica. Los programas implementan algoritmos y están formados por un conjunto de comandos que la computadora puede



entender y ejecutar. Pueden ser tan simples como un pequeño script o tan complejos como un sistema operativo completo.

- **Estadísticas**

clickee sobre le link para ir a la página web

1. [PYPL - PYPL PopularitY of Programming Language index](#)
2. [TIOBE Index](#)
3. [GitHub Pull Requests Statistics](#)
4. [Stackscale - Ranking Python vs Java](#)
5. [Stack Overflow Developer Survey Trends](#)
6. [Stack Overflow Developer Survey](#)
7. <https://lenguajesdeprogramacion.net/>
8. [Programas en Python que serían estas grandes marcas sin este lenguaje](#)
9. [Lenguajes de programación más usados en 2023](#)
10. [Mejores lenguajes de programación para 2023](#)
11. [Lenguajes de programación en 2023](#)
12. [Lenguajes de programación más demandados en 2023 y cómo aprenderlos](#)
13. [Top 6 de lenguajes de programación para 2023](#)
14. [Lenguajes de programación más demandados en 2023](#)
15. [Mejores lenguajes de programación](#)
16. [Lenguajes de programación en el ámbito laboral](#)
17. [Mejores lenguajes de programación para 2023](#)
18. [5 lenguajes de programación más usados en 2022](#)
19. [Lenguajes de programación más demandados](#)
20. [Lenguajes de programación más populares según Stackscale](#)

- **Instalación de Python intérpreter** (junto al docente para personalizarla)

Descarga de <https://www.Python.org/downloads/> la última versión del intérprete Python

Link de descargas para:

Windows 8.1 en adelante Linux

MacOs 10 en adelante (Tener en cuenta el microprocesador que posee intel o AMR)

(Se prefiere la actualización por intermedio del propio Linux o macOS como Homebrew o apt-get).

- **Organización Python.org (intérprete) y comunidad Python**

Python es el intérprete que lee el código y lo convierte para que el procesador pueda entenderlo. También es el lenguaje en el que se escribe el código que es interpretado, y finalmente, es una comunidad de programadores que trabajan en este lenguaje. Hay reglas impuestas por el intérprete que no admiten discusión; se aceptan o el código no funciona. Además, existen reglas de la comunidad para que el código pueda ser leído y comprendido por el conjunto de programadores."

Python es un lenguaje de programación gratuito, multiplataforma y de código abierto que es potente y fácil de aprender. Es ampliamente utilizado y compatible. Para obtener más información sobre Python,



visite Python.org.

La misión de Python Software Foundation es promover, proteger y hacer avanzar el lenguaje de programación Python, y apoyar y facilitar el crecimiento de una comunidad diversa e internacional de programadores de Python.

Python Software Foundation (PSF) es una organización de membresía sin fines de lucro dedicada al avance de la tecnología de código abierto relacionada con el lenguaje de programación Python.

La directiva de la PSF está formada con personas que tienen como objetivo mejorar la comunidad de Python en todo el mundo.

• ¿Qué es un IDE?

Un Entorno de Desarrollo Integrado (IDE, por sus siglas en inglés) para Python es una herramienta que proporciona un conjunto de características y herramientas integradas diseñadas específicamente para facilitar el desarrollo, la depuración y la administración de proyectos en Python. Estos entornos están diseñados para mejorar la productividad del programador ofreciendo características como resaltado de sintaxis, completado automático, depuración integrada, gestión de paquetes y otros servicios.

PyCharm: Desarrollado por JetBrains, PyCharm es uno de los IDE más populares para Python. Ofrece una variedad de características, como resaltado de sintaxis avanzado, completado automático, depuración integrada, pruebas unitarias y soporte para frameworks web como Django.

Visual Studio Code (VSCode): Aunque es un editor de código fuente creado por Microsoft, VSCode tiene una amplia gama de extensiones que lo convierten en un entorno de desarrollo completo para Python. Ofrece resaltado de sintaxis, sugerencias de código, integración con control de versiones y depuración.

Jupyter Notebooks: Especialmente popular en el ámbito científico y de análisis de datos, Jupyter Notebooks permite la creación de documentos interactivos que contienen código, visualizaciones y texto narrativo. Es ideal para la exploración de datos y la presentación de resultados.

Spyder: Es un IDE diseñado específicamente para la ciencia de datos. Incluye herramientas como consola interactiva, explorador de variables, y soporte para IPython.

Atom: Similar a VSCode, Atom es un editor de código fuente extensible que puede ser convertido en un entorno de desarrollo para Python mediante la instalación de paquetes y extensiones adecuados.

Sublime Text: Un editor de texto ligero y altamente personalizable que puede ser mejorado con paquetes específicos para el desarrollo de Python.

IDLE: Este es el IDE que se incluye con la instalación estándar de Python. Proporciona un entorno básico con una shell interactiva y editor de texto, pero carece de algunas características avanzadas presentes en otros IDE.

,



<ul style="list-style-type: none"> Instalación del IDE en tu propia pc 		*
Download Visual Studio Code	https://code.visualstudio.com/download	*
Geany	https://www.geany.org/	*
Eclipse	https://www.eclipse.org/downloads/	
pyzo	https://pyzo.org/start.html	
Eric	https://eric-ide.Python-projects.org/eric-download.html	
PyCharm	https://www.jetbrains.com/pycharm/download/	
Spyder	https://www.spyder-ide.org/	
Anaconda Jupyter, para ciencia de datos	https://www.anaconda.com/download	
<u>Disponibles para:</u> Windows 8.1 en adelante		
Linux		
MacOs 10 en adelante		

* Recomendado para instalar

<ul style="list-style-type: none"> Uso de un IDE en línea (si bien más lento no requiere instalación) en caso de no poseer autorización como administrador para instalar software
REPL.it https://replit.com/ (uno de los mejores) <p>Es un intérprete para Python desarrollado de manera abierta y disponible en Github. Es un compilador online que emula un terminal en el cual se ejecuta todo el código escrito. Además de los lenguajes ya citados ofrece soporte para escribir en otros 16 lenguajes [variados] incluido Javascript y Coffee Script.</p> <p>Algunas características</p> <ul style="list-style-type: none"> Desarrollado en js y Python: Replit ofrece una interfaz minimalista. Se puede acceder desde cualquier dispositivo. Codificación directa y libre de distracción. Provee de ejemplos necesarios para cada lenguaje. Permite Guardar y compartir nuestro código.
PythonAnywhere https://www.pythonanywhere.com/ <p>Es un ambiente de desarrollo integrado (IDE) online para Python y servicio de alojamiento en línea. Tiene 512 MB disponibles para almacenamiento de archivos en las cuentas gratuitas, terminal bash, Python, MySQL, PyPy, IPython, y terminales personalizadas. También cuenta con posibilidad de ejecutar aplicaciones web de Python y Jupyter Notebooks para cuentas no gratuitas. Se pueden programar tareas y manejar bases de datos Postgres y MySQL. Al contar con una terminal bash, PythonAnywhere permite utilizar el gestor de paquetes de Python, pip, para instalar módulos que no pertenezcan a la librería estándar de Python.</p>
Ideone.com https://www.ideone.com/ <p>Es un intérprete Python y compilador e IDE online para Python, C/C++, Java, Haskell, PHP, y muchos otros lenguajes de programación. Cuenta con una sencilla pero muy útil interfaz en la que se puede copiar el</p>



código de un programa para luego compilarlo o ejecutarlo. Es una buena alternativa para alguien que necesite ejecutar o probar un programa de forma rápida, anónima y sencilla. Tiene opciones para cambiar la entrada estándar, cambiar la privacidad del código e ingresar notas alternativas al código del programa.

Codeanywhere <https://www.codeanywhere.com/>

Es un IDE multilenguaje basado en la nube. Su interfaz cuenta con un editor de texto y una terminal que varía dependiendo del entorno con el cual se defina el contenedor, el cual es el lugar en donde se guardarán los archivos que se redacten o carguen. Cuenta con ambientes de Ubuntu y CentOS. Tiene posibilidades de instalar paquetes personalizados en los ambientes con el gestor de paquetes apt, en el entorno Ubuntu, y yum, en el entorno CentOS, y cuenta también con el gestor de paquetes de Python, pip. Inicialmente se cuenta con una prueba gratuita del sistema con la cual se tiene acceso por 7 días y se puede generar un contenedor y una conexión remota.

Nota: Hay muchos más IDEs online en la web para que puedas elegir.

Tenga en cuenta que normalmente cada uno deberá grabar los scripts (nombre y extensión) para poder ejecutarlo. En caso de uso en línea esto no es del todo necesario.

Si usa un ide en línea y no desea registrarse una vez finalizada el script y para guardar el ejercicio podrá copiarse y pegarse en un block de notas para que el alumno tenga los scripts guardados

En caso de instalación en la pc del alumno recuerde que probablemente Python en su primer pantalla de instalación marcar add Python to Path así se agrega a las variables de entorno en Windows. Esta información paso a paso se encuentra en la guía de instalación que acompaña este archivo.

- desarrollo en consola "CLI", "Command Line Interface" (Interfaz de Línea de Comandos).

Consola - CLI:

La "consola" se refiere a una interfaz de línea de comandos (CLI) que permite a los usuarios interactuar con un sistema operativo o una aplicación mediante comandos de texto.

Definición: La consola es una interfaz de texto en la que los usuarios ingresan comandos para realizar acciones en un sistema operativo.

Características:

- Se ejecuta en un entorno de texto sin gráficos.
- Los comandos se ingresan mediante teclado y se muestran como texto.
- Es eficiente para usuarios experimentados y tareas automatizadas mediante scripts.
- Puede proporcionar información detallada y salida de texto.
- Ejemplos: La Terminal en sistemas Unix/Linux, el Command Prompt en Windows.

PowerShell:

Definición: PowerShell es una interfaz de línea de comandos avanzada y un marco de scripting desarrollado por Microsoft.

Características:

- Combina la funcionalidad de la consola tradicional con características de scripting avanzadas y objetos.
- Utiliza cmdlets (comandos especializados) para realizar acciones específicas.
- Ofrece acceso a funciones del sistema operativo y a productos de Microsoft.
- Es altamente extensible y permite la automatización de tareas complejas.
- Ejemplos: PowerShell en sistemas Windows.

Interfaces Gráficas de Usuario (GUI):

Definición: Las GUI proporcionan una interfaz visual que permite a los usuarios interactuar con un sistema utilizando elementos gráficos como ventanas, botones y menús.

Características:

- Utilizan elementos visuales para representar funciones y opciones.
- Ofrecen una experiencia de usuario más amigable, especialmente para usuarios menos experimentados.



No requieren el uso de comandos de texto para la mayoría de las tareas.

Son comunes en sistemas operativos de escritorio y aplicaciones.

Ejemplos: El escritorio de Windows, el entorno de escritorio en sistemas Unix/Linux (por ejemplo, GNOME, KDE).

Diferencias:

La consola y PowerShell son interfaces basadas en texto, mientras que las GUI son interfaces gráficas.

La consola tradicional utiliza comandos específicos del sistema operativo, mientras que PowerShell utiliza cmdlets para realizar acciones específicas y proporciona un entorno de scripting más avanzado.

Las GUI ofrecen una experiencia más visual y suelen ser preferidas por usuarios que no están familiarizados con la línea de comandos.

PowerShell puede interactuar con el sistema operativo y productos de Microsoft de manera más integral, mientras que la consola tradicional es más general y puede depender de comandos específicos del sistema operativo.

• Python en los sistemas operativos (Linux, MacOS)

Python suele venir preinstalado en muchos sistemas operativos basados en Unix, como macOS y varias distribuciones de Linux, si bien no es un componente esencial del kernel de los sistemas operativos, es imprescindible para su trabajo. Por lo que NO borres o actualices Python por afuera del OS porque puede dejar de funcionar y ya no bootear.

La presencia de Python en estos sistemas es para brindar a los usuarios una herramienta de programación y scripting por defecto.

Es importante tener cuidado al realizar actualizaciones o cambios en la configuración de Python, especialmente si se trata de la versión que viene preinstalada en el sistema operativo. Modificar o eliminar la versión predeterminada de Python puede afectar la funcionalidad de algunas herramientas y aplicaciones del sistema que pueden depender de una versión específica de Python.

Recomendaciones:

- Entornos Virtuales - cajas de arena: Cuando trabajas en proyectos específicos que requieren una versión específica de Python y sus dependencias, se recomienda utilizar entornos virtuales (por ejemplo, con virtualenv o venv) para aislar las dependencias del proyecto y evitar conflictos con la versión del sistema.
- Gestión de Paquetes: Usa herramientas de gestión de paquetes como pip para instalar y gestionar paquetes específicos para proyectos individuales. Esto ayuda a evitar conflictos entre las dependencias del sistema y las del proyecto.
- Actualizaciones Cautelosas: Si decides actualizar Python, hazlo de manera consciente y asegúrate de comprender las posibles implicaciones. En algunos casos, las actualizaciones pueden romper la compatibilidad con ciertas aplicaciones.
- Reparación o Restauración: Si por alguna razón Python deja de funcionar después de una actualización o cambio, puedes intentar reparar o reinstalar la versión predeterminada del sistema operativo desde los repositorios oficiales.



- **Algunos ejemplos de Python en el mercado IT**



El uso de Python en diversas empresas líderes como Instagram, Google (incluyendo el buscador, Google Cloud, Google Maps, YouTube), Spotify, Netflix, Quora, Dropbox, Reddit, Uber, Stripe, entre muchos otros, es evidencia de la prominencia y versatilidad del lenguaje en el ámbito tecnológico. La migración hacia Python se extiende incluso a gigantes tecnológicos como Meta (creador de PyTorch), Messenger, Facebook, Whatsapp, Amazon, y Microsoft, que recientemente incorporó a Guido van Rossum, el creador de Python, a su División de Desarrolladores desde noviembre de 2020.

Según las últimas noticias de Microsoft, Python se integrará en Excel mediante los conectores Power Query integrados. Para acceder a las funciones de Python en Excel, basta con escribir 'PY' en cualquier celda, similar a cómo se utiliza '=' para acceder a funciones en Excel. A través de Anaconda, el repositorio de Python para empresas, se harán disponibles librerías reconocidas como pandas, statsmodels, Matplotlib, y otras, facilitando su utilización en Excel.

Es esencial resaltar que los cálculos realizados en Python se ejecutarán en Microsoft Cloud, y los resultados se verán reflejados en las hojas de cálculo una vez procesados. Esta integración permitirá a los usuarios de Excel desarrollar fórmulas, tablas, gráficos y otras funcionalidades basadas en Python, lo que ampliará significativamente las capacidades y aplicaciones de este popular lenguaje de programación en el ámbito empresarial.

Para acceder a esta nueva funcionalidad, los usuarios pueden participar en el canal para insiders de Microsoft 365 o el canal beta. Inicialmente, se llevará a cabo una prueba limitada para usuarios de Windows, con planes de lanzamiento para otras plataformas en etapas posteriores. Es importante señalar que Python en Excel estará incluido en la suscripción de Microsoft 365, aunque algunas características pueden estar restringidas sin una licencia de pago. Esto subraya la naturaleza comercial de Microsoft, donde ciertas funcionalidades premium pueden requerir una suscripción paga para su acceso completo, como es común en su modelo de negocio. (sino no sería Microsoft ¿No?).

<https://support.microsoft.com/es-es/office/seguridad-de-datos-y-Python-en-excel-33cc88a4-4a87-485e-9ff9-f35958278327>

El entorno de ejecución del código Python ha experimentado una notable evolución gracias a la integración en la nube de Microsoft con seguridad de nivel empresarial. Este avance permite ejecutar el código en contenedores seguros, específicamente construidos en Azure Container Instances y aislados mediante hipervisores.

Dentro de estos contenedores, se incluye el intérprete de Python junto con un conjunto de bibliotecas protegidas



proporcionadas por Anaconda, utilizando la Distribución de Anaconda para Python. Este entorno, que ofrece una capa adicional de seguridad, contiene Python y bibliotecas creadas de origen, directamente suministradas por Anaconda.

Es crucial destacar que el código Python ejecutado en este entorno no tiene acceso a los dispositivos, cuentas ni redes del usuario. Además, no se concede acceso a tokens de usuario, asegurando un ambiente completamente aislado y seguro.

Una característica destacada es la capacidad del código Python para acceder a los datos a través de referencias utilizando la función xl(), integrada como parte de una fórmula de Python. Esto implica que las fórmulas de Python pueden leer valores de celda dentro del libro o valores de fuentes de datos externos mediante el nombre de conexión Power Query.

La salida del código Python se devuelve a los libros mediante la función de Excel =PY(), que muestra el resultado en la celda correspondiente. Sin embargo, es importante señalar que las funciones de Python solo pueden devolver resultados y no otros tipos de objetos como macros, código VBA o fórmulas adicionales.

El código Python no tiene acceso a otras propiedades del libro, como fórmulas, gráficos, tablas dinámicas, macros o código VBA. Los contenedores permanecen en línea mientras el libro está abierto o hasta que se agote el tiempo de espera, asegurando una experiencia fluida y eficiente. Es esencial mencionar que los datos no se conservan en Microsoft Cloud, resguardando así la privacidad y seguridad de la información. Este avance representa un paso significativo hacia un entorno de desarrollo y análisis de datos más seguro y conectado.

La integración de Python en Excel ofrece un entorno seguro y conectado en la nube de Microsoft con niveles de seguridad empresarial compatibles con Microsoft 365. Esta implementación permite trabajar colaborativamente en documentos almacenados en OneDrive y utilizar características como Analizar datos de Excel, destacando como ejemplos de experiencias conectadas de Microsoft 365.

Cuando se trata de abrir libros desde Internet o fuentes no confiables, Python en Excel sigue las mismas directivas de seguridad que Excel. La Vista protegida de Excel no ejecutará las fórmulas de Python en un libro abierto desde Internet, y si se abre con Microsoft Defender Protección de aplicaciones, las fórmulas de Python no se ejecutan de forma predeterminada. Además, Python en Excel ejecuta las fórmulas de Python en libros no confiables dentro de su propio contenedor aislado de hipervisor dedicado, evitando posibles interacciones o interferencias con otros códigos Python de libros abiertos.

La actualización de los contenedores en los que se ejecuta el código Python en Excel se realiza periódicamente. Los contenedores en Azure, entornos aislados y seguros, se actualizan para garantizar seguridad y confiabilidad. Se aplican revisiones al sistema operativo subyacente automáticamente, protegiendo el contenedor contra vulnerabilidades. Además, Python y las bibliotecas suministradas por Anaconda se actualizan para mantener la seguridad de los datos y la consistencia de los resultados numéricos.

Entre los casos de uso en los que se puede aprovechar Python en Excel se encuentran la importación de datos desde diversas fuentes, la limpieza y transformación de datos, la creación de visualizaciones interactivas, la exportación de datos o gráficos a diferentes formatos, y la automatización de procesos o flujos de trabajo mediante scripts o funciones de Python.

A pesar de las ventajas, hay algunos inconvenientes a considerar. La función está actualmente disponible en vista previa pública solo para los miembros del Canal Beta de Microsoft 365 Insiders y puede tener errores o problemas de rendimiento. También requiere una conexión a Internet, lo que puede ser inconveniente en entornos sin conexión o con conexiones lentas. Algunas capacidades estarán detrás de un muro de pago, sin detalles específicos sobre qué capacidades o cuánto costarán. Además, hay limitaciones en las bibliotecas y paquetes de Python que se pueden utilizar, así como diferencias en la sintaxis y el comportamiento de Python en comparación con otros



entornos.

- [Ejecución de scripts de Python en Power BI Desktop](#)
- [Python en el Editor de Consultas de Power BI Desktop](#)

- | |
|---|
| <ul style="list-style-type: none"> • Lenguajes complicados e interpretados. <ul style="list-style-type: none"> ◦ Ventajas y desventajas de cada una. ◦ Python y su posibilidad de trabajar como interpretado y poder ser compilador |
|---|

Los lenguajes complicados, en el contexto de la programación, se refieren a aquellos que tienen una sintaxis y semántica complejas, lo que puede hacer que su aprendizaje y uso sean más desafiantes para los programadores. Por otro lado, los lenguajes interpretados son aquellos cuyo código fuente se ejecuta línea por línea en tiempo de ejecución por un intérprete, sin la necesidad de una etapa de compilación previa.

Ventajas de Lenguajes Complicados:	Desventajas de Lenguajes Complicados:
<p>Eficiencia de Ejecución: Algunos lenguajes complicados están optimizados para ejecutarse de manera eficiente, lo que puede resultar en un rendimiento mejorado en comparación con lenguajes más simples.</p> <p>Control Fino: Ofrecen un mayor control sobre los recursos de la máquina y permiten realizar optimizaciones específicas.</p>	<p>Curva de Aprendizaje Pronunciada: La complejidad puede dificultar la curva de aprendizaje, especialmente para programadores principiantes.</p> <p>Mayor Probabilidad de Errores: La complejidad puede aumentar la probabilidad de errores y hacer que la depuración sea más difícil.</p>

Ventajas de Lenguajes Interpretados:	Desventajas de Lenguajes Interpretados:
<p>Portabilidad: Los programas escritos en lenguajes interpretados son generalmente más portátiles, ya que el código fuente puede ejecutarse en cualquier máquina que tenga el intérprete adecuado.</p> <p>Desarrollo Rápido: La capacidad de ejecutar y probar el código de inmediato facilita el desarrollo rápido y la experimentación.</p>	<p>Menor Rendimiento: En general, los lenguajes interpretados tienden a ser más lentos que los compilados, ya que el código se interpreta en tiempo de ejecución.</p> <p>Dependencia del Intérprete: La necesidad de un intérprete puede ser una desventaja en términos de distribución y dependencias del sistema.</p>

Python es un lenguaje interpretado por naturaleza, pero existen herramientas como Cython y PyInstaller que permiten compilar el código de Python a código máquina o empaquetarlo como una aplicación independiente. Esto brinda flexibilidad al desarrollador, ya que puede optar por la interpretación para la facilidad de desarrollo y la portabilidad, o la compilación para mejorar el rendimiento en situaciones específicas.

Ventajas de Interpretación en Python:	Ventajas de Compilación en Python:
<p>Desarrollo Rápido: La ejecución interactiva y la facilidad de escribir y probar código rápidamente son ventajas significativas.</p> <p>Portabilidad: Los programas Python son fácilmente portables entre diferentes sistemas operativos sin la necesidad de compilación separada.</p>	<p>Rendimiento Mejorado: Al compilar el código, se pueden lograr mejoras significativas en el rendimiento.</p> <p>Protección de Código Fuente: La compilación puede ocultar el código fuente, proporcionando cierto nivel de protección contra ingeniería inversa.</p>

<p>Nuitka</p> <p>“Traduce Python en un programa C que luego está vinculado contra libPython para ejecutar exactamente como CPython”</p> <p>La carpeta de distribución que emite parece comenzar alrededor de 24mb en Win64 (4mb de los cuales son las libs Tk no adquiridas) y casi se duplicó para una aplicación que tengo que usa widgets wx. La salida es</p>	<p>GCC:cython</p> <p>Cython convierte el código a lenguaje C para posteriormente compilarlo con un compilador C como puede ser GCC:</p> <pre>pip install cython</pre> <p>Después transformamos el código Python en C y compilamos con GCC:</p> <pre>cython --embed -o test.c test.py</pre>
--	---



una carpeta que contiene un archivo *.exe junto con cualquier archivo *.dll y *.pyd necesarios. Si no está familiarizado con *.pyd, en realidad es solo un *.dll que contiene function(s) con una firma específica que permite a Python importar de ello.

Como se puede ver, se ha generado un fichero ejecutable elf de Linux (exe en Windows), depende de las librerías de Python (y como en cualquier binario hay que tener cuidado de lo que se deja en los strings ya que con el comando strings se puede obtener texto en claro)

Pyinstaller:

Esta forma realmente empaquetamos el script Python y sus librerías dentro de un binario (no se requiere ni Python ni sus librerías instaladas en el sistema que se ejecute el binario generado)

```
pip install pyinstaller  
pyinstaller -F test.py
```

Auto-py-to-exe:

Forma gráfica de usar muy simplemente pyinstaller

Lenguajes compilados e interpretados.

- Código abierto
- Código cerrado - compilado

Existen dos formas de transformar un programa de un lenguaje de programación de alto nivel a un lenguaje de máquina:

Compilación: el código fuente se traduce una vez (sin embargo, este proceso debe repetirse cada vez que modifiquemos el código fuente) lenguajes de alto nivel a lenguaje ensamblador o lenguaje máquina, ejemplo, un archivo con extensión .exe en Microsoft Windows; este último será el distribuido a los usuarios finales. El programa que realiza la traducción se llama compilador.

Por lo general los lenguajes compilados son más rápidos y consumen menos recursos que los lenguajes interpretados en vista de que el archivo resultante es código de bajo nivel, mientras que los lenguajes interpretados deben seguir un proceso a través de varios niveles de abstracción hasta que las instrucciones son ejecutadas por el sistema.

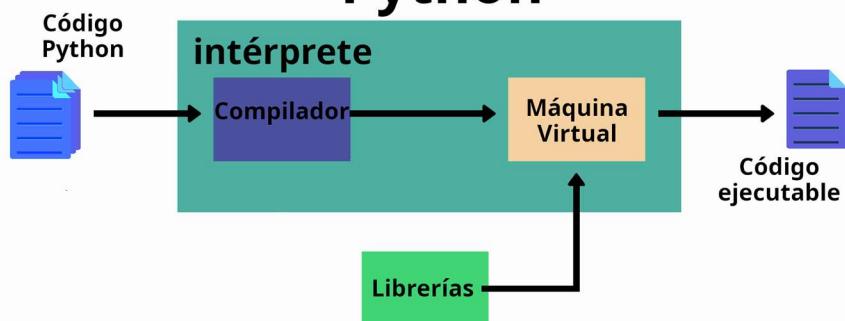
El script se compila en su totalidad creando un archivo de ejecución o de lectura directa por el cpu, Ej un .exe

Interpretación: cualquiera puede traducir el programa cada vez que se ejecute; el programa que realiza este tipo de transformación se denomina intérprete, ya que interpreta el código cada vez que este deba realizarse; esto significa que el usuario final también necesitará del programa intérprete para ejecutar nuestro programa.

En script que se interpreta lo hace a medida que el usuario lo requiere, no en la totalidad, en por ejemplo funciones donde el usuario no desea entrar el cpu nunca recibirá el código correspondiente. No se crea otro archivo

La programación es el acto de establecer una secuencia de instrucciones con la cual se causará el efecto deseado.

Proceso de interpretado Python





Cuando se ejecuta el código fuente de Python, el intérprete de Python compila el código en bytecode. El bytecode es una representación intermedia del programa que es independiente de la arquitectura de la máquina. No se puede leer directamente, pero el intérprete de Python lo entiende luego es necesario compilar el bytecode específicamente para la arquitectura de la máquina en la que se está ejecutando el programa. Esto implica que se utilice un compilador Just-in-Time (JIT) o un compilador AOT (Ahead-of-Time) que traduzca el bytecode a código de máquina adecuado para esa arquitectura. Una vez que se ha generado el código de máquina, la CPU de la máquina puede ejecutarlo directamente para producir los resultados.

El proceso de convertir el bytecode a código de máquina generalmente se realiza internamente por el intérprete de Python y los programadores normalmente no necesitan ocuparse por este proceso, ya que el intérprete se encarga de todo en tiempo de ejecución.

Python es un lenguaje interpretado de alto nivel.

ver apunte “Compiladores vs Interpretes.pdf”

Lenguajes de alto y bajo nivel.

Los lenguajes de programación un principio se pueden clasificar en dos categorías principales: lenguajes de alto nivel y lenguajes de bajo nivel. Estas categorías se refieren al nivel de abstracción que ofrecen al programador y su proximidad al lenguaje de máquina.

El avance de la tecnología redefine en nivel de abstracción y de cercanía al lenguaje humano, por lo que hoy hay una gama de grises entre assembly y Python

Lenguajes de bajo nivel:

Los lenguajes de bajo nivel constan de un conjunto básico de instrucciones que son ejecutados directamente por la unidad de procesamiento de un sistema de cómputo, tal como es el caso del lenguaje ensamblador. Dichos lenguajes están ligados intrínsecamente al tipo de procesador que los ejecuta y resultan ser muy complicados de elaborar e interpretar por las personas.

Los lenguajes de bajo nivel son aquellos que están más cerca del lenguaje de máquina y la arquitectura del hardware. Estos lenguajes requieren un mayor nivel de detalle y conocimiento sobre la computadora subyacente, ya que permiten un control más directo sobre el hardware y los recursos del sistema.

Ejemplos de lenguajes de bajo nivel son Assembly, C, C++, y algunos aspectos de otros lenguajes como C con sus características de punteros y acceso a memoria directa.

Ventajas de los lenguajes de bajo nivel:

- Permiten un control más fino sobre el hardware, lo que puede ser útil en situaciones específicas donde se necesita un alto rendimiento o acceso a características de bajo nivel.
- Son más eficientes en el uso de recursos de la computadora, ya que el código se traduce directamente a instrucciones de máquina.

Lenguajes de alto nivel:

Por su parte, los lenguajes de alto nivel son más accesibles para el ser humano e incluso menos dependientes del tipo de hardware, pero deben de ser a su vez traducidos a lenguaje de de bajo nivel. Se diseñan para que sean más fáciles de entender y utilizar por los programadores. Estos lenguajes se caracterizan por tener una sintaxis más cercana al lenguaje humano y ofrecer abstracciones de alto nivel que permiten expresar tareas y algoritmos de manera más natural. Esto significa que los programadores pueden concentrarse en resolver problemas sin preocuparse por los detalles específicos de la arquitectura de la computadora.



Ejemplos de lenguajes de alto nivel son Python, Java, C#, JavaScript, Ruby, y muchos otros. Estos lenguajes se utilizan ampliamente para el desarrollo de aplicaciones, sitios web, software empresarial y una variedad de otras soluciones.

Ventajas de los lenguajes de alto nivel:

- Facilitan la programación y la resolución de problemas.
- Son más portables, ya que están diseñados para ser independientes de la arquitectura de la computadora.
- Ofrecen bibliotecas y frameworks que facilitan el desarrollo de software.



- Para poder entender a Python en un contexto histórico les dejo el siguiente artículo

Fuente: https://es.wikipedia.org/wiki/IBM_Personal_Computer_XT

El IBM Personal Computer XT, normalmente abreviado como IBM XT o simplemente XT, fue el sucesor de IBM al IBM PC original. Fue puesto a la venta como IBM número de producto 5160 el 8 de marzo de 1983. XT son las siglas de eXtended Technology (Tecnología eXtendida). Estaba basado esencialmente en la misma arquitectura que el PC original, únicamente añadiendo algunas mejoras: se añadió un disco duro, 8 slots de expansión en vez de 5, más memoria en la tarjeta madre, una fuente de alimentación de mayor potencia y se le quitó la interfaz para cassetes del IBM PC original. Se convirtió en un estándar. El sistema estaba concebido para usuarios de negocios y un 3270 PC correspondiente implementaba la emulación de la terminal IBM 3270 que se comercializaría después, en octubre de 1983. Posteriormente, con el AT llegaría una nueva arquitectura de bus de 16 bits.

Características

El estándar XT traía de serie 128KB de memoria RAM en la tarjeta madre, una disquetera 5 1/4" de doble cara, doble densidad, de 360KB de tamaño completo, un disco duro Seagate de 10MB Seagate ST-412 y un adaptador asíncrono serial (RS 232) una fuente de alimentación de 130w suministraba energía eléctrica a todos los componentes. La placa base tenía ocho ranuras de expansión ISA de 8 bits, y al igual que el IBM PC, un microprocesador Intel 8088 corriendo a 4,77 MHz y un zócalo para coprocesador matemático Intel 8087. El sistema operativo con el que se solía vender fue el PC-DOS 2.0 y superior. Las ocho ranuras de expansión eran un aumento de las cinco del IBM PC, aunque tres de ellas eran utilizadas por el adaptador de la unidad de discos, el adaptador del disco duro y la tarjeta de video. Pronto se actualizó la especificación básica para estandarizarla a 256KB de memoria RAM en la tarjeta madre.

Había dos versiones de la placa base del XT. La original podía admitir hasta 256kB en la misma placa (en cuatro bancos de chips de 64kB), con un máximo de 640kB alcanzados usando tarjetas de expansión. La segunda revisión de la placa, introducida en 1986, podía admitir los 640kB enteros en la placa base, en dos bancos de chips



de 256kb y dos de 64kB. Las placas más recientes podían ser adaptadas para las últimas especificaciones después de un par de modificaciones menores. La segunda revisión de la placa tenía además un IBM BIOS revisado, que incluía soporte para el teclado expandido y reducía el tiempo de arranque a la mitad.

Como pueden ver la personal computer (pc) xt original podía admitir hasta 256kB en la misma placa en 1983/86
Trabajamos con potencia de 2

$$2^{**n} = 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536$$

1 byte = 8 bits

La Comisión Electrotécnica Internacional (IEEE) recomienda que se use k minúscula para decimal y K mayúscula para binario.

1024 bytes = 1 Kb (kilo byte)

Nuestro uso habitual en matemáticas es en base 10 o potencias de 10:

$$10^{**1} = 10$$

$$10^{**2} = 10 * 10 = 100$$

$$10^{**3} = 10 * 10 * 10 = 1000$$

$$10^{**6} = 1000000$$

En computación usamos base 2:

$$2^{**1} = 2$$

$$2^{**2} = 2 * 2 = 4$$

$$2^{**3} = 2 * 2 * 2 = 8$$

$$2^{**10} = 1024$$

$$2^{**20} = 1048576$$

Unidad		En bytes
1 byte (B)	1B	8 bits
1 kilobyte (KB)	1KB	1024 bytes
1 megabyte (MB)	1024 KB	1048576 bytes
1 gigabyte (GB)	1024 MB	1073741824 bytes
1 terabyte (TB)	1024 GB	1099511627776 bytes
1 petabyte (PB)	1024 TB	1125899906842624 bytes
1 Exabyte (EB)	1024 PB	1,125,899,906,842,624 bytes
1 Zettabyte (ZB)	1024 EB	1,152,921,504,606,846,976 bytes
1 Yottabyte (YB)	1024 ZB	1,180,591,620,717,411,303,424 bytes
hipotética 1 Brontobyte (BB)	1024 YB	1,208,925,819,614,629,174,706,176 bytes
hipotética 1 Geopbyte (GB)	1024 BB	1,237,940,039,285,380,274,899,124,224 bytes



Lenguaje humano:

Facultad del ser humano de expresarse y comunicarse con los demás a través del sonido articulado o de otros sistemas de signos. (<https://dle.rae.es/lenguaje?m=form>)

En resumen, es un medio, una herramienta de comunicación. Los lenguajes de programación son métodos de comunicación con un procesador CPU y/o GPU

Un circuito electrónico permite el traspaso de información en estados digitales binarios donde cargado eléctricamente es 1 o True y sin carga 0 o false. También analíticamente se puede transmitirse información es valores de escala entre un valor base y un máximo.

Por ejemplo, si la base es 0 y el máximo es 255 y se envía un 128 estaría mandando un estado de 50% la escala establecida.

Entre la comunicación necesaria para estos manejos de compuertas, datos y demás procesos electrónicos básicos y el de comunicación habitual entre los humanos hay una gran diferencia.

Estas diferencias se zanjan con distintos lenguajes que permiten la comunicación entre el humano y los microprocesadores.

Los lenguajes cercanos a los procesos electrónicos se denominan de bajo nivel y aquellos que están más cerca del lenguaje humano se denomina lenguajes de alto nivel.

La comunicación en un lenguaje de bajo nivel es muy rápida ya que es leída por el hardware y ejecutada sin intermediarios, también son realmente muy complicadas para ser escritas y desarrollados por un humano.

Por otro lado, si escribimos en un lenguaje de alto nivel, cercano al humano, se debe procesar mucho esta comunicación para poder ser ejecutada por un microprocesador lo que es mucho más lenta en la ejecución.

Para poder comunicar dos entidades (humanos, animales, computadoras, etc.) se requieren que ambas puedan comprender un conjunto de sonidos o símbolos para intercambiar información. Si es escrito tenemos un alfabeto como en inglés, español, kanji en japonés, hanzi (Hansi - 汉斯 – Hānsī) chinos, números indo arábigos, romanos, 0 y 1 en binarios.

Una vez establecido el alfabeto en común para que las entidades se entiendan se necesitan de un léxico o un diccionario que establezca el significado de un conjunto de sonidos o símbolos como ideas, órdenes, palabras, oraciones, etc.

Posteriormente al tener un alfabeto y un léxico en común debemos establecer una sintaxis, reglas para que la sucesión de palabras se haga de forma que se interprete el significado. Este conjunto de reglas utilizadas para determinar si una sucesión de palabras forma, o no, una oración válida.

Por último hay un conjunto de reglas semánticas para que las entidades puedan dar el mismo significado a una frase o línea de comando.

Si una entidad recibe una frase o sentencia y no puede interpretarla o lo hace en otro sentido al que la emitió la comunicación no fue exitosa. No se realizó la comunicación, alguien emitió un información u órdenes y el receptor no las entendió por lo que no las ejecutó



Debajo encontrarás información del importantísimo rol del lenguaje Python en programación y porque este curso se dicta en él.

Iniciaremos con un paradigma funcional y luego nos introduciremos en el paradigma de programación orientada a objetos (P.O.O.).

Instalación de Python Consola - power shell "CLI-Command Line Interface" (Interfaz de Línea de Comandos). – "GUI Graphical User Interfac"

- Un lenguaje de programación es un conjunto de reglas, símbolos y convenciones utilizados para escribir programas de computadora. Estos lenguajes permiten a los programadores comunicarse con las computadoras y darles instrucciones para realizar tareas específicas.
- Sintaxis y Gramática Específica: Los lenguajes de programación tienen una sintaxis y gramática específicas que deben seguirse para escribir código válido. Cada lenguaje tiene sus propias reglas y convenciones para estructurar instrucciones
- Capacidad de Expresión Lógica y Algorítmica: Un lenguaje de programación debe permitir la expresión de lógica y algoritmos para resolver problemas. Debe proporcionar estructuras para la toma de decisiones, la repetición de tareas y la manipulación de datos.
- Compilación o Interpretación: Los programas escritos en un lenguaje de programación pueden ser compilados o interpretados. La compilación implica traducir el código fuente a un formato ejecutable antes de la ejecución, mientras que la interpretación traduce y ejecuta el código línea por línea durante la ejecución.
- Uso para Desarrollar Software: Un lenguaje de programación se considera como tal si se utiliza para desarrollar software. Esto puede incluir aplicaciones de escritorio, sitios web, sistemas operativos, scripts, entre otros.
- Abstracción de Bajo Nivel o Alto Nivel: Los lenguajes de programación pueden ser de bajo nivel (más cercanos a la arquitectura de la máquina) o de alto nivel (más abstractos y fáciles de entender para los humanos). Ambos tipos son considerados lenguajes de programación.

¿Cómo funciona Python?

El proceso desde el código fuente en Python hasta la ejecución involucra varios pasos, incluyendo la compilación a bytecode y, opcionalmente, la generación de código de máquina nativo.

Compilación del código Fuente:

El código fuente en Python se compila a bytecode por el compilador de Python. Este bytecode no es código de máquina específico para el procesador de la computadora, sino una serie de instrucciones en un formato específico para la máquina virtual de Python.

Interpretación o Compilación Just-In-Time (JIT):

El bytecode se ejecuta a través de la máquina virtual de Python, que es un programa que se encarga de interpretar y ejecutar el bytecode. En algunos casos, puede haber una compilación Just-In-Time (JIT), donde partes del bytecode se traducen a código de



máquina nativo para mejorar el rendimiento.

Ejecución por el Microprocesador:

Si hay JIT, el código de máquina nativo generado se ejecuta directamente en el microprocesador. En caso contrario, la máquina virtual interpreta y ejecuta las instrucciones del bytecode, utilizando el conjunto de instrucciones del procesador subyacente.

En resumen, el microprocesador ejecuta el código de máquina nativo generado por una máquina virtual o interpreta directamente las instrucciones de bytecode según lo implementado por la máquina virtual del lenguaje de programación. Este proceso de ejecución es específico para cada lenguaje y su máquina virtual correspondiente.

código Fuente en Python:

Tienes un archivo con código fuente escrito en Python, que tiene la extensión .py.

Compilación a Bytecode:

- El bytecode generado por un intérprete de lenguaje de programación (como Python) no es directamente ejecutable por un microprocesador.
- Cuando ejecutas un script o importas un módulo, el intérprete de Python compila el código fuente a bytecode. Esto se realiza mediante el compilador de Python.
- El bytecode es una representación intermedia específica del intérprete y necesita ser interpretado o traducido por una máquina virtual específica de ese lenguaje.
- Esta representación intermedia del código fuente y se almacena en archivos con extensión .pyc (o en la memoria caché como parte del proceso de importación).

Máquina Virtual de Python (PVM):

La Máquina Virtual de Python (Python Virtual Machine - PVM) es responsable de interpretar y ejecutar el bytecode generado. La PVM es una parte esencial del intérprete de Python.

JIT Compilation (Opcional):

En algunos casos, puede haber una compilación Just-In-Time (JIT) donde partes del bytecode se traducen a código de máquina nativo para mejorar el rendimiento. No todas las implementaciones de Python utilizan JIT.

Ejecución del código Nativo o intérprete de Bytecode:

- Si se utiliza JIT, se ejecuta el código de máquina nativo generado directamente en el microprocesador.
- Si no hay JIT, la PVM interpreta y ejecuta las instrucciones del bytecode directamente.

En resumen, el proceso puede visualizarse como:

código Fuente (.py) -> Compilación a Bytecode (.pyc) -> PVM (Interpretación/Posiblemente JIT)

Es importante tener en cuenta que diferentes implementaciones de Python pueden tener enfoques ligeramente diferentes en este proceso. Por ejemplo, CPython es la implementación de referencia y utiliza principalmente la interpretación, mientras que Jython y IronPython están diseñados para ejecutarse en la máquina virtual Java y la plataforma .NET, respectivamente. Estas implementaciones pueden tener enfoques específicos para la compilación y ejecución



Para pensar



Regla de tres

1) Si una PC AT 80286 tenía 640 KB o 1 MB, ¿cuánta memoria tiene en su celular?

Calcular que cantidad de pc se deberían conformar un centro de cómputos con pc XT (8086) o AT 286 (80286) para igualar la capacidad de memoria de tu celular.

2) Compara las características de la PC que usas con las que tenía el computador del Apolo que llegó a la Luna.

El computador utilizado en las misiones Apolo que llevaron al ser humano a la Luna en la década de 1960 y principios de la década de 1970 estaba ubicado en la nave espacial y se conocía como la "Computadora de Orientación y Control" (AGC, por sus siglas en inglés).

Características de la AGC:

Memoria:

Memoria de Acceso Aleatorio (RAM): 4 KB (kilobytes).

Memoria de Lectura Única (ROM): 36 KB. Contenía el software de vuelo básico y las rutinas de control.

Velocidad de Procesador: La velocidad de reloj de la computadora del Apolo era de aproximadamente 2 MHz (2,048 megahercios estándar).

Tamaño y Peso: La computadora AGC era físicamente pequeña y liviana en comparación con las computadoras modernas. Tenía dimensiones de aproximadamente 24 x 32 x 16 centímetros y pesaba alrededor de 32 kilogramos.

Arquitectura y Sistema Operativo:

La AGC utilizaba una arquitectura de 16 bits y funcionaba con un sistema operativo diseñado específicamente para las misiones Apolo.

Interfaz de Entrada/Salida:

La interfaz de entrada/salida incluía teclados y pantallas para que los astronautas ingresaran comandos y recibieran información.

3) Una foto de una cámara de celular de 8 megapíxeles en formato JPEG con una calidad de compresión típica podría ocupar alrededor de 2 a 4 megabytes.

Calcular cuántos disquetes 5.25" de doble cara, baja densidad (360 KB) o doble densidad (1.2 MB), así como cuántos disquetes de 3.5" de doble cara, baja densidad (720 KB) o doble densidad (1.44 MB), se necesitan para cargar una imagen de 8 megapíxeles.

La cámara del telescopio James Webb nos ha enviado una fotografía (por ejemplo, como suma de 1.000 imágenes individuales) en su formato original tiene 150 megapíxeles.

Disquettes	Baja densidad	Alta densidad
5,25"	8mp= 150mp=	8mp= 150mp=
3,5"	8mp= 150mp=	8mp= 150mp=

4) Relacionar el tamaño de un archivo .txt o .py con los sistemas de respaldo de Microsoft, AWS, Google, IBM, etc. (exabytes, zettabytes o yottabytes).



A) Conceptos Básicos que todo programador debe conocer para ir a una entrevista o trabajar en grupo:

¿Qué es un IDE?

¿Qué es un SDK?

¿Qué es un framework?

¿Qué es una API?

Una Interfaz de Programación de Aplicaciones (API, por sus siglas en inglés) es un conjunto de reglas y definiciones que permite que diferentes aplicaciones se comuniquen entre sí. Facilita la interacción entre software al definir cómo los componentes deben interactuar.

¿Es verdadero o falso afirmar que Python se puede escribir en un block de notas? ¿Por qué?

¿Qué diferencias hay entre ellos y por qué son necesarios?

¿Qué es Backend (server-side) y qué lugar tienen C, JavaScript y Python?

¿Qué es Frontend (user-side) y qué lugar tienen C, JavaScript y Python?

¿Qué hardware y software usarías en cada lado?

Si te piden hacer un código para celulares o tablets -App - (Android, IO-MAC) ¿Sería para Backend o Frontend?

¿Y en servers sería para Backend o Frontend?

A.4) ¿Qué hace un Full Stack Developer y un Rock Star Developer?

B) ¿Qué es una librería, para que se usan?

C) En tus palabras

¿Qué es un algoritmo en programación?

¿Qué es un programa?

¿cómo elegís que lenguaje utilizar si alguien te pide un programa?

¿Qué es el control de versiones y por qué es importante en el desarrollo de software?

¿Cuál es la diferencia entre compilación y ejecución en el contexto de la programación?

Explica el concepto de variables y su importancia en la programación.

¿Qué es un bucle (loop) y por qué es útil en programación?

Explique la diferencia entre una función y un método en programación.

¿Qué significa el concepto de "scope" (ámbito) en programación y cómo afecta a las variables?

¿Cuál es la diferencia entre una base de datos SQL y NoSQL? Dé ejemplos de cada uno.

¿Cuándo se debe utilizar la programación orientada a objetos y cuáles son sus principios fundamentales?

¿Qué es la recursividad y cuándo se debe utilizar en programación?

Explique la diferencia entre un error de sintaxis y un error de lógica en programación.



¿Qué es la virtualización y cómo se utiliza en el desarrollo de software?

¿Cuál es la importancia de las pruebas unitarias en el desarrollo de software?

¿Qué es la complejidad temporal (Big O) y cómo afecta la eficiencia del algoritmo?

Explique el concepto de concurrencia y paralelismo en programación.



Objetos en Python:

¿Qué es un objeto en Python?

Simplemente Todo. Todo en Python son objetos

Los números, reales o float, binarios, hexadecimales, octales, los strings un carácter o cadenas de estos, las funciones. Los objetos tienen identidad, tipo y valor. La identidad es única y se refiere al objeto en sí, el tipo indica la clase a la que pertenece, y el valor es la información que el objeto almacena. La orientación a objetos en Python es un paradigma fundamental para la programación en este lenguaje, proporcionando un enfoque poderoso y flexible para la organización y estructuración del código.

Ahora en difícil:

Un objeto es una instancia de una clase. Una clase es una estructura que define un conjunto de características (atributos) y comportamientos (métodos) comunes a todos los objetos que se crean a partir de ella. Los objetos son instancias específicas de estas clases y representan entidades concretas o conceptuales en el programa.

En Python, un objeto es una instancia de una clase. Las clases definen características y comportamientos comunes para los objetos. Todo en Python, ya sean números, cadenas, funciones, etc., son objetos. Cada objeto tiene una identidad única, un tipo que indica la clase a la que pertenece y un valor que representa la información que almacena. La orientación a objetos en Python es un enfoque fundamental que proporciona flexibilidad y organización en el desarrollo de programas.

Apuntes (para más adelante):

Conceptos clave relacionados con objetos en Python:

Clase:

Una clase es un plano o un modelo para la creación de objetos. Define atributos y métodos que los objetos creados a partir de ella heredarán.

Una clase es como un "molde" que se utiliza para construir objetos. Contiene atributos y métodos que definen la estructura y comportamiento de los objetos. .

Instancia:

Una instancia es un objeto específico creado a partir de una clase.

Cada vez que creas un objeto a partir de una clase, estás creando una instancia de esa clase.

Cada instancia tiene su propia identidad y estado único.

Objeto:

Un objeto es una instancia particular de una clase. Es una entidad única que tiene sus propios atributos y puede realizar acciones específicas definidas por los métodos de su clase. Representa una entidad concreta o conceptual en el programa.

Atributo:

Un atributo es una característica o propiedad de un objeto que almacena información sobre el objeto. Puede ser cualquier tipo de dato, como números, cadenas, u otros objetos. Los atributos representan el estado de un objeto.

Método:

Un método es una función asociada a una clase que define el comportamiento de los objetos creados a partir de esa clase. Los métodos son acciones que un objeto puede realizar y pueden interactuar con los atributos del objeto.

Herencia:



La herencia es un concepto que permite a una clase heredar atributos y métodos de otra clase (otro constructor de normas) . Esto facilita la reutilización de código y la organización de las clases en una jerarquía.

Polimorfismo:

El polimorfismo permite que objetos de diferentes clases respondan al mismo nombre o etiqueta de método de objetos provenientes de clases distintas. Puede haber múltiples implementaciones para el mismo nombre de método.

Abstracción:

La abstracción consiste en simplificar la complejidad al ocultar detalles innecesarios y destacando la funcionalidad esencial.

Generalidades de los objetos Python:

Tipos de Datos Básicos:

int :(Números enteros) Este tipo de dato representa números enteros, es decir, números sin decimales. Se utiliza para modelar cantidades completas sin fracciones.

float: (Números de punto flotante) Representa números reales con decimales. Es adecuado para modelar cantidades que pueden tener fracciones o decimales.

str: (Cadenas de texto) Este tipo de dato representa secuencias de caracteres. Las cadenas son utilizadas para representar texto y son inmutables.

bool: (Valores booleanos - True o False) Representa valores de verdad o falsedad. Es esencial en lógica booleana y se utiliza para la toma de decisiones en programación.

NoneType: Este tipo de dato tiene un solo valor, None, que se utiliza para representar la ausencia de un valor o la falta de definición.

Colecciones de Datos:

list: (Listas) Representa una secuencia ordenada y modificable de elementos. Se puede acceder a los elementos de la lista mediante índices.

tuple: (Tuplas) Similar a las listas, pero son inmutables, lo que significa que no se pueden modificar después de la creación. Se utilizan para datos que no deben cambiar.

set: (Conjuntos) Representa una colección no ordenada de elementos únicos. Útil para operaciones de conjuntos como unión, intersección y diferencia.

dict: (Diccionarios) Representa una colección de pares clave-valor. Permite acceder a los valores mediante claves, lo que facilita la búsqueda eficiente.

Secuencias:

str: Además de representar texto, las cadenas también pueden ser vistas como secuencias de caracteres.

list y tuple: Ambos son tipos de secuencias, donde los elementos están ordenados y se accede a ellos mediante índices.

Funciones: (function)

Funciones y métodos pertenecen al tipo function.

'**builtin function or method'** funciones preconstituidas por Python.

A abs() aiter() all() anext() any() ascii()	C callable() chr() classmethod ()	E enumerate() eval() exec()	H hasattr() hash() help() hex()	L len() list() locals()	O object() oct() open() ord()	R range() repr() reversed() round()	T tuple() type()
B	D	F filter() float() format()	I id() input()	M map() max() memoryview()	P pow() print()	S set() setattr()	V vars() zip()



<code>bin()</code>	<code>delattr()</code>	<code>frozenset()</code>	<code>int()</code>	<code>)</code>	<code>property()</code>	<code>slice()</code>	<code>_import_(</code>
<code>bool()</code>	<code>dict()</code>	<code>G</code>	<code>isinstance()</code>	<code>_min()</code>		<code>sorted()</code>	<code>_</code>
<code>breakpoint()</code>	<code>dir()</code>	<code>getattr()</code>	<code>)</code>	<code>N</code>		<code>staticmetho</code>	<code>d()</code>
<code>)</code>	<code>divmod()</code>	<code>globals()</code>	<code>_issubclass()</code>	<code>next()</code>		<code>str()</code>	<code>sum()</code>
<code>bytearray()</code>			<code>)</code>			<code>super()</code>	
<code>bytes()</code>			<code>_iter()</code>				

funciones propias **def return**

Clases y Objetos:

Puedes definir tus propias clases y crear instancias de esas clases.

Módulos y Paquetes:

Módulos y paquetes son también tipos de objetos que contienen código Python.

Otros:

`file`: Objetos relacionados con archivos (por ejemplo, devueltos por la función `open()`).

`type`: Tipo de un objeto. Iteradores y Generadores:

`iter`: Objeto que implementa el protocolo de iterador.

`generator`: Objeto que produce una secuencia de valores usando la palabra clave `yield`.

Operadores y Expresiones:

Los operadores y expresiones no son objetos en sí mismos, pero actúan sobre objetos para realizar operaciones.

Context Managers:

Objetos que se utilizan en una declaración `with` para gestionar recursos, como archivos (`open()` actúa como un context manager).



Índice Modulo 1

Salida de datos -operadores de asignación - literales - Expresiones - objetos variables - tipos de datos

- ¿Qué es un lenguaje?
- ¿Qué es el Mundo IT?. Información de Python en el mercadoIT
- ¿Qué es un algoritmo? ¿Qué es un programa?
- Instalación de Python intérpreter (junto al docente para personalizarla)
 - Organización Python.org (intérprete) y comunidad Python
- ¿Qué es un IDE?
- Selección e Instalación de uno o más IDEs
 - Instalación del IDE en tu propio pc.
 - Uso de un IDE en línea
- Consola - power shell – GUI
- Python en los sistemas operativos (Linux, MacOS)
- Algunos ejemplos de Python en el mercado IT
- Lenguajes complicados e interpretados.
 - Ventajas y desventajas de cada una.
 - Python y su posibilidad de trabajar como interpretado y poder ser compilador

Módulo 1:

Lenguaje humano: Facultad del ser humano de expresarse y comunicarse con los demás a través del sonido articulado o de otros sistemas de signos. (<https://dle.rae.es/lenguaje?m=form>)

En resumen, es un medio, una herramienta de comunicación. Los lenguajes de programación son métodos de comunicación con un procesador CPU y/o GPU

Un circuito electrónico permite el traspaso de información en estados digitales binarios donde cargado eléctricamente es 1 o True y sin carga 0 o false. También analíticamente se puede transmitir información es valores de escala entre un valor base y un máximo.

Por ejemplo, si la base es 0 y el máximo es 255 y se envía un 128 estaría mandando un estado de 50% la escala establecida.

Entre la comunicación necesaria para estos manejos de compuertas, datos y demás procesos electrónicos básicos y el de comunicación habitual entre los humanos hay una gran diferencia.

Estas diferencias se zanján con distintos lenguajes que permiten la comunicación entre el humano y los microprocesadores

Los lenguajes cercanos a los procesos electrónicos se denominan de bajo nivel y aquellos que están más cerca del lenguaje humano se denomina lenguajes de alto nivel

La comunicación en un lenguaje de bajo nivel es muy rápida ya que es leída por el hardware y ejecutada sin intermediarios, también son realmente muy complicadas para ser escritas y desarrollados por un humano.

Por otro lado, si escribimos en un lenguaje de alto nivel, cercano al humano, se debe procesar mucho esta comunicación para poder ser ejecutada por un microprocesador lo que es mucho más lento en la ejecución.

Para poder comunicar dos entidades (humanos, animales, computadoras, etc.) se requieren que ambas puedan comprender un conjunto de sonidos o símbolos para intercambiar información. Si es escrito tenemos un alfabeto como en inglés, español, kanji en japonés, hans (Hansi - 汉斯 – Hānsī) chinos, números indo arábigos, romanos, 0



y 1 en binarios.

Una vez establecido el alfabeto en común para que las entidades se entiendan se necesitan de un léxico o un diccionario que establezca el significado de un conjunto de sonidos o símbolos como ideas, órdenes, palabras, oraciones, etc.

Posteriormente al tener un alfabeto y un léxico en común debemos establecer una sintaxis, reglas para que la sucesión de palabras se haga de forma que se intérprete el significado. Este conjunto de reglas utilizadas para determinar si una sucesión de palabras forma, o no, una oración válida.

Por último hay un conjunto de reglas semánticas para que las entidades puedan dar el mismo significado a una frase o línea de comando.

Si una entidad recibe una frase o sentencia y no puede interpretarla o lo hace en otro sentido al que la emitió la comunicación no fue exitosa. No se realizó la comunicación, alguien emitió una información u órdenes y el receptor no las entendió por lo que no las ejecutó

Debajo encontrarás información del importantísimo rol del lenguaje Python en programación y porque este curso se dicta en él.

Iniciaremos con un paradigma funcional y luego nos introduciremos en el paradigma de programación orientada a objetos (POO).

Antes que nada, debemos aclarar algo para los que vienen de otro lenguaje.

En Python, todo se considera un objeto. Un objeto es una entidad que combina datos (**atributos**) y funciones (**métodos**) relacionadas que operan en esos datos. Es la base del paradigma de programación orientada a objetos (POO) en Python.

Conceptos clave relacionados con los objetos en Python:

Atributos:

Los atributos son las variables que pertenecen a un objeto. Pueden ser variables de instancia (definidas dentro de un método) o variables de Módulo (definidas en el Módulo pero fuera de los métodos). Cada instancia de una Módulo puede tener diferentes valores para los atributos.

Métodos:

Los métodos son las funciones asociadas a un módulo. Pueden realizar operaciones en los datos del objeto y pueden acceder y modificar los atributos. Los métodos pueden ser llamados en las instancias de la Módulo y actúan en contexto a los datos específicos de la instancia.

Nota: En adelante utilizaremos y reforzaremos el término objeto. Ej Objeto variable tipo string.

Los objetos son las entidades fundamentales que encapsulan datos y funcionalidad relacionada. Los módulos definen la estructura y el comportamiento de los objetos, mientras que las instancias son objetos específicos creados a partir de una módulo. La programación orientada a objetos en Python facilita la organización, la reutilización y la modularidad del código.

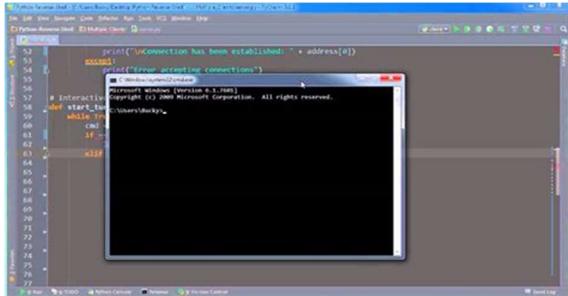
El curso al desarrollarse en Python requiere que se instale el intérprete y un IDE.

Tenga en cuenta que cada alumno puede estar en Linux, en alguna de sus varias distribuciones, Windows 32 o 64 y en versiones 7,8,10 u 11 o MacOS con procesador Intel o ARM.

- desarrollo en consola "CLI", "Command Line Interface" (Interfaz de línea de Comandos).

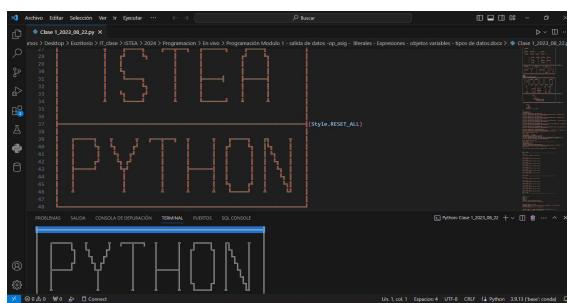
Consola – CLI:

La "consola" se refiere a una interfaz de línea de comandos



(CLI) que permite a los usuarios interactuar con un sistema operativo o una aplicación mediante comandos de texto.

Definición: La consola es una interfaz de texto en la que los usuarios ingresan comandos para realizar acciones en un sistema operativo.



Características:

Se ejecuta en un entorno de texto sin gráficos.

Los comandos se ingresan mediante teclado y se muestran como texto.

Es eficiente para usuarios experimentados y tareas automatizadas mediante scripts.

Puede proporcionar información detallada y salida de texto.

Ejemplos: La Terminal en sistemas Unix/Linux, el Command Prompt en Windows.

PowerShell:

Definición: PowerShell es una interfaz de línea de comandos avanzada y un marco de scripting desarrollado por Microsoft.

Características:

Combina la funcionalidad de la consola tradicional con características de scripting avanzadas y objetos.

Utiliza cmdlets (comandos especializados) para realizar acciones específicas.

Ofrece acceso a funciones del sistema operativo y a productos de Microsoft.

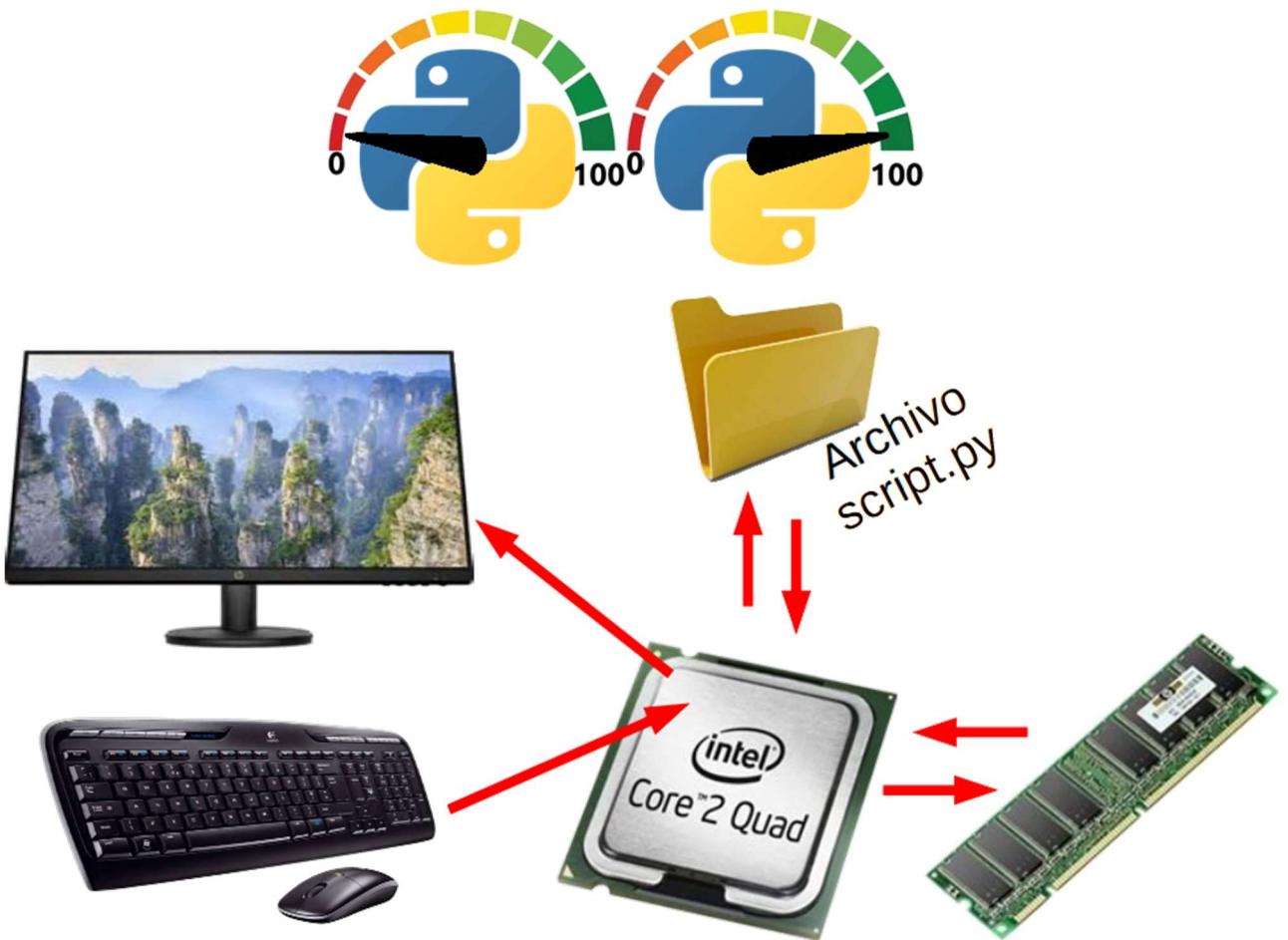
Es altamente extensible y permite la automatización de tareas complejas.

Ejemplos: PowerShell en sistemas Windows.

Tenga en cuenta que normalmente cada uno deberá grabar (con nombre y extensión – py) los scripts para poder ejecutar.



En caso de usar un ide en línea y que no desee registrarse, una vez finalizado el script podrá copiarlo y pegarlo en un block de notas para que el alumno tenga los scripts guardados
Esta información paso a paso se encuentra en la guía de instalación que acompaña este archivo.



```
print("Este texto saldrá en consola")
```

Este texto saldrá en consola



```
print ("hola Curso Python")
```

Si se instaló en la pc se podrá encontrar es necesario compilar o construir y luego ejecutar el script

Depende del IDE emerge consola o se ve el resultado en la barra de powershell

En línea veremos o particiones de la pantalla o emergentes o demás salidas, pero en todas se verán

```
hola Curso Python
```

Siguiente paso es que podamos escribir varios literales y ver la salida del código.

Se deberá dar una explicación del proceso interpretación.

Uso de comillas en Python

' Comillas simple

" Comillas doble

Al inicio y final de una cadena de texto (string)

Son utilizables uno u otra, pero si se abre con un tipo se debe cerrar con el mismo

'Hola mundo IT curso Python'

"Hola mundo IT curso Python"

** **

inicio fin

Se pueden usar uno dentro de otra

'Hola "mundo IT" curso Python'

"Hola 'mundo IT' curso Python"

** ** internas ** **

inicio fin

Tenga en cuenta que en inglés las comillas simples se suelen usar muy habitualmente como apóstrofe

'Tom's cat'

** ** **

inicio fin inicio.....

"I don't want eat this candy"

** ** interna **

inicio fin

''' Comillas simple x 3

""" Comillas doble x 3

Esto habilita usar strings de más de una línea

"""" esto es

un

texto

de varias

líneas """

Uso de secuencias de escape más usadas en Python

\n newline (salto de línea)

\t tab (tabulación)

\\" Backslash (barra)



```
\' Comillas simple  
\\" Comillas doble
```

```
# \n para generar una nueva línea entre el texto anterior y el posterior en la salida  
print ("hola Curso \n Python")
```

```
# Salida esperada por consola  
hola Curso  
Python
```

```
# \n para generar una nueva línea entre el texto anterior y el posterior en la salida  
print ("\"\"\"hola Curso  
Python\"\"")
```

```
# Salida esperada por consola  
hola Curso  
Python
```

```
# \t para tabular (de 4 a 8 espacios) entre el texto anterior y el posterior en la salida  
print ("hola Curso \t Python")
```

```
# Salida esperada por consola  
hola Curso      Python
```

```
# \t para tabular (de 4 a 8 espacios) entre el texto anterior y el posterior en la salida  
print ("hola Curso      Python")
```

```
# Salida esperada por consola  
hola Curso      Python
```

```
#     \"  imprimir comillas  
#     \'  imprimir comillas  
print ("hola Curso \"\\" Python\\\" \"")
```



```
# Salida esperada por consola  
hola Curso "Python"
```

```
#     \"  imprimir comillas  
#     \'  imprimir comillas  
print ('hola Curso \"Python\"')  
print ("hola Curso 'Python'")
```



```
# Salida esperada por consola  
hola Curso "Python"  
hola Curso 'Python'
```

\a que el alumno investigue que salida tiene, bell un sonido



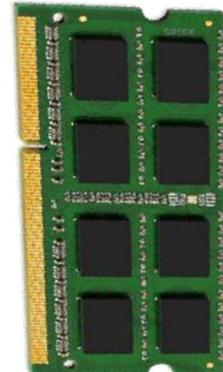
La primer función en invocar será para poder sacar información desde el programa al usuario

Conceptos de compilador, lenguajes, niveles, consola, salida de información. No dejamos más que la idea de trabajar son lo echo sin avanzar en nuevos temas para esta práctica.

Si quieren esbozar los conceptos variables y constantes como lugares en memoria donde se guardan datos.

```
print ("hola curso, ¿como va todo?")
```

hola curso, ¿como va todo?



```
variable = "Bienvenidos a Python"
```

```
print (variable)
```

Bienvenidos a Python

```
print ("Hola todos. ", variable, "¿como va todo?")
```

Hola todos. Bienvenidos a Python ¿como va todo?

```
print (f"Hello todos. { variable } ¿como va todo?")
```

Hola todos. Bienvenidos a Python ¿como va todo?

- Concepto de funciones Built-in

Hay palabras reservadas (fuertes) que no deben ser usadas como identificadores o nombres de variables y otras (débiles) que no se aconseja su uso. (<https://docs.python.org/3/library/functions.html>)

<https://docs.python.org/es/3.12/library/functions.html>

Reservadas fuertes:

and, del, from, not, while, as, elif, global, or, with, assert, else, if, pass, yield, break, except, import, print, class, exec, in, raise, continue, finally, is, return, def, for, lambda, try

Reservadas débiles:

all, any, input, elif, int, float, str, list, tuple, dict, set, frozenset, except, open, close, read, write, nombres de librerías / bibliotecas que estés usando, etc.

Clic sobre cada función para acceder a una descripción a python.org

A	E	L	R
---	---	---	---



abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
any()			round()
anext()	F	M	
ascii()	filter()	map()	S
B	float()	max()	set()
bin()	format()	memoryview()	setattr()
bool()	frozenset()	min()	slice()
breakpoint()	G	N	sorted()
bytearray()	getattr()	next()	staticmethod()
bytes()	globals()		str()
C	H	O	sum()
callable()	hasattr()	object()	super()
chr()	hash()	oct()	T
classmethod()	help()	open()	tuple()
compile()	hex()	ord()	type()
complex()			
D	I	P	V
delattr()	id()	pow()	vars()
dict()	input()	print()	Z
dir()	int()	property()	zip()
divmod()	isinstance()		-
	issubclass()		import __()
	iter()		



Módulo 2 - Operadores:

- operadores de relacionados
- operador ternario
- if else elif
- Operadores Lógicos o booleanos en condicionales:
- match case

Operadores de comparación o relacionales en condicionales:

Estos operadores se utilizan para comparar valores numéricos y de caracteres.

Los operadores de comparación en Python son:

==	chequea si dos valores son iguales.
!=	chequea si dos valores son diferentes.
<	chequea si el primer valor es menor que el segundo.
<=	chequea si el primer valor es menor o igual que el segundo.
>	chequea si el primer valor es mayor que el segundo.
>=	chequea si el primer valor es mayor o igual que el segundo.

Expresión lógica u operador ternario

En una línea de código en cuestión evalúa una expresión lógica y utiliza una f-string para imprimir el resultado de esa expresión. La salida será "True" o "False" según las condiciones especificadas en las expresiones lógicas son verdaderas o falsas (según tabla de la verdad).

Expresión lógica: Consiste en una condición, (o más condiciones conectadas mediante el operador lógico and u or).

Uso de f-string: La expresión lógica se incluye dentro de una f-string, que es una cadena de formato en Python (identificada por el prefijo f antes de las comillas). Dentro de la f-string, la expresión lógica está encerrada entre llaves {} y precedida por un signo igual =. Esto indica que el resultado de la expresión se imprimirá como parte de la cadena.

Resultado esperado: La salida esperada es "TRUE" (en mayúsculas). Esto se debe a que la expresión lógica es verdadera en el contexto dado. Si ambas condiciones dentro de la expresión lógica son verdaderas (x no es igual a 0 e y no es menor que x), entonces toda la expresión es verdadera.

```
x = 2
y = x + 2
# Muestra el valor de 'y'
print(f"y={y}")           # Salida: 4
# Salida esperada por consola
y=4
```

```
x = 2
y = x + 2
# Muestra si 'y' es mayor que 'x'
```



```
print(f"{{y>x=}}")      # Salida: True
```

Salida esperada por consola
y>x=True

```
x = 2
y = x + 2
# Muestra si 'x' es mayor o igual que 'y'
print(f"{{x>=y=}}")      # Salida: False
```

Salida esperada por consola
x>=y=False

```
x = 2
y = x + 2
# Muestra si 'x' es igual a 'y-2'
print(f"{{(x==y-2)=}}")    # Salida: True
```

Salida esperada por consola
(x==y-2)= True

```
x = 2
y = x + 2
# Muestra si 'x' no es igual a 0 y 'y' no es menor que 'x'
print(f"{{(x!=0 and not y<x)=}}") # Salida: True
```

Salida esperada por consola
(x!=0 and not y<x)= True

```
# Operador ternario con operadores lógicos 'and' y 'or'
num = 15
print (f"La categoría es {{('Impar y/o negativo', 'Par y positivo')[num % 2 == 0 and num > 0]}}")
```

Salida esperada por consola
La categoría es Impar y/o negativo

```
# Operador ternario con operadores lógicos 'and' y 'or'
num = 16
print (f"La categoría es {{('Impar y/o negativo', 'Par y positivo')[num % 2 == 0 and num > 0]}}")
```

Salida esperada por consola
La categoría es Par y positivo

```
# Operador ternario con operadores lógicos 'and' y 'or'
num = 15
print (f"La categoría es {{('Impar', 'Par')[num % 2 == 0 and num != 0 ]} y {{('negativo',
'Positivo')[num < 0 ]}}")
```



```
# Salida esperada por consola  
La categoría es Impar y negativo
```

```
# Operador ternario con operadores lógicos 'and' y 'or'  
num = 16  
print(f"La categoría es {('Impar', 'Par')[num % 2 == 0 and num != 0]} y {('negativo',  
'Positivo')[num < 0]}")
```

```
# Salida esperada por consola  
La categoría es Par y positivo
```

```
Y si num=0
```

```
# Operador ternario con operador lógico 'not'  
flag = False  
print(f"flag (booleano) {'Esta en falso' * (not flag) or 'Esta en verdadero'}")
```

```
# Salida esperada por consola  
flag (booleano) Esta en falso
```

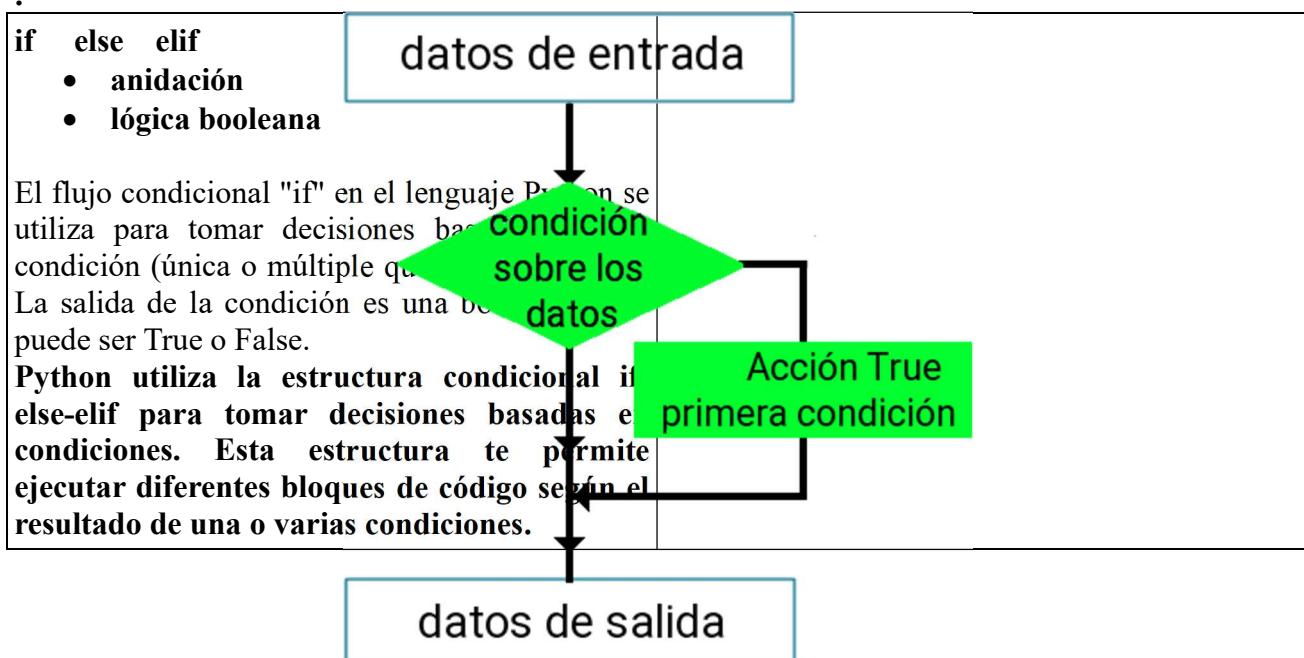


Modificación de flujo condicional if

Hasta ahora el flujo del programa fue de arriba hacia abajo y de izquierda a derecha. En occidente eso es muy normal. Pero hay lenguajes como árabes, hebreo, etc que se escriben de derecha a izquierda. Incluso hay ejemplos de poesía oriental incluso manga donde se escriben en columnas de abajo hacia arriba.

Damos condicionales antes de ingreso de datos para que el alumno no se complique con dos temas al mismo tiempo. Hay muy pocas cosas para hacer con el ingreso de datos sin condiciones. Pero dar condicionales nos permite que sobre este tema pase de ser el programador quien ingresa el dato al usuario. Pero la estructura de filtros que veremos a continuación se mantiene.

En IDE	comentarios	Salida esperada por consola
print (0)	El flujo del programa es de arriba hacia abajo	0
print (1)		1
print (2)		2
print (3)		3
print (4)		4
print (5)		5
print (6)		6
print (7)		7
print (8)		8
print (9)		9
print (10)		10

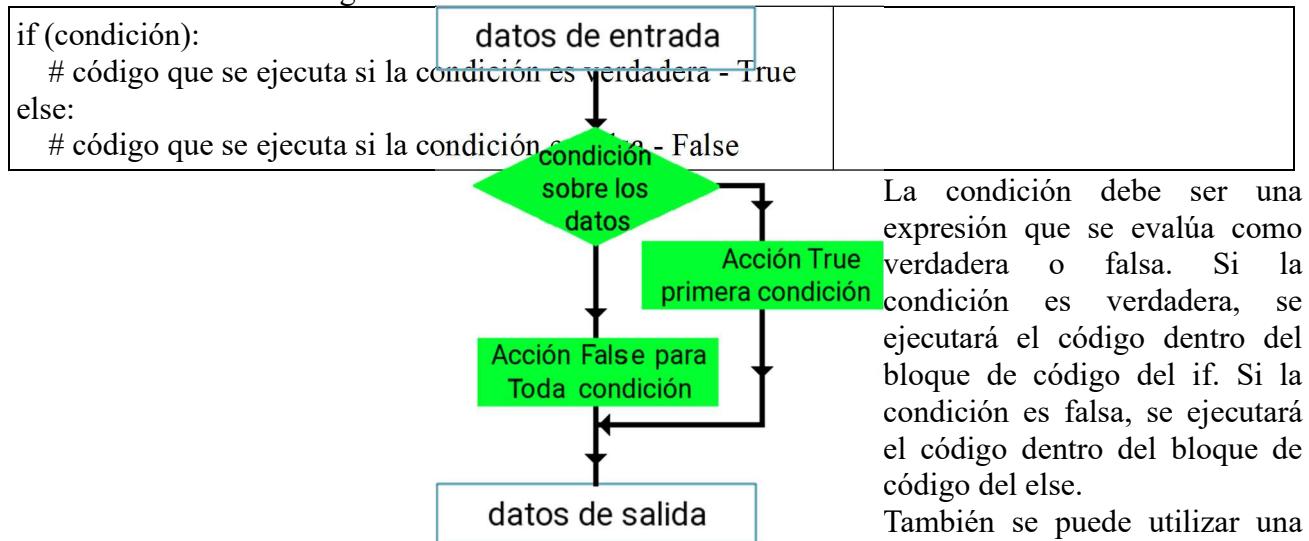




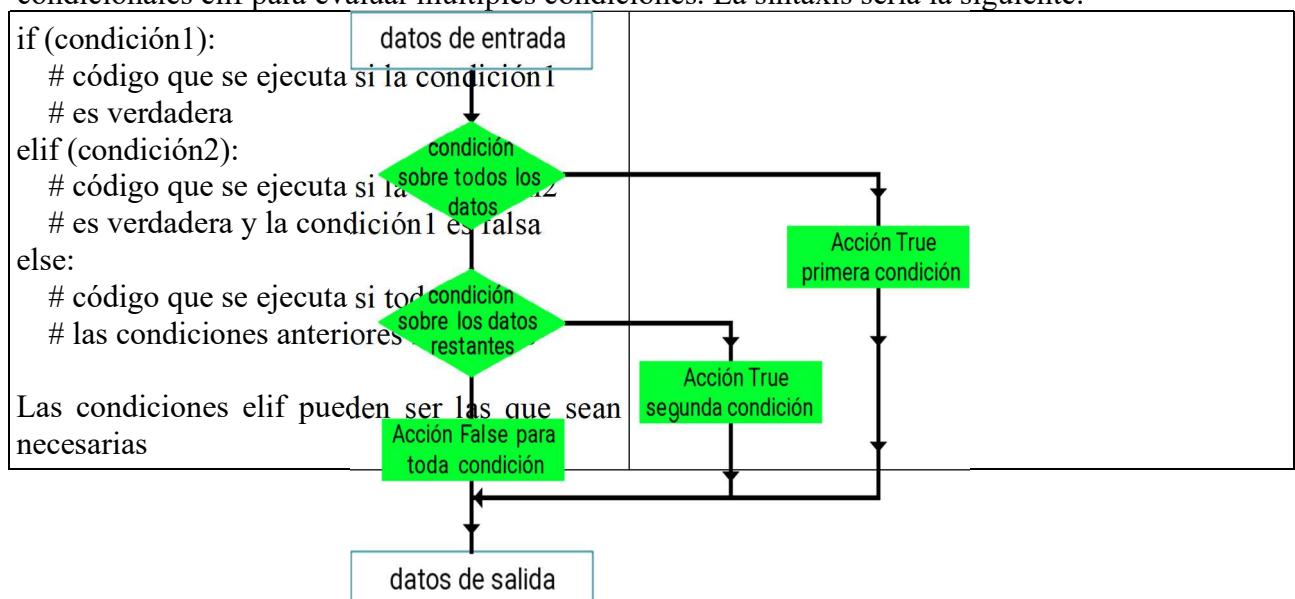
If-else: Permite especificar una acción alternativa en caso de que la condición del "if" no se cumpla. El bloque de código dentro del "else" se ejecutará si la condición del "if" es falsa.

If-elif-else: Se puede encadenar varios bloques "elif" después de un "if" para evaluar múltiples condiciones en secuencia. El primer bloque "elif" cuya condición sea verdadera se ejecutará, o si ninguna condición es verdadera, se ejecutará el bloque "else" final.

La sintaxis básica es la siguiente:



condicionales elif para evaluar múltiples condiciones. La sintaxis sería la siguiente:





En este caso, se evalúa cada condición en orden. Si alguna de las condiciones es verdadera, se ejecutará el código dentro del bloque de código correspondiente y se saldrá de la estructura condicional. Si todas las condiciones son falsas, se ejecutará el código dentro del bloque de código del else final.

Cabe destacar que la estructura condicional if-else puede anidarse para crear decisiones más complejas en el código.

```
nota_1er_p = 9
nota_2do_p = 2
nota_3er_p = 7
nota_4to_p = 6
suma = nota_1er_p + nota_2do_p + nota_3er_p + nota_4to_p
media= suma/4
print (f'la media de las notas es {media}')
if (media <7):#      desde -infinito a 6.999
    print (f'no alcanzó el mínimo de 7 para aprobar el curso')
else:#  todo valor que no haya entrado en la condición if, media debe ser mayor o igual a 7
    print (f"curso aprobado")
```

Salida esperada por consola
la media de las notas es 7.50 -
curso aprobado

con

```
nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 7
nota_4to_p = 6
```

Salida esperada por consola
la media de las notas es 7.50 -
curso aprobado

```
nota_1er_p = 9
nota_2do_p = 2#<-----modificado
nota_3er_p = 7
nota_4to_p = 6
```

Salida esperada por consola
la media de las notas es 6.00 -
no alcanzó el mínimo de 7 para aprobar el curso

```
nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 7
nota_4to_p = 6
suma = nota_1er_p + nota_2do_p + nota_3er_p + nota_4to_p
media= suma/4
```



```
print(f'la media de las notas es {media}')
if (media <7):#      desde -infinito a 6.999
    print(f'no alcanzó el mínimo de 7 para aprobar el curso')
elif (media <9):#  todo valor que no haya entrado en la condición if,
#                  media debe ser mayor o igual a 7 pero debe ser menor a 9
    print(f"curso aprobado")
else:#  todo valor que no haya entrado en la condición if,
#        media debe ser mayor o igual a 9
    print(f" felicitaciones. Al cuadro de honor")
```

con

```
nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 7
nota_4to_p = 6
```

Salida esperada con
la media de las notas es 7.50 -
curso aprobado

con

```
nota_1er_p = 9
nota_2do_p = 2#-----modificado
nota_3er_p = 7
nota_4to_p = 6
```

#Salida esperada por consola
la media de las notas es 6.00 -
no alcanzó el mínimo de 7 para aprobar el curso

con

```
nota_1er_p = 10#-----modificado
nota_2do_p = 9#-----modificado
nota_3er_p = 10#-----modificado
nota_4to_p = 9#-----modificado
```

Salida esperada con
la media de las notas es 9.50 -
felicitaciones. Al cuadro de honor

Podemos generar if anidados o elif para evaluar si tiene que rendir recuperatorio o se lleva la materia entera con parámetros entre 4 y 7 o menor a 4 por ejemplo.
Luego de ver operadores se procede con un ejercicio que abarca el tema en su totalidad y permite que luego el usuario ingrese los valores

Operadores Lógicos o booleanos en condicionales:

Estos operadores se utilizan para combinar expresiones booleanas y producir un resultado booleano. Los operadores lógicos en Python son:

Python	Python NO	Descripción
--------	-----------	-------------



Estándar	Estándar	
and	&	El operador "y" lógico. Devuelve verdadero si ambas expresiones son verdaderas.
or	 	El operador "o" lógico. Devuelve verdadero si al menos una de las expresiones es verdadera.
not	~	El operador "no" lógico. Invierte el valor de la expresión.

```

nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 2
nota_4to_p = 10
suma = nota_1er_p + nota_2do_p + nota_3er_p + nota_4to_p
media= suma/4
print (f"la media de las notas es {media}")
if (media <7):#      desde -infinito a 6.999
    print (f"no alcanzó el mínimo de 7 para aprobar el curso")
elif (nota_1er_p>=4) and (nota_2do_p>=4) and (nota_3er_p>=4) and (nota_4to_p>=4) :
#
#          todo valor que no haya entrado en la condición
#
#          media mayor a 7
#
#          todos los bimestres aprobados
    print (f"curso aprobado")
else:
#
#          media mayor a 7
#
#          al menos un bimestre menos a 4
    print (f"tenes que recuperar un bimestre")

```

con

```

nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 7
nota_4to_p = 6

```

Salida esperada por consola
la media de las notas es 7.50 -
curso aprobado

con

```

nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 2
nota_4to_p = 10

```

Salida esperada por consola
la media de las notas es 7.25 -



tenes que recuperar un bimestre



Menú de opciones múltiple match case

```
optar = input("Ingresa una opción:  
A) para Abrir  
B) para Borrar  
C) para Copiar  
S) para Salir").upper()  
  
match (optar):  
    case 'A':  
        print (f" opción abrir")  
    case 'B':  
        print (f" opción borrar")  
    case 'C':  
        print (f" opción copiar")  
    case 'S':  
        print (f" adiós")  
    case other:  
        print (f" opción no valida")
```

La estructura match case en lenguaje Python se utiliza para realizar selecciones múltiples basadas en el valor de una expresión.

Aquí tienes un ejemplo básico de cómo usar match case:

En este ejemplo, se le pide al usuario que seleccione una opción del 1 al 3. Luego, se utiliza match (optar) para evaluar el valor de optar y ejecutar el bloque de código correspondiente al caso coincidente.

```
optar = input("Ingresa una opción:  
1) para Abrir  
2) para Borrar  
3) para Copiar  
4) para Salir")  
  
match (int(optar)):  
    case 1:  
        print (f" opción abrir")  
    case 2:  
        print (f" opción borrar")  
    case 3:  
        print (f" opción copiar")  
    case 4:  
        print (f" adiós")  
    case other:  
        print (f" opción no valida")
```

Case en Python permite or



```
match entero:  
    case (1) | (2) | (3):
```

Case en python permite colecciones

```
match string:  
    case ("A", "B", "C", "D", "E", "F"):
```

.

La anidación en programación se refiere a la práctica de incluir una estructura dentro de otra. Esto puede aplicarse a varias estructuras de control dentro de otras estructuras de control. La anidación permite organizar y estructurar el código de manera jerárquica, facilitando la comprensión y mantenimiento del código.

```
if x > 0:  
    if y > 0:  
        print("Ambas coordenadas son positivas.")  
    else:  
        print("La coordenada x es positiva, pero la coordenada y no lo es.")  
else:  
    print("La coordenada x no es positiva.")
```



Módulo 3: Objetos – tipos

- Entrada de datos desde el usuario
- Casting

Python.org tiene un consejo directivo que ordena y guía el desarrollo del intérprete Python

Por otro lado, hay una **comunidad de programadores** que aporta y genera reglas para el trabajo en común.

➤ Guardar datos en memoria.

En Python, un objeto es en parte un contenedor que se utiliza para almacenar datos.

Se puede pensar como una caja etiquetada donde puedes guardar diferentes tipos de información, como números, cadenas de texto o listas.

Algunos conceptos clave sobre los objetos variables en Python son los siguientes:

No es necesario declarar explícitamente el tipo de una variable antes de usarla.



Simplemente puedes asignar un valor a una variable utilizando el operador de asignación "=" esto guarda en memoria un dato asociado al nombre del objeto y el intérprete chequea el tipo de dato y determina que tipo de objeto es.

edad = 25# guarda en memoria el valor 25 asignándole una etiqueta con el nombre edad, este objeto sera del tipo int - entero

pi = 3.14159# guarda en memoria el valor 3.14159 asignándole una etiqueta con el nombre pi, este objeto será del tipo float – con punto decimal

Cada objeto tiene un nombre único, dependiendo del tipo de objeto su asociación a un dato en guardado en memoria puede variar en el transcurso del script. Esto permite que el programa pueda acceder y manipular los valores y los tipos durante su ejecución.

Los objetos variables NO son espacios de memoria reservados para almacenar datos.

Por ejemplo, para declarar una variable entera llamada "edad", se utilizaría la siguiente línea de **código**:

Se recomienda el uso de print (f'descripción: {objeto} ')



```
# no se requiere declarar int edad como en C
edad = 25
print (f'La edad guardada es de {edad} años')
```

Salida esperada por consola

La edad guardada es de 25 años

En Python algunos objetos con un dato son de los siguientes tipos:



int:	Representa números enteros. dato = 8
float:	Representa números de punto flotante (números con decimales). Dato = 8.5
str:	Representa una cadena de caracteres. Cada Eslabón es un carácter aunque puede haber strings con un solo carácter e incluso sin ninguno. Siempre se coloca entre comillas (simples, dobles o tres juegos de alguna para multilíneas) dato = "Hola mundo IT" dato = """ esto es un texto de varias líneas """ dato = "T" dato = ""
bool:	Representa valores booleanos (True verdadero o False falso). dato = True dato = False

Formato antiguo:

```
objeto = "hola mundo Python"
print (objeto)

#Salida esperada por consola
hola mundo Python
```

Formato moderno:

objeto imprime el contenido en memoria guardado con la etiqueta objeto

```
objeto = "hola mundo Python"
print (f"el contenido del objeto es {objeto} ")

#Salida esperada por consola
el contenido del objeto es hola mundo Python
```

objeto= imprime el nombre del objeto y el contenido en memoria guardado con la etiqueta objeto

```
objeto = "hola mundo Python"
print (f"el contenido del objeto es {objeto=}")

#Salida esperada por consola
el contenido del objeto es objeto='hola mundo Python'
```

Mas adelante veremos algunos objetos con múltiples datos: listas ('list'), tuplas ('tuple'), diccionarios ('dict'), entre otros.

Después de declarar una variable, se puede asignar un valor utilizando el operador de asignación =

> Operador de asignación igual (=)

Para guardar un dato en memoria debemos previamente haber declarado el nombre de la variable con el propósito que tendrá. Ahora veremos cómo se asigna y que reglas hay



Luego veremos el resto de los operadores de asignación el la Módulo de bucles

`lado_recibe = lado_dato`

En el lado que recibe debemos poner algún nombre de fantasía que represente el dato a ingresar.

Las reglas del lenguaje exijan que inicie con letras, puede usarse el signo guin de abajo (_) y números pero nunca al inicio.

La comunidad te sugiere que los nombres se relacionen con el dato a guardar.

En Python, una variable es un contenedor que se utiliza para almacenar datos. Puedes pensar en una variable como una caja etiquetada donde puedes guardar diferentes tipos de información, como números, cadenas de texto o listas.

> Variables y Constantes en Python Mmmmmmm.....

Algunos conceptos clave sobre los objetos variables en Python son los siguientes:

> Declaración de objetos variables: En Python, NO es necesario declarar explícitamente el tipo de una variable antes de usarla.

> Asignación de valores: Para asignar un valor a una variable, utilizas el operador de asignación "=" seguido del valor que deseas asignar.

```
edad = 25
nombre = "Juan"
```

> Nombres de objetos:

Los nombres de los objetos en Python deben seguir ciertas reglas para el intérprete Python

Solo pueden contener letras, dígitos y guiones bajos.

Deben comenzar con una letra o un guion bajo (_)

Python distingue entre mayúsculas y minúsculas en los nombres de los objetos.

Los nombres de los objetos en Python deben seguir ciertas reglas para la comunidad Python

Estilo general

- snake_case, donde las palabras se separan con guiones bajos
- minúsculas para objetos mutables, variables, que cambian en el trascurso del script.
- mayúsculas para objetos que si bien pueden ser mutables, variables, que cambian en el trascurso del script el programador marca en caracteres mayúsculas para marcar su contenido como constante y que no debería ser modificado.

Estilo de módulos

- pascal_case Primer carácter en mayúsculas o Capitalize
- Es importante seguir estas convenciones para que el código sea más legible y coherente.

Ejemplos válidos para el intérprete

`nombre = "Pedro"#` Valido para el intérprete Python y la comunidad Python

`nombre_1 = "Pedro"#` Valido para el intérprete Python y la comunidad Python

`nombre_alumno = "Pedro"#` Valido para el intérprete Python y la comunidad Python

`nombreAlumno = "Pedro"#` Valido para el intérprete Python y la comunidad Python

`nombre = "María"#` Valido para el intérprete Python pero NO la comunidad Python

`Nombre = "Pedro"#` Valido para el intérprete Python pero NO la comunidad Python

`xdcfvgb = 9#` Valido para el intérprete Python pero NO la comunidad Python

En Python, por lo general, los valores se pisan, se sobre escriben o la etiqueta o nombre se asigna a otro dato

- Reasignación de valores: Puedes cambiar el valor de una variable simplemente asignándole un nuevo valor. Python permite reasignar una variable a diferentes tipos de datos a lo largo del programa.

```
valor = 25
valor = 30 # Cambio el valor del objeto variable
valor = 3.55 # Cambio el valor y tipo del objeto variable
```



Puedes utilizar objetos variables en expresiones matemáticas o concatenarlas con cadenas de texto utilizando los operadores correspondientes.

```
x = 10
y = 5
suma = x + y
mensaje = "El resultado es: " + str(suma) # modo antiguo cambiando
                                              el entero (10+5=15)
                                              a str(15) = "15"
mensaje = f"El resultado es: {suma}"      # modo actual
```

Los objetos variables en Python te permiten almacenar y manipular datos de manera flexible en tus programas. Puedes usarlas para realizar cálculos, almacenar resultados intermedios, guardar información del usuario y mucho más.

```
nombre = "Juan"
apellido = "Pérez"
nombre_completo = nombre + apellido
print(f"el nombre completo es {nombre_completo} ")
nombre_completo_con_espacio = nombre + " " + apellido
print(f"el nombre completo con espacios es {nombre_completo_con_espacio} ")

#Salida esperada por consola
el nombre completo es JuanPérez
el nombre completo con espacios es Juan Pérez
```

Se recomienda el uso de

```
print(f"descripción: {objeto} ")
print(f"tipo de objeto: {type(objeto)} ")
```

Es importante tener en cuenta que en Python los objetos deben existir antes de que se utilicen en el código. Esto significa no puedo imprimir un objeto antes de que este sea creado. Para crearlo debo asignar al nombre un valor (y por ende un tipo)

En relación a los objetos numéricos int, float o complex no se tendrá en cuenta el tamaño, valor o grado de exactitud o definición de cantidad de decimales.

Noten que dije punto decimal y no coma. En Argentina rige una ley (SiMeLA) sistema métrico legal argentino donde se declara que la parte entera de la decimal se separa por una coma. A los lenguajes de programación no les importa mucho esta ley y para separar la parte decimal de la entera se usa un punto. NO hay separador de miles, sino que se escriben de corrido.

```
cien_mil= 100000;          # no 100.000
PI=3.14159;                # no 3,14159
```

> Ámbitos de los objetos variables:

Las variables en Python tienen un alcance, es decir, la parte del programa donde son accesibles y válidas.

Los ámbitos de los distintos objetos se verá luego de ver funciones propias.

La variable declarada dentro de una función solo es accesible dentro de esa función (a menos que se utilicen mecanismos como global o nonlocal para ampliar su alcance).

Las variables declaradas fuera de una función tienen un alcance global y son accesibles en todo el programa.

> Expresiones:

En el lenguaje de programación Python, los operadores son símbolos que se utilizan para realizar operaciones matemáticas, de comparación y de asignación.

> Operadores aritméticos en Python:

Estos operadores se utilizan para realizar operaciones matemáticas básicas en valores numéricos.



operadores	Funciones estándar entre operandos
+	Suma - adiciona dos operandos.
-	Resta - sustracción al valor del operando de la izquierda el valor del de la derecha. Cambia el signo sobre un único operador.
*	Producto - Multiplicación de dos operandos.
/	Divide el operando de la izquierda por el de la derecha. La salida siempre es un float (genera un casting que veremos más adelante)
//	División entera se obtiene el cociente 'entero' de dividir el operando de la izquierda por el de la derecha.
%	Módulo es el residuo que se obtiene el resto de dividir el entero del operando de la izquierda por el de la derecha.
operadores	Funciones Python entre operandos
**	Potencia - Exponenciación eleva el operando de la izquierda a la potencia del operador del de la derecha.
**(1/n)	Radicación eleva el operando de la izquierda a la potencia de 1 (uno) dividido el valor operador del de la derecha.

- En Python además de los estándar suma(+), resta (-), división estándar (/), división entera (//) , resto de división (%) , multiplicación (*)
- además
 - Potenciación o exponenciación (**) $x^{**}2$ x al cuadrado
 - Radicación (**) $(1/r)$ $x^{**}(1/2)$ raíz cuadrada de x

➤ Comentarios en Python

Una línea # comentario

Multilíneas – un objeto string sin nombre por lo que no se asigna a memoria

""" comentario

multi
línea """

➤ Tipos de datos que se deben declarar en las variables a utilizar .

```
nota =9;
print(f'la nota es {nota} ')
puts("¡Felicitaciones!")
```

```
#Salida esperada por consola
la nota es 9
¡Felicitaciones!
```

Podremos generar varios ejemplos para cálculos de volumen, áreas, y diversas fórmulas que se requieren en el uso diario.
Calcule el IVA de un producto de ferretería 21%, elementos de electrónica o computación 10.5%, y sin IVA (0%) como algunos elementos de la canasta básica.

```
nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 7
nota_4to_p = 6
suma = nota_1er_p + nota_2do_p + nota_3er_p + nota_4to_p
```



```
media= suma/4
print (f"la media de las notas es {media} puntos / 10")
```

#Salida esperada por consola
la media de las notas es nota es 7.5 puntos / 10

Se recomienda agregar al uso de
print (f"descripción: {objeto=}")
descripción permite una comprensión del contenido del objeto
print (f"tipo de objeto: {type(objeto)=}")
type muestra el tipo asignado por el intérprete según dato

```
nombre="Juan Pérez"
nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 7
nota_4to_p = 6
suma = nota_1er_p + nota_2do_p + nota_3er_p + nota_4to_p
media= suma/4
print (f"nombre la media de las notas es {media} puntos / 10")
print (f"suma= tipo de objeto:{ type(suma)=}" )
print (f"media= tipo de objeto:{ type(media)=}" )
```

#Salida esperada por consola
Juan Pérez la media de las notas es 7.5 puntos / 10
suma=30 tipo de objeto: type(suma) =<class 'int'>
media=7.5 tipo de objeto: type(media)=<class 'float'>

Para poder mostrar que es un método en un objeto y que los distintos objetos tienen en general distintos métodos se agrega dir
Se recomienda agregar al uso de
print (f"descripción: {objeto=}")
descripción permite una comprensión del contenido del objeto
print (f"tipo de objeto: {type(objeto)=}")
type muestra el tipo asignado por el intérprete según dato
print (f"métodos del tipo de objeto: {dir(objeto)=}")
muestra los métodos del tipo de objeto asignado por el intérprete según dato

Atributos: Los atributos son las variables que pertenecen a un objeto. Pueden ser variables de instancia (definidas dentro de un método) o variables de Módulo (definidas en el Módulo pero fuera de los métodos). Cada instancia de una Módulo puede tener diferentes valores para los atributos.

Métodos: Los métodos son las funciones asociadas a un módulo. Pueden realizar operaciones en los datos del objeto y pueden acceder y modificar los atributos. Los métodos pueden ser llamados en las instancias de la Módulo y actúan en contexto a los datos específicos de la instancia.

Métodos:

En los siguientes ejemplos empezaremos a usar algunos métodos de los objetos.

```
nombre="Juan Pérez"
nota_1er_p = 9
nota_2do_p = 8
nota_3er_p = 7
```



```

nota_4to_p = 6
suma = nota_1er_p + nota_2do_p + nota_3er_p + nota_4to_p
media= suma/4
print (f"nombre la media de las notas es {media puntos / 10}")
print (f"{nombre} tipo de objeto: {type(nombre)= }")
print (f"{suma} tipo de objeto: {type(suma)= }")
print (f"{media} tipo de objeto: {type(media)= }")
print (f"{type(nombre)= }\n métodos de objeto: {dir(nombre)= } ")
print (f"{type(suma)= }\n métodos de objeto: {dir(suma)= } ")
print (f"{type(media)= }\n métodos de objeto: {dir(media)= } ")

```

#Salida esperada por consola

Juan Pérez la media de las notas es 7.5 puntos / 10

nombre='Juan Pérez' tipo de objeto: type(nombre)=<class 'str'>

suma=30 tipo de objeto: type(suma)=<class 'int'>

media=7.5 tipo de objeto: type(media)=<class 'float'>

type(nombre)=<class 'str'>

metodos de objeto: dir(nombre)=['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

type(suma)=<class 'int'>

metodos de objeto: dir(suma)=['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']

type(media)=<class 'float'>

metodos de objeto: dir(media)=['__abs__', '__add__', '__bool__', '__ceil__', '__class__', '__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getformat__', '__getnewargs__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__int__', '__le__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__', '__pow__', '__radd__', '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '__round__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', 'as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']

<class 'str'> =

```

['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

```

<class 'int'> =



```
[as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

```
<class 'float'>=
['as_integer_ratio', 'conjugate', 'fromhex', 'hex', 'imag', 'is_integer', 'real']
```

```
<class 'bool'>=
['as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

```
<class 'str'> =
['capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

métodos de string:

estilo:['capitalize', 'casefold', 'lower', 'swapcase', 'title', 'upper']

Tanto casefold() como lower() son métodos utilizados en Python para convertir cadenas de caracteres a minúsculas, pero hay una diferencia sutil entre ellos:

casefold():

Este método es más agresivo en cuanto a la conversión de caracteres a minúsculas. Se considera más adecuado para comparaciones de cadenas que deben ser insensibles a las diferencias de mayúsculas y minúsculas en cualquier idioma.

La conversión realizada por casefold() es más amplia que la de lower(). Por ejemplo, en turco, la letra "İ" (mayúscula con punto arriba) se convierte en "i" con casefold(), mientras que lower() no hace esta conversión.

lower():

Este método convierte los caracteres de la cadena a minúsculas, pero la conversión puede no ser adecuada para todos los idiomas. Por ejemplo, en turco, la letra "I" (mayúscula sin punto arriba) se convierte en "i" con lower(), pero esto no es adecuado para todas las situaciones en turco, ya que la "ı" minúscula se utiliza solo al final de las palabras.

lower() se basa principalmente en reglas de conversión de mayúsculas y minúsculas del alfabeto inglés.

En resumen, casefold() proporciona una conversión más amplia y segura de cadenas a minúsculas, lo que lo hace más adecuado para aplicaciones que requieren una comparación insensible a las diferencias de mayúsculas y minúsculas en diferentes idiomas. Por otro lado, lower() es más simple y puede ser suficiente en situaciones donde solo se necesita una conversión básica de mayúsculas a minúsculas y se sabe que el texto se limita a un solo idioma, como el inglés.

```
print(f"● Métodos de presentacion.")
print("*****")
texto = "Hola curso. Bienvenidos al mundo IT"
metodos_y_atributos=['capitalize',
                     'casefold',
                     'lower',
                     'title',
                     'upper',
                     'swapcase']
print(f"Original = {texto}")
```



```
print(f"capitalize = {texto.capitalize()}")
print(f"casefold = {texto.casefold()}")
print(f"lower = {texto.lower()}")
print(f"title = {texto.title()}")
print(f"upper = {texto.upper()}")
print(f"swapcase = {texto.swapcase()}")
```

- Métodos de presentación.

```
*****
Original = Hola curso. Bienvenidos al mundo IT
capitalize = Hola curso. bienvenidos al mundo it
casefold = hola curso. bienvenidos al mundo it
lower = hola curso. bienvenidos al mundo it
title = Hola Curso. Bienvenidos Al Mundo It
upper = HOLA CURSO. BIENVENIDOS AL MUNDO IT
swapcase = hOLA CURSO. bIENVENIDOS AL MUNDO it
*****
```

ubicación:['center', 'ljust', 'rjust',]

```
print(f"• Métodos de ubicación y tabulación.")
print("****50)
texto = "Hola curso. Bienvenidos al mundo IT"
metodos_y_atributos=[
    'center',
    'ljust',
    'rjust'
]
print(f"texto.center(50) = |{texto.center(50)}|")
print(f"texto.ljust(50) = |{texto.ljust(50)}|")
print(f"texto.rjust(50) = |{texto.rjust(50)}|")
print("****50)
```

#Salida esperada por consola

- Métodos de ubicación y tabulación.

```
*****
texto.center(50) = |    Hola curso. Bienvenidos al mundo IT    |
texto.ljust(50) = |Hola curso. Bienvenidos al mundo IT        |
texto.rjust(50) = |        Hola curso. Bienvenidos al mundo IT|
*****
```

booleanos:['isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'startswith', 'endswith']

#métodos de string: booleano

isalnum():	Devuelve True si todos los caracteres de la cadena son alfanuméricos (letras o números), de lo contrario, devuelve False.
isalpha():	Devuelve True si todos los caracteres de la cadena son letras (no incluye números ni caracteres especiales), de lo contrario, devuelve False.
isascii():	Devuelve True si todos los caracteres de la cadena son ASCII (American Standard Code for Information Interchange), de lo contrario, devuelve False.
isdecimal():	Devuelve True si todos los caracteres de la cadena son dígitos decimales (0-9), de lo contrario,



	devuelve False.
isdigit():	Devuelve True si todos los caracteres de la cadena son dígitos (incluyendo subíndices, superíndices y otros dígitos unicode), de lo contrario, devuelve False.
isidentifier():	Devuelve True si la cadena es un identificador válido de Python (por ejemplo, si puede ser un nombre de variable), de lo contrario, devuelve False.
islower():	Devuelve True si todos los caracteres de la cadena están en minúsculas y al menos un carácter está en mayúsculas, de lo contrario, devuelve False.
isnumeric():	Devuelve True si todos los caracteres de la cadena son numéricos, de lo contrario, devuelve False.
isprintable():	Devuelve True si todos los caracteres de la cadena son imprimibles o la cadena está vacía, de lo contrario, devuelve False. (Ej: \n\t'a)
isspace():	Devuelve True si todos los caracteres de la cadena son espacios en blanco (por ejemplo, espacio, tabulación, salto de línea), de lo contrario, devuelve False.
istitle():	Devuelve True si la cadena está en formato de título, es decir, si todas las palabras comienzan con mayúscula y las demás letras son minúsculas, de lo contrario, devuelve False.
isupper():	Devuelve True si todos los caracteres de la cadena están en mayúsculas y al menos un carácter está en mayúsculas, de lo contrario, devuelve False.
startswith() (prefix[, start[, end]]):	Comprueba si una cadena comienza con un prefijo específico. prefix es el prefijo que se está buscando. start (opcional) es el índice de inicio para la búsqueda. end (opcional) es el índice de fin para la búsqueda.
endswith() (suffix[, start[, end]]):	Comprueba si una cadena termina con un sufijo específico. suffix es el sufijo que se está buscando. start (opcional) es el índice de inicio para la búsqueda. end (opcional) es el índice de fin para la búsqueda.

```
# métodos de string: booleano
print(f"\n● Métodos condicional.\n")
print("*****50)
metodos_y_atributos=[ 'endswith',           'isalnum',          'isalpha',          'isascii',
                      'isdecimal',         'isdigit',          'isidentifier',    'islower',
                      'isnumeric',        'isprintable',      'isspace',         'istitle',
                      'isupper',          'startswith' ]
texto = "Hola curso. Bienvenidos al mundo IT"
print(f"texto.endswith('IT')    = {texto.endswith('IT')} ")
print(f"texto.endswith('Pepe')  = {texto.endswith('Pepe')} ")
print(f"texto.startswith('Hola') = {texto.startswith('Hola')} ")
print(f"texto.startswith('chau') = {texto.startswith('chau')} ")
print(f"'123'.isalnum()       = {'123'.isalnum()} ")
print(f"'1.3'.isalnum()       = {'1.3'.isalnum()} ")
print(f"'—'.isalnum()         = {'—'.isalnum()} ")
print(f"'123'.isdecimal()     = {'123'.isdecimal()} ")
print(f"'1.3'.isdecimal()     = {'1.3'.isdecimal()} ")
print(f"'—'.isdecimal()       = {'—'.isdecimal()} ")
print(f"'123'.isdigit()      = {'123'.isdigit()} ")
print(f"'1.3'.isdigit()      = {'1.3'.isdigit()} ")
print(f"'—'.isdigit()         = {'—'.isdigit()} ")
print(f"'hola'.islower()       = {'hola'.islower()} ")
```



```

print(f"Hola".islower()      = {'Hola'.islower()}"')
print(f"HOLA".isupper()     = {'HOLA'.isupper()}"')
print(f"Hola".isupper()     = {'Hola'.isupper()}"')
print(f"holá".isalpha()      = {'holá'.isalpha()}"')
print(f"Hola 2023".isalpha() = {'Hola 2023'.isalpha()}"')
print(f"holà 2023".isascii() = {'holà 2023'.isascii()}"')
print(f"—".isascii()        = {'—'.isascii()}"')
print(f"Hola".isidentifier() = {'Hola'.isidentifier()}"')
print(f"holà —".isidentifier() = {'holà —'.isidentifier()}"')
print(f"Hola 2023".isprintable()= {'Hola 2023'.isprintable()}"')
print(f"holà —".isprintable() = {'holà —'.isprintable()}"')
print(f" ".isspace()         = {' '.isspace()}"')
print(f" x ".isspace()       = {' x '.isspace()}"')
print("*****50)

```

- Métodos condicional.

```

*****
texto.endswith('TT')    = True
texto.endswith('Pepe')   = False
texto.startswith('Hola') = True
texto.startswith('chau') = False
'123'.isalnum()          = True
'1.3'.isalnum()          = False
'—'.isalnum()            = True
'123'.isdecimal()        = True
'1.3'.isdecimal()        = False
"—".isdecimal()          = False
'123'.isdigit()          = True
'1.3'.isdigit()          = False
'—'.isdigit()            = False
'holá'.islower()          = True
'Hola'.islower()          = False
'HOLA'.isupper()          = True
'Hola'.isupper()          = False
'holá'.isalpha()          = True
'Hola 2023'.isalpha()     = False
'holà 2023'.isascii()     = True
'—'.isascii()             = False
'Hola'.isidentifier()     = True
'holà —'.isidentifier()   = False
'Hola 2023'.isprintable() = True
'holà —'.isprintable()    = True
' '.isspace()              = True
' x '.isspace()            = False
*****

```

```

# Casting de entero a string
numero_str = "1025468"
if numero_str.isdigit():#reemplazar por isdecimal() o isnumeric()
    print(f"ingreso: {numero_str} {type(numero_str)}")
    numero_int = int(numero_str)
    print(f"casting: {numero_int} {type(numero_int)}")

```



```
else:  
    print(f"El valor no puede pasar a entero: {numero_str}")
```

#Salida esperada por consola
ingreso: 1025468 <class 'str'>
casting: 1025468 <class 'int'>

```
# Casting de entero a string  
numero_str = "102no468"  
if numero_str.isdigit():#reemplazar por isdecimal() o isnumeric()  
    print(f"ingreso: {numero_str} {type(numero_str)}")  
    numero_int = int(numero_str)  
    print(f"casting: {numero_int} {type(numero_int)})")  
else:  
    print(f"El valor no puede pasar a entero: {numero_str}")
```

#Salida esperada por consola
El valor no puede pasar a entero: 102no68

```
# Casting de entero a string  
numero_str = "102.468"  
#           ^ punto  
if numero_str.isdigit():#reemplazar por isdecimal() o isnumeric()  
    print(f"ingreso: {numero_str} {type(numero_str)}")  
    numero_int = int(numero_str)  
    print(f"casting: {numero_int} {type(numero_int)})")  
else:  
    print(f"El valor no puede pasar a entero: {numero_str}")
```

#Salida esperada por consola
El valor no puede pasar a real / float: 102.468

```
# Casting de entero a string  
numero_str = "102.468"  
#           ^ punto  
if numero_str.replace(".",",").isdigit():#reemplazar por isdecimal() o isnumeric()  
    print(f"ingreso: {numero_str} {type(numero_str)}")  
    numero_int = int(numero_str)  
    print(f"casting: {numero_int} {type(numero_int)})")  
else:  
    print(f"El valor no puede pasar a entero: {numero_str}")
```

#Salida esperada por consola
ingreso: 102.468 <class 'str'>
casting: 102.468 <class 'float'>



➤ Entrada de datos por Teclado (función input)

En el lenguaje Python, puedes utilizar la función `input` para leer datos ingresados por el usuario desde la consola. Esta función solo permite ingresar un solo tipo de dato, `string`

Un `input` siempre devuelve un `string`

un `string` es una cadena de caracteres, no de abecedario, sino de todo símbolo, letra, número, etc. de los que está dentro de las tablas unicode / utf8 según corresponde

Esto significa que se puede entrar en un `input` datos en chino, cirílico, etc.

Aquí tienes un ejemplo básico de cómo utilizar `input` para ingresar datos por consola:

```
string1 = input(f"Ingrese su nombre: ")
string2 = input(f"Ingrese su apellido: ")
gracia = string1+" "+string2#      agrego un espacio entre nombre y apellido
print(f"su nombre y apellido es: {gracia}")
```

#Salida esperada por consola

Ingresá tu nombre: Juan#<-----ingreso de la altura del usuario

Ingresá tu apellido: Perez#<-----ingreso de la altura del usuario

tu nombre y apellido es: Juan Perez

➤ Casting:

El casting de datos, también conocido como conversión de tipos, se refiere a la acción de cambiar el tipo de objeto y valor de un dato a otro tipo y valor siempre que se cumplan ciertas reglas. En Python, puedes realizar conversiones de tipos utilizando funciones incorporadas que permiten cambiar entre tipos numéricos, cadenas y otros tipos de datos:

<code>str(x)</code>	Convierte (si puede) el obj <code>x</code> al tipo de dato <code>string</code>
<code>int(x)</code>	Convierte (si puede) el obj <code>x</code> al tipo de dato entero
<code>float(x)</code>	Convierte (si puede) el obj <code>x</code> al tipo de dato flotante, real o decimal
<code>bool(x)</code>	Convierte (si puede) el obj <code>x</code> al tipo de dato booleano

Casting de string a entero

numero_str = "10"

```
numero_int = int(numero_str)#
print(f'Resultado: {numero_int}')
```

casting directo de string a int

#Salida esperada por consola

Resultado: 10

Casting de punto flotante a entero

numero_float = 3.14

```
numero_int = int(numero_float)#
print(f'Resultado: {numero_int}')
```

casting directo de float a int

#Salida esperada por consola

Resultado: 3



```
# Casting de booleano a entero
booleano = True
numero_int = int(booleano)#
print(f"Resultado: {numero_int}")
```

casting directo de bool a int

#Salida esperada por consola
Resultado: 1

```
# Casting de entero a punto flotante
numero_int = 10
numero_float = float(numero_int)#
print(f"Resultado: {numero_float}")
```

casting directo de int a float

#Salida esperada por consola
Resultado: 10.0

```
# Casting de string a punto flotante
numero_str = "3.14"
numero_float = float(numero_str)#
print(f"Resultado: {numero_float}")
```

casting directo de string a float

#Salida esperada por consola
Resultado: 3.14

```
# Casting de booleano a punto flotante
booleano = False
numero_float = float(booleano)#
print(f"Resultado: {numero_float}")
```

casting directo de bool a float

#Salida esperada por consola
Resultado: 0.0

```
# Casting de entero a string
numero_int = 10
numero_str = str(numero_int)#
print(f"Resultado: {numero_str}")
```

casting directo de int a string

#Salida esperada por consola
Resultado: "10"

```
# Casting de booleano a string
booleano = True
numero_str = str(booleano)#
print(f"Resultado: {numero_str}")
```

casting directo de string a bool

#Salida esperada por consola
Resultado: "True"

Recuerda que el casting solo es posible si la conversión tiene sentido semántico. Por ejemplo, no es posible convertir una cadena de caracteres que no representa un número válido a entero o punto flotante.

```
# Casting de booleano a string
```



```
numero_str = "3.14"
numero_int = int(numero_str)#
print(f"Resultado: {numero_int}")
```

casting directo de string con . decimal a int

#Salida esperada por consola
 Traceback (most recent call last):
 File "ejercicios.py", line xx, in <module>
 numero_int = int(numero_str)
 ****=
 ValueError: invalid literal for int() with base 10: '3.14'

```
variable = input("ingrese un dato:")
"""

```

Recibe el dato y lo guarde en la memoria ram en un espacio que lleva la etiqueta “variable”.

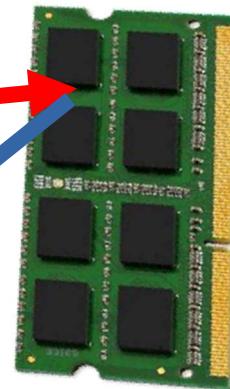
#Salida esperada por consola
 ingrese un dato:**'HOLA'**

```
print(f"El contenido ingresado por el usuario es {variable}")
"""

```

imprime desde la memoria ram el objeto guardado con la etiqueta “variable”.

#Salida esperada por consola
 El contenido ingresado por el usuario es variable=**'HOLA'**



Ingresá la primer nota: 9
 Ingresá la segunda nota: 7
Houston. We have a problem!



```
nota_1 = input("Ingresa la primer nota: ")
nota_2 = input("Ingresa la segunda nota: ")
promedio = (nota_1 + nota_2) / 2
print(f"El promedio de tus notas es {promedio} ")
```

¿Cuál es el error?

Aquí usamos 3 variables. 2 datos ingresan por el usuario desde el teclado. La 3er variable guarda el promedio de las ingresadas por teclado. Luego lo imprimo

#Salida esperada por consola



Ingresá la primera nota: **9**

nota_2 = input("Ingresá la segunda nota: **7**")

El promedio de tus notas es promedio=**¿?;?**

Todo ingreso desde input es un string

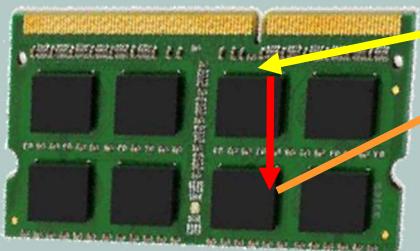
Por lo que si debo ingresar un dato numérico

Debo hacer un casting, un cambio de tipo de variable

dato = int (dato) pasa dato a entero

dato = float (dato) pasa dato a flotante

dato = str (dato) pasa dato a string



Cada tipo de dato tiene su propio lugar en la RAM

nota_1 = input("Ingresá la primera nota: ")

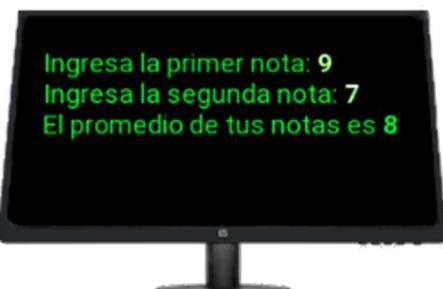
nota_1 = int(nota_1)

nota_2 = input("Ingresá la segunda nota: ")

nota_2 = int(nota_2)

promedio = (nota_1 + nota_2) / 2

print(f"El promedio de tus notas es {promedio} ")



Aquí usamos 3 variables. 2 datos ingresan por el usuario desde el teclado, convierto a entero nota_1 y nota_2. La 3er variable guarda el promedio que como es una división se convierte en float, luego lo imprimo.

#Salida esperada por consola

Ingresá la primera nota: **9**

Ingresá la segunda nota: **7**

El promedio de tus notas es promedio = **8**

Metodo	Diferencia
str.isnumeric()	True si todos los caracteres son arábigos (0-9) o numéricos (lo que abarca una gran cantidad de strings) print("2³".isnumeric()) # True (subíndices o superíndices numéricos)



	<pre>print("½".isnumeric()) # True (fracción) print("IV".isnumeric()) # True (número romano) print("୫୯୩".isnumeric()) # True (números Devanagari) print("三".isnumeric()) # True (número chino) print("۱۲۳".isnumeric()) # True (números árabes-índicos) print("கூ-ஞ்".isnumeric()) # True (números tamil)</pre> <p>Cada cadena contiene caracteres que representan números en sistemas de escritura no latinos, como fracciones, números romanos, números Devanagari (usados en hindi y otros idiomas), números chinos, números árabes-índicos, y números tamil.</p>
str.isdigit()	True si todos los caracteres son arábigos (0-9) estándar, subíndices o superíndices numéricos. <code>print("2³".isnumeric()) # True (subíndices o superíndices numéricos)</code>
str.isdecimal ()	True si todos los caracteres son solo arábigos (0-9)

.format()	
# n = 3 => 3 + 33 + 333 = 369 n = input('Escriba el valor de n: ') nn = int('{} {}'.format(n, n)) nnn = int('%s%s%s' % (n, n, n)) n = int(n) suma = n + nn + nnn printf(f la suma es {suma})# 2 + 22 + 222 = 246	
fecha_evento = (2024, 9, 22) print(type(fecha_evento)) print(fecha_evento) print('El evento programado será para la fecha:', fecha_evento) print('El evento programado será para la fecha: %i/%i/%i' % fecha_evento) año, mes, dia = fecha_evento print('El evento programado será para la fecha: {} / {} / {} '.format(año, mes, dia))	



Módulo 4: Modificador de flujo

Condiciones y Bucles while

En Python, la función while se utiliza para crear bucles o ciclos que se ejecutan mientras una determinada condición sea verdadera. El bloque de código dentro del while se repetirá continuamente hasta que la condición se evalúe como falsa.

La sintaxis básica de while es la siguiente:

```
while (condición):
```

```
    # Código que se ejecuta MIENTRAS la condición es verdadera - True
```

```
    Código que se ejecuta cuando la condición pasa a ser falsa - False
```

```
contador = 0
while contador < 5:
    print(f"\tEl contador es: {contador}")
    contador = contador + 1
print("Adiós...")
```

#Salida esperada por consola

```
    El contador es: 0
    El contador es: 1
    El contador es: 2
    El contador es: 3
    El contador es: 4
```

```
Adiós...
```

En el ejemplo anterior, creamos una variable contador inicializada en 0. Luego, utilizamos la función while para repetir un bloque de código mientras la condición contador < 5 sea verdadera.

Dentro del bucle while, imprimimos el valor actual del contador y luego incrementamos su valor en 1 utilizando contador = contador + 1.

Esto asegura que el bucle eventualmente terminará cuando la condición se evalúe como falsa.

En el siguiente trabajamos a la inversa

```
contador = 5
while contador !=0:
    print(f"\tEl contador es: {contador}")
    contador = contador - 1
print("Adiós...")
```

#Salida esperada por consola

```
    El contador es: 5
    El contador es: 4
    El contador es: 3
    El contador es: 2
    El contador es: 1
```

```
Adiós...
```



break

```
contador = 15
while contador !=0:
    print(f"\tEl contador es: {contador}")
    contador = contador - 1
    if contador == 5:
        print("break")
        break
print("Adiós...")
```

#Salida esperada por consola

```
El contador es: 15
El contador es: 14
El contador es: 13
El contador es: 12
El contador es: 11
El contador es: 10
El contador es: 9
El contador es: 8
El contador es: 7
El contador es: 6
```

```
break
```

```
Adiós...
```

continue

```
contador = 8
while contador !=0:
    print(f"El contador es: {contador}", end="")
    contador = contador - 1
    if contador%2 == 0:
        print("\t\tcontinue x par")
        continue
    print("\timpar")
print("Adiós...")
```

#Salida esperada por consola

```
El contador es: 8      impar
El contador es: 7      continue x par
El contador es: 6      impar
El contador es: 5      continue x par
El contador es: 4      impar
El contador es: 3      continue x par
El contador es: 2      impar
El contador es: 1      continue x par
```



Anidación

```
externo=0
while externo <10:
    interno=0
    print("\t",end="")
    while interno <10:
        print(f"\t{externo} - {interno}", end=" | ")
        interno+=1
    print()
    externo+=1
```

Salida esperada por consola

```
0 - 0 | 0 - 1 | 0 - 2 | 0 - 3 | 0 - 4 | 0 - 5 | 0 - 6 | 0 - 7 | 0 - 8 | 0 - 9 |
1 - 0 | 1 - 1 | 1 - 2 | 1 - 3 | 1 - 4 | 1 - 5 | 1 - 6 | 1 - 7 | 1 - 8 | 1 - 9 |
2 - 0 | 2 - 1 | 2 - 2 | 2 - 3 | 2 - 4 | 2 - 5 | 2 - 6 | 2 - 7 | 2 - 8 | 2 - 9 |
3 - 0 | 3 - 1 | 3 - 2 | 3 - 3 | 3 - 4 | 3 - 5 | 3 - 6 | 3 - 7 | 3 - 8 | 3 - 9 |
4 - 0 | 4 - 1 | 4 - 2 | 4 - 3 | 4 - 4 | 4 - 5 | 4 - 6 | 4 - 7 | 4 - 8 | 4 - 9 |
5 - 0 | 5 - 1 | 5 - 2 | 5 - 3 | 5 - 4 | 5 - 5 | 5 - 6 | 5 - 7 | 5 - 8 | 5 - 9 |
6 - 0 | 6 - 1 | 6 - 2 | 6 - 3 | 6 - 4 | 6 - 5 | 6 - 6 | 6 - 7 | 6 - 8 | 6 - 9 |
7 - 0 | 7 - 1 | 7 - 2 | 7 - 3 | 7 - 4 | 7 - 5 | 7 - 6 | 7 - 7 | 7 - 8 | 7 - 9 |
8 - 0 | 8 - 1 | 8 - 2 | 8 - 3 | 8 - 4 | 8 - 5 | 8 - 6 | 8 - 7 | 8 - 8 | 8 - 9 |
9 - 0 | 9 - 1 | 9 - 2 | 9 - 3 | 9 - 4 | 9 - 5 | 9 - 6 | 9 - 7 | 9 - 8 | 9 - 9 |
```

Operadores de asignación:

Estos operadores se utilizan para asignar valores a variables.

=	Asigna el valor de la expresión a la variable.
+=	Suma el valor de la expresión a la variable y almacena el resultado.
-=	Resta el valor de la expresión a la variable y almacena el resultado.
/=	Divide el valor de la expresión a la variable y almacena el resultado.
//=	Divide el valor entero de la expresión a la variable y almacena el resultado.
%=	Calcula el resto de la división entre la variable y la expresión y lo almacena.
*=	Multiplica el valor de la expresión a la variable y almacena el resultado.
**=	Exponencia el valor de la expresión a la variable y almacena el resultado.

<u>Operadores de asignación de Python</u>			
=	x = 7	x = 7	7
+=	x += 8	x = x + 8	15
-+	x -= 10	x = x - 10	5
*=	x *= 8	x = x * 8	40
/=	x /= 8	x = x / 8	2
**=	x **= 10	x = x ** 10	1024
//=	x // 5	x = x // 5	204



%=	x % = 5	x = x % 5	4
asignación a nivel de bits se verán por separado			

Validación de datos: while

```
while entrada != 'S':
    entrada = input('¿desea salir ? (S/N) :').upper()
    print('seguimos en el while')
print ('Salimos, la variable entrada es igual a S')
```

#Salida esperada por consola

¿ desea salir ? (S/N) : **n**

seguimos en el while

¿ desea salir ? (S/N) : **S**

seguimos en el while

Salimos, la variable entrada es igual a S

Operadores de identidad en condicionales:

is	True si ambos operandos hacen referencia al mismo objeto. False en caso contrario.
is not	True si ambos operandos NO hacen referencia al mismo objeto. False en caso contrario.

Registraremos un objeto string para poder usar sus métodos isnumber() o isdigit() o isnumeric()
Generamos un while MIENTRAS el objeto string ingresado NO este compuesto solo por números.
Ingresamos un objeto string mediante un input dentro de un while

El bucle while se mantiene mientras la condición sea True por lo que si la niego o pido que sea falsa
False obtendré una salida en el momento que ingrese un numero

Un string es una cadena de caracteres.

Cada eslabón es un carácter.

isnumber() o isdigit() o isnumeric() evalúa que cada carácter de la cadena sea un numero (0 a 9)

Creo un objeto tipo str para poder usar sus métodos

Todo input devuelve un string

Para validar usamos algún método booleano de string

Si es necesario validar números .isdecimal() o isnumeric() o isdigit() por ahora son similares
casting de str a int

entrada=""

while entrada.isdecimal() is False:



```

entrada = input ('un numero entero :')
print ('Salimos')
print (f'El tipo de dato es {type(entrada)}')
entrada = int(entrada) # casting de str a int
print (f'la variable entrada es igual a {entrada}')
print (f'El tipo de dato es {type(entrada)}')

```

Salida esperada por consola

```

un numero entero : puntos 3
un numero entero : 3 puntos
un numero entero : 8

```

Salimos
El tipo de dato es <class 'str'>
la variable entrada es igual a 8
El tipo de dato es <class 'int'>

Si es un numero con punto decimal
usamos replace dentro del while donde si encontramos puntos ('.') lo cambiamos por un string vacío ('')
Un string es una cadena de caracteres.
Cada eslabón es un carácter.
En un numero decimal tengo un carácter punto (.) que separa la parte entera de las decimales
pi= 3.14159
**no es numero
isnumber() o isdigit() o isnumeric() evalúan que cada carácter de la cadena sea un número (0 a 9)
pi= 3.14159
pi.replace(".", "")
se lee como 314159 sin punto, Esto permite usar isnumber() o isdigit() o isnumeric()

```

entrada=""
while entrada.replace(".","",).isdecimal() is False:
    entrada = input ('un numero flotante :')
print ('Salimos')
print (f'El tipo de dato es {type(entrada)}')
entrada = float(entrada) # casting de str a float
print (f'la variable entrada es igual a {entrada}')
print (f'El tipo de dato es {type(entrada)}')

```

Salida esperada por consola

```

un numero flotante : 3 puntos
un numero flotante : 3 puntos 14159
un numero flotante : 3.14159

```

Salimos

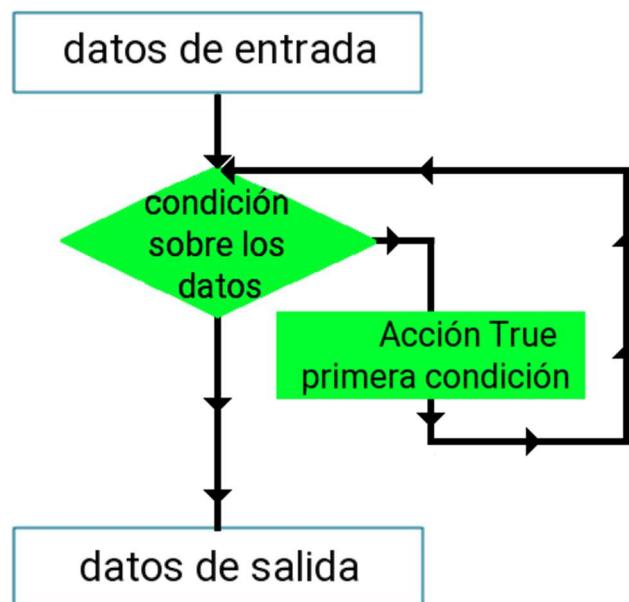


El tipo de dato es <class 'str'>
la variable entrada es igual a 3.14159
El tipo de dato es <class 'float'>

Se puede usar la misma función evaluada e formas distintas.

```
while entrada.replace(".", "").isdecimal() is False:  
while entrada.replace(".", "").isdecimal() is not True:  
while not entrada.replace(".", "").isdecimal():
```

comprobar que el bucle mientras no se cumple deja encerrado al usuario.



La función **while** lleva una condición como un **if** pero el bloque de información que esta en un bloque, indentada, se repite, itera, hasta que la condición sea **True**. Cuando la condición se vuelve **False** sale del bucle



Módulo 5: colecciones

- Notas
- tuplas
- listas
- set (frozenset)
- Pilas y colas
- anidación - pseudomatrices
- diccionarios

Nota aclaratoria:

Según la costumbre latina y, en especial, la ley argentina denominada SiMeLA (Sistema Métrico Legal Argentino), la parte entera se separa de la decimal por una coma (",").

En programación, esto no es válido; el separador es el punto (".").

objeto = 3.14159

¿Qué tipo de dato es?

Nosotros diríamos que es un número real o flotante. Pero no en programación, donde la coma separa los objetos. Aquí tendríamos dos objetos: el 3 (entero) y el 14159 (también entero).

Código Python

```
objeto = 3,14159
print(f"objeto={} es del tipo {type(objeto)}")
```

#Salida esperada por consola

```
objeto=(3, 14159) es del tipo <class 'tuple'>
```

Es una tupla, una colección de dos (2) datos enteros.

objeto = 3,14159

La coma separa objetos en una colección

En una tupla los paréntesis son recomendados por la comunidad pero no obligatorios por el intérprete Python

Una mejor manera de representar una tupla es la siguiente

objeto = (3,14159)

 \ \ /

 | |

 2do dato 14159 tipo entero

 1er dato 3 tipo entero

Colecciones:

Una colección en Python es un tipo de objeto que puede contener múltiples elementos u objetos. Estas colecciones son utilizadas para almacenar, organizar y manipular conjuntos de datos de manera eficiente.

Hay diferentes tipos de colecciones, unas integradas en su biblioteca estándar (que son ampliamente utilizadas en programación) y otras a través de librerías, bibliotecas o módulos.



Iterables:

Un iterable en programación se refiere a un objeto capaz de proporcionar una secuencia de elementos de uno en uno. Estos elementos pueden ser accedidos en un orden específico, y el objeto iterable facilita la iteración sobre sus elementos. En muchos lenguajes de programación, la iteración se logra mediante el uso de bucles, como el bucle for.

En el contexto de Python, un iterable es cualquier objeto sobre el cual se puede iterar. Para que un objeto sea iterable, debe implementar el método especial `__iter__()` que devuelve un objeto iterador. Un iterador, a su vez, debe implementar el método `__next__()` que proporciona el siguiente elemento de la secuencia.

Toda colección es un iterable pero no todo iterable es una colección:

Toda colección es un iterable:

Una colección, como una lista, tupla, conjunto o diccionario en Python, es un tipo de estructura de datos que agrupa elementos bajo un mismo nombre o identificador.

Todas las colecciones en Python son iterables, lo que significa que se pueden recorrer elemento por elemento utilizando bucles for u otras construcciones de iteración.

No todo iterable es una colección:

Aunque todas las colecciones son iterables, existen otros tipos de objetos que también son iterables pero no se consideran colecciones en el sentido tradicional.

Ejemplos de iterables, que no son colecciones incluyen rangos, cadenas de texto, generadores, y archivos.

Un rango (range), que es un iterable, no es una colección porque no almacena todos los elementos en la memoria de una vez, sino que los genera a medida que son necesarios.

Un string (str) - una cadena de texto es iterable, no se considera una colección pero es inmutable, lo que significa que no se pueden modificar sus elementos.

tuple (Tuplas):

Son colecciones ordenadas, con índice e inmutables de objetos separados por comas.

- Ordenadas
- Con índice
- Inmutables
- Separados por comas.
- No se pueden modificar después de la creación.

Estos objetos que la componen pueden ser de cualquier tipo (str, int, float, bool y.....colecciones listas,tuplas, diccionarios, sets, frozensets,etc.).

La sintaxis básica es la siguiente:

una tupla con objetos int

```
nombre_tupla = (1,2,3,4,5,6,7,8,9,0)
```

una tupla con objetos float

```
nombre_tupla = (3.14159, 2.99792458, 4.5)
```

una tupla con objetos str

```
nombre_tupla = ("Hola", "Mundo", "IT")
```

una tupla con objetos bool



nombre_tupla = (True,False,True,False)

una tupla con objetos mix

nombre_tupla = (1, 2, 3.14159, "Hola", True).

list (Listas):

Son colecciones ordenadas, con índice y mutables de objetos separados por comas.

- Ordenadas
- Con índice
- Mutables
- Separados por comas.
- Se pueden modificar (agregar, eliminar, modificar elementos).

Estos objetos que la componen pueden ser de cualquier tipo (str, int, float, bool y.....colecciones listas,tuplas, diccionarios, sets, frozensets,etc.)

La sintaxis básica es la siguiente:

una lista vacía

nombre_lista = list()
o
nombre_lista = []

una lista con objetos int

nombre_lista = [1,2,3,4,5,6,7,8,9,0]

Set (Conjuntos):

Son colecciones desordenadas, sin índice, únicos y mutables de objetos separados por comas.

- Desordenadas
- Sin índice No se accede a los elementos mediante índices
- Únicos No permite duplicados.
- Mutables
- separados por comas.
- Ideal para realizar operaciones de conjuntos, como unión, intersección y diferencia.

Estos objetos que la componen pueden ser de cualquier tipo (str, int, float, bool y.....colecciones listas,tuplas, diccionarios, sets, frozensets,etc.)

La sintaxis básica es la siguiente:

un set vacío

nombre_set = set()
o
nombre_set = {}#<-----ver mas adelante con dict

una set con objetos int

nombre_set = {1,2,3,4,5,6,7,8,9,0}



FrozenSet (Conjuntos):

Son colecciones desordenadas, sin index, únicos y inmutables de objetos separados por comas.

Desordenadas

Sin índice No tienen un índice o índices.

Únicos No permiten duplicados

Inmutables

separados por comas.

Estos objetos que la componen pueden ser de cualquier tipo (str, int, float, bool y.....colecciones listas,tuplas, diccionarios, sets, frozensets,etc.)

La sintaxis básica es la siguiente:

nombre_frozenset = frozenset([])

una frozenset con objetos int

nombre_frozenset = frozenset([1,2,3,4,5,6,7,8,9,0])

Dictionary (Diccionarios):

Son colecciones de pares de objetos clave-valor / key - value

- Colección no ordenada de pares clave-valor.
- Cada elemento tiene una clave única asociada.
- El primero es la clave o llave (key) – única - inmutable
- El segundo es el valor (value) – no única - mutables e inmutable
- Se pueden realizar operaciones como agregar, eliminar y modificar elementos.

además son desordenadas, sin index, únicos y inmutables de objetos separados por comas.

Desordenadas

Sin índice No tienen un índice o índice.

keys

Únicos No permiten duplicados

Inmutables

Estos objetos que la componen pueden ser

(enteros (int), flotantes (float), tuplas (tuple) y frozenset.)

Separadores clave1:valor, clave2:valor, clave3:valor

: dos puntos las claves se separan de los valores con dos puntos(:)

Values

No Únicos Si permiten duplicados

Mutables

Estos objetos que la componen pueden ser de cualquier tipo

(str, int, float, bool y.....colecciones listas,tuplas, diccionarios, sets, frozensets,etc.)

Los pares clave-valor separados por comas.

#un dict vacío

```
nombre_dic = dict()
# o
nombre_dict = {}#<-----ver más adelante con set
```

#una dict con claves-keys con objetos permitidos

```
my_dict = {
    "nombre": "key str",
    42 : "key int",
    (1, 2, 3) : "key tuple",
    3.14 : "key float",
    frozenset([1,2, 3]) : "key frozenset"
}
```



#una dict con valores con objetos simples y colecciones

```
my_dict = {
    "value str" : "Juan",
    "value int" : 25,
    "value float" : 3.14159,
    "value complex" : 3j
    "value tuple" : (1, 2, 3)
    "value list" : [1, 2, 3]
    "value set" : {1, 2, 3}
    "value frozenset" : frozenset([1, 2, 3])
    "value dict" :
        {
            "1er sub key" : 1,
            "2do sub key": 2,
            "3er sub key" : 3,
            "4to sub key" : 4
        }
}
```

Range (Rangos):

- No es una colección en sí misma, pero se utiliza para generar secuencias inmutables e números.
- Representa una secuencia inmutable de números y se utiliza principalmente en bucles y en situaciones donde se necesita una secuencia de valores.
- Se crea mediante la función range(inicio, limite_final, step) donde el valor inicial es por defecto 0, limite_final es valor no incluido en la secuencia, y step es el paso entre los números por defecto 1.
- Los rangos son eficientes en términos de memoria, ya que generan valores a medida que se necesitan en lugar de almacenar toda la secuencia en memoria.

Se utiliza principalmente en bucles para iterar sobre un rango de valores.

La sintaxis básica es la siguiente:

una range puede tener 3 parámetros

```
nombre_range = range(inicio, limite_final, step)
nombre_range = range(5,50,2)
                    desde 5
                    hasta 50 no incluido 99 real
                    de 2 en 2
nombre_range = range(0,20,1)
nombre_range = range(0,20)
nombre_range = range(20)
```

Devuelve una secuencia inmutable de números.

```
nombre_range = range(5,50,2)
print (f'rango ={nombre_range}')
print (f'lista ={list(nombre_range)}')
```

#Salida esperada por consola

```
rango =range(5, 50, 2)
lista =[5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43, 45, 47, 49]
```



```
rango = range(0,20,1) #      range(inicio, limite_final, step)
print(f'rango (I,F,S) es {rango=}')

rango = range(0,20) #      range(inicio, limite_final,step)
#                  se omite el , step cuando es igual a 1 – None
print(f'rango (I,F) es {rango=}')

rango = range(20)#       range(inicio, limite_final, step)
#                  se omite el , inicio cuando es igual a 0 – None
print(f'rango (I) es {rango=}')
print(f'lista ={list(rango)}')

#Salida esperada por consola
rango (I,F,S) es rango=range(0, 20)
rango (I,F) es rango=range(0, 20)
rango (I) es rango=range(0, 20)
lista =[0,1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Python también proporciona módulos adicionales en su biblioteca estándar que ofrecen colecciones más especializadas y eficientes.

Ejemplos:

collections: Este Módulo ofrece tipos de colecciones adicionales como deque (cola doblemente terminada), Counter (contador), OrderedDict (diccionario ordenado) y namedtuple (tupla con nombre), entre otros.

array: Este Módulo proporciona un tipo de colección llamado array, que es una estructura de datos eficiente para almacenar elementos del mismo tipo de datos en un arreglo.

heapq: Este Módulo implementa estructuras de datos de montículos (heaps) que se utilizan para mantener el orden en una colección de manera eficiente.

queue: Este Módulo proporciona implementaciones de diferentes tipos de colas, como Queue, LifoQueue (pila) y PriorityQueue (cola de prioridad).

Y debemos tener muy en cuenta los arrays de la librería **numpy** ([Numerical Python](#)) y los DataFrames de **pandas** ([Panel Datos](#))

En física, existen varias constantes fundamentales que juegan un papel crucial en diversas teorías y ecuaciones. Aquí tienes algunos ejemplos de constantes importantes en física:

¿Qué estructura de datos utilizarías?

Velocidad de la luz en el vacío (c): Es la velocidad máxima a la que puede propagarse la luz en el vacío y es una constante fundamental en la teoría de la relatividad. Su valor aproximado es 299792458 metros por segundo.

Constante gravitacional (G): Es una constante que aparece en la ley de gravitación universal de Newton y determina la fuerza gravitatoria entre dos objetos masivos. Su valor aproximado es $6.67430 \times 10^{-11} \text{ m}^3/(\text{kg}\cdot\text{s}^2)$.

Carga elemental (e): Es la carga eléctrica fundamental de un electrón o un protón. Su valor es aproximadamente 1.602×10^{-19} culombios.

Constante de Planck (h): Es una constante que está relacionada con la cuantización de la energía y se utiliza en la mecánica cuántica. Su valor es aproximadamente $6.62607015 \times 10^{-34}$ julios-segundo.

Constante de Boltzmann (k): Es una constante que relaciona la temperatura con la energía en la física



estadística y termodinámica. Su valor es aproximadamente $1.380649 * 10^{**-23}$ julios por kelvin.

Número de Avogadro (NA): Es el número de átomos o moléculas en un mol de sustancia. Su valor aproximado es $6.022 * 10^{**23}$ moléculas por mol.

Masa del electrón (me): Es la masa de un electrón, una partícula subatómica con carga negativa. Su valor es aproximadamente $9.10938356 * 10^{**-31}$ kilogramos.

Masa del protón (mp): Es la masa de un protón, una partícula subatómica con carga positiva. Su valor es aproximadamente $1.67262192 * 10^{**-27}$ kilogramos.

Masa del neutrón (mn): Es la masa de un neutrón, una partícula subatómica sin carga eléctrica. Su valor es aproximadamente $1.674927471 * 10^{**-27}$ kilogramos.

Constante de la ley de Coulomb (k_e): Es una constante que aparece en la ley de Coulomb, que describe la fuerza electrostática entre dos cargas eléctricas. Su valor es aproximadamente $8.9875517923 * 10^{**9} \text{ N m}^{**2}\text{/C}^{**2}$.

Constante de permeabilidad del vacío (μ_0): Es una constante que está relacionada con la magnetostática y determina la fuerza magnética entre corrientes eléctricas. Su valor es aproximadamente $4\pi * 10^{**-7} \text{ T m/A}$.

Constante de la ley de Stefan-Boltzmann (σ): Es una constante que aparece en la ley de Stefan-Boltzmann, que relaciona la radiación emitida por un cuerpo negro con su temperatura. Su valor es aproximadamente $5.670374419 * 10^{**-8} \text{ W/(m}^{**2}\text{ K}^{**4})$.

Tuplas y listas:

Son colecciones ordenadas, con índice y con objetos separados por comas.

Ordenadas

Con índice

Mutables

No tuplas

Si listas

Separados por comas.

Índex

En el contexto de una lista o tupla en Python, un índice se refiere a la posición numérica de un elemento dentro de la secuencia. Cada elemento en una lista o tupla tiene asignado un índice único que lo identifica iniciando en cero (0) y termina en el último N-1. El uso de len cuenta la cantidad de objetos desde 1 a N

En Python, los índices comienzan desde cero, lo que significa que el primer elemento de una lista o tupla tiene un índice de 0, el segundo elemento tiene un índice de 1, y así sucesivamente. Puedes acceder a un elemento específico de una lista o tupla utilizando su índice correspondiente.

Aquí tienes un ejemplo de cómo acceder a elementos individuales utilizando índices:

En el ejemplo anterior, se accede a elementos individuales de la lista mi_lista y la tupla mi_tupla utilizando corchetes [] seguidos del índice correspondiente.

Se debe tener en cuenta que los índices deben estar dentro del rango válido de la lista o tupla. Si intentas acceder a un índice que está fuera de ese rango, se producirá un error de índice fuera de rango (list index out of range).



```
lista = [1,2,3,"hola",99,3.14159,True, False]
print (f"el {lista} es de la Módulo {type (lista)=}")
print (f"el {lista[0]} es de la Módulo {type (lista[0])=}")
print (f"el {lista[1]} es de la Módulo {type (lista[1])=}")
print (f"el {lista[2]} es de la Módulo {type (lista[2])=}")
print (f"el {lista[3]} es de la Módulo {type (lista[3])=}")
print (f"el {lista[4]} es de la Módulo {type (lista[4])=}")
print (f"el {lista[5]} es de la Módulo {type (lista[5])=}")
print (f"el {lista[6]} es de la Módulo {type (lista[6])=}")
print (f"el {lista[7]} es de la Módulo {type (lista[7])=}")
```

#Salida esperada por consola

```
el lista =[1, 2, 3, 'hola', 99, 3.14159, True, False] es de la Módulo type (lista)=<class 'list'>
el lista[0]=1 es de la Módulo type (lista[0])=<class 'int'>
el lista[1]=2 es de la Módulo type (lista[1])=<class 'int'>
el lista[2]=3 es de la Módulo type (lista[2])=<class 'int'>
el lista[3]='hola' es de la Módulo type (lista[3])=<class 'str'>
el lista[4]=99 es de la Módulo type (lista[4])=<class 'int'>
el lista[5]=3.14159 es de la Módulo type (lista[5])=<class 'float'>
el lista[6]=True es de la Módulo type (lista[6])=<class 'bool'>
el lista[7]=False es de la Módulo type (lista[7])=<class 'bool'>
```

Métodos y atributos

```
mi_tupla = ("A","B","C","D","E","F")
print (f"el contenido de {mi_tupla} \t{type(mi_tupla)}")
print (f"los métodos disponibles son {dir(mi_tupla)}")
```

#Salida esperada por consola

```
el contenido de mi_tupla=(‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’) <class ‘tuple’>
los métodos disponibles son dir(mi_tupla)=[‘_add’, ‘_class’, ‘_contains’, ‘_delattr’, ‘_dir’, ‘_doc’, ‘_eq’, ‘_format’, ‘_ge’, ‘_getattribute’, ‘_getitem’, ‘_getnewargs’, ‘_getstate’, ‘_gt’, ‘_hash’, ‘_init’, ‘_init_subclass’, ‘_iter’, ‘_le’, ‘_len’, ‘_lt’, ‘_mul’, ‘_ne’, ‘_new’, ‘_reduce’, ‘_reduce_ex’, ‘_repr’, ‘_rmul’, ‘_setattr’, ‘_sizeof’, ‘_str’, ‘_subclasshook’, ‘count’, ‘index’]

['count', 'index']
```

```
mi_lista = ["A","B","C","D","E","F"]
print (f"el contenido de {mi_lista} \t{type(mi_lista)}")
print (f"los métodos disponibles son {dir(mi_lista)}")
```

#Salida esperada por consola

```
el contenido de mi_lista=[‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’] <class ‘list’>
los métodos disponibles son dir(mi_lista)=[‘_add’, ‘_class’, ‘_contains’, ‘_delattr’, ‘_delitem’, ‘_dir’, ‘_doc’, ‘_eq’, ‘_format’, ‘_ge’, ‘_getattribute’, ‘_getitem’, ‘_getstate’, ‘_gt’, ‘_hash’, ‘_iadd’, ‘_imul’, ‘_init’, ‘_init_subclass’, ‘_iter’, ‘_le’, ‘_len’, ‘_lt’, ‘_mul’, ‘_ne’, ‘_new’, ‘_reduce’, ‘_reduce_ex’, ‘_repr’, ‘_reversed’, ‘_rmul’, ‘_setattr’, ‘_setitem’, ‘_sizeof’, ‘_str’, ‘_subclasshook’, ‘append’, ‘clear’, ‘copy’, ‘count’, ‘extend’, ‘index’, ‘insert’, ‘pop’, ‘remove’, ‘reverse’, ‘sort’]

[‘append’, ‘clear’, ‘copy’, ‘count’, ‘extend’, ‘index’, ‘insert’, ‘pop’, ‘remove’, ‘reverse’, ‘sort’]
```

Métodos y atributos utilizables de forma simple son los que no poseen doble guion delante y detrás. Los dejaremos par más adelante como los constructores de módulos



Los métodos de las tuplas se incluyen dentro de los de las listas, por lo que solo es necesarios explicarlos una vez.

En adelante veremos los métodos y atributos de las listas y tuplas

'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort'

Len devuelve la cantidad de objetos que contienen en esta caso la colección

Separaremos los métodos por su acción:

- **Agregar datos a la lista (las tuplas son inmutables)**

- append() agrega un solo dato al final de la lista
- insert() agrega un solo dato en el lugar que se indique por un índice
- extend() agrega una colección de datos al final de la lista original

```
lista = [1,2,3,"hola",99,3.14159,True, False]
print (f"el contenido de original {lista}\n\tcantidad de elementos:{len(lista)}")
#
# agrego dato(s)
lista.append("chau")#al final
print (f"lista.append('chau'):\n{lista}\n\tcantidad de elementos:{len(lista)}")
#
lista.insert(2,"dato insertado")# en el lugar solicitado
print (f"lista.insert(2,'dato insertado'):{lista}\n\tcantidad de elementos:{len(lista)}")
#
# agrego una colección de datos
lista_ext = ["A","B","C","D","E","F"]
lista.extend(lista_ext)
print (f"lista.extend(lista_ext):\n{lista}\n\tcantidad de elementos:{len(lista)}")
```

#Salida esperada por consola

el contenido de original lista=[1, 2, 3, 'hola', 99, 3.14159, True, False]
 cantidad de elementos:8

lista.append('chau'):
 [1, 2, 3, 'hola', 99, 3.14159, True, False, 'chau']
 cantidad de elementos:9

lista.insert(2,'dato insertado'):
 [1, 2, 'dato insertado', 3, 'hola', 99, 3.14159, True, False,'chau']
 cantidad de elementos:10

lista.extend(lista_ext):
 [1, 2, 'dato insertado', 3, 'hola', 99, 3.14159, True, False, 'chau', 'A', 'B', 'C', 'D', 'E', 'F']
 cantidad de elementos:16

- **eliminar datos de una lista (las tuplas son inmutables)**

- pop() elimina un solo dato ubicado en el índice final de la lista
- pop(I) elimina un solo dato ubicado en el índice I de la lista
- remove(D) elimina el primer valor D que encuentra desde índice 0
- clear() elimina todo el contenido de la colección. (La colección sigue existiendo)



```

lista = [1,2,3,"hola",99,3.14159,True, False]
print (f"el contenido de original {lista} \n\tcantidad de elementos:{len(lista)}")
#
# elimino un dato
lista.pop()#por index () en el lugar solicitado ultimo
print (f"lista.pop():\n{lista}\n\tcantidad de elementos:{len(lista)}")
#
lista.pop(2)#por index (2) en el lugar solicitado
print (f"lista.pop(2):\n{lista}\n\tcantidad de elementos:{len(lista)}")
#
lista.remove("hola")# por contenido
print (f"lista.remove('hola'):\n{lista}\n\tcantidad de elementos:{len(lista)}")
#
lista.remove(2)# por contenido
print (f"lista.remove(2):\n{lista}\n\tcantidad de elementos:{len(lista)}")
#
lista.clear()# vació la lista
print (f"lista.clear ():\n{lista}\n\tcantidad de elementos:{len(lista)}")

#Salida esperada por consola
el contenido de original lista=[1, 2, 3, 'hola', 99, 3.14159, True, False]
    cantidad de elementos: 8
lista.pop():
[1, 2, 3, 'hola', 99, 3.14159, True]
    cantidad de elementos: 7
lista.pop(2):
[1, 2, 'hola', 99, 3.14159, True]
    cantidad de elementos: 6
lista.remove('hola'):
[1, 2, 99, 3.14159, True]
    cantidad de elementos: 5
lista.remove(2):
[1, 99, 3.14159, True]
    cantidad de elementos: 4
lista.clear ():
[]
    cantidad de elementos: 0

```

- **Organización de datos de una lista (las tuplas son inmutables)**
 - reverse() invierte el orden de la lista
 - sort() ordena la lista de mayor a menor
 - lista.sort(reverse=True) elimina el primer valor D que encuentra desde índice 0

Tengan en cuenta que sort ordena de mayor a menor objetos numéricos y por cantidad de caracteres en objetos string.

Si se utiliza listas con objetos de ambos tipos se obtendrá un error

```

lista = [1,2,3,4,5,6,7,8,9,8,7,6,5,6,7,8,9,11,2]
print (f"el contenido de original {lista} \n\tcantidad de elementos:{len(lista)}")
#
lista.reverse()
print (f"lista.reverse:\n{lista}\n\tcantidad de elementos:{len(lista)}")
#
lista.sort()

```



```
print (f"lista sort:\n{lista}\n\tcantidad de elementos:{len(lista)}")
#-----
print (f"lista sort(reverse=True):\n{lista}\n\tcantidad de elementos:{len(lista)}")

#Salida esperada por consola
el contenido de original lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 8, 7, 6, 5, 6, 7, 8, 9, 11, 2]
    cantidad de elementos:19
lista.reverse():
[2, 11, 9, 8, 7, 6, 5, 6, 7, 8, 9, 8, 7, 6, 5, 4, 3, 2, 1]
    cantidad de elementos:19
lista sort:
[1, 2, 2, 3, 4, 5, 5, 6, 6, 6, 7, 7, 7, 8, 8, 8, 9, 9, 11]
    cantidad de elementos:19
lista sort(reverse=True):
[11, 9, 9, 8, 8, 8, 7, 7, 7, 6, 6, 6, 5, 5, 4, 3, 2, 2, 1]
    cantidad de elementos:19
```

Información sobre el contenido de datos de una lista o una tupla.

Estos Atributos no modifican el contenido y son usados en ambas colecciones

- count() invierte el orden de la lista
- index() ordena la lista de mayor a menor

```
lista = [1,2,3,4,5,6,7,8,9,8,7,6,5,6,7,8,9,11,2]
print (f"el contenido de original {lista} \n\tcantidad de elementos:{len(lista)}")
#-----
print (f"lista.count(9) está {lista.count(9)} veces en la lista.\n\tcantidad de elementos:{len(lista)}")
print (f"lista.count(11) está {lista.count(11)} veces en la lista.\n\tcantidad de elementos:{len(lista)}")
#-----
print (f"lista.index(3) está en el índice:{lista.index(3)}\n\tcantidad de elementos:{len(lista)}")
#-----
```

```
el contenido de original lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 8, 7, 6, 5, 6, 7, 8, 9, 11, 2]
    cantidad de elementos:19
lista.count(9) esta 2 veces en la lista.
    cantidad de elementos:19
lista.count(11) esta 1 veces en la lista.
    cantidad de elementos:19
lista.index(3) está en el índice:2
    cantidad de elementos:19
```

Slicing

El slicing en Python es una técnica que te permite extraer una porción (subsecuencia) de una secuencia, como una cadena, una lista o una tupla. Permite seleccionar un rango de elementos utilizando un inicio, un fin y un paso.

La sintaxis básica del slicing es la siguiente:

secuencia[inicio:limite_final]

inicio: El índice donde comienza el slice que se necesitar (este inice si se incluye).

limite_final: El índice limite final donde termina la porción que deseas extraer.

Al ser un límite, este valor nunca se llega, por lo que no se incluye en el slice que se necesita.



secuencia[inicio:limite_final:step]

Step o paso (opcional): El incremento entre los índices seleccionados. Por defecto es 1.

Aquí tienes algunos ejemplos para ilustrar cómo funciona el slicing:

```

cadena = "Python-es-Genial"
print(f"{{cadena[0:6]}}| Extrae los caracteres desde el índice 0 al 6 (5)"# |Python|
print(f"{{cadena[:6]}}| Extrae los caracteres desde el inicio al 6 (5)"# |Python|
print(f"{{cadena[-10]}}| Extrae los caracteres desde el inicio al -10"# |Python|
print(f"{{cadena[6:16]}}| Extrae los caracteres desde el índice 7 hasta el 16"# |-es-Genial|
print(f"{{cadena[6:]}}| Extrae los caracteres desde el índice 7 hasta el final"# |-es-Genial|
print(f"{{cadena[-10:]}}| Extrae los caracteres desde el índice -10 hasta el final"# |-es-Genial|
print(f"{{cadena[6:10]}}| Extrae los caracteres desde el inicio 6 hasta el 10 (9)"# |-es-|
print(f"{{cadena[-10:-6]}}| Extrae los caracteres desde el inicio -10 hasta el-6(-7)"# |-es-|
print(f"{{cadena[::2]}}| Extrae los caracteres con un paso de 2"# |Pto-sGna|
print(f"{{cadena[::-1]}}| Invierte el orden") # | laineG-se-nohtyP|
```

|Python| Extrae los caracteres desde el índice 0 al 6 (5)
|Python| Extrae los caracteres desde el inicio al 6 (5)
|Python| Extrae los caracteres desde el inicio al -10
|-es-Genial| Extrae los caracteres desde el índice 7 hasta el 16
|-es-Genial| Extrae los caracteres desde el índice 7 hasta el final
|-es-Genial| Extrae los caracteres desde el índice -10 hasta el final
|-es-| Extrae los caracteres desde el inicio 6 hasta el 10 (9)
|-es-| Extrae los caracteres desde el inicio -10 hasta el-6(-7)
|Pto-sGna| Extrae los caracteres con un paso de 2
|laineG-se-nohtyP| Invierte el orden

Recuerda que el índice de inicio y fin pueden ser números negativos, lo que indica que se cuentan desde el final de la secuencia.

El slicing en Python es una herramienta muy útil para manipular secuencias y extraer partes específicas según tus necesidades. Salida esperada por consola

Pilas y colas

FIFO First In > Fist Out

LIFO Last In > First Out –o- First In > Last Out

```

append()
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16]
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16, 17]
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16, 17, 18]
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16, 17, 18, 19]
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16, 17, 18, 19, 20]
pop()
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16, 17, 18, 19]
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16, 17, 18]
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16, 17]
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15, 16]
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15]
pop(0) inicio
lista=[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15]
lista=[3, 4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15]
lista=[4, 5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15]
```



```
lista=[5, 6, 7, 8, 9, 10, 11, 112, 13, 14, 15]  
lista=[6, 7, 8, 9, 10, 11, 112, 13, 14, 15]
```



Matrices - Pseudomatrices:

Académicamente, una matriz es una estructura matemática bidimensional compuesta por números, símbolos o expresiones dispuestos en filas y columnas. La notación común para denotar una matriz es utilizando. Formalmente, una matriz se denota por una letra mayúscula, como A, B, o C, y sus elementos individuales se denotan por letras minúsculas con índices.

Atributos Matemáticos de una Matriz:

Número de Filas y Columnas:

Una matriz con m filas y n columnas se denota como una matriz $m \times n$. Por ejemplo, una matriz A con 2 filas y 3 columnas sería una matriz 2×3 .

Elementos:

Cada número individual en la matriz se llama elemento. En la matriz A, a_{ij} representa el elemento en la fila i y la columna j.

Dimensión:

La dimensión de una matriz es el par ordenado (m, n) , donde m es el número de filas y n es el número de columnas.

Matriz Cuadrada:

Una matriz cuadrada tiene el mismo número de filas y columnas ($m = n$).

Matriz Nula:

Una matriz nula es una matriz en la que todos sus elementos son cero.

Matriz Identidad:

Una matriz identidad es una matriz cuadrada con unos en la diagonal principal y ceros en el resto de los elementos.

Suma y Resta de Matrices:

Las matrices se pueden sumar o restar si tienen la misma dimensión.

Producto de Matrices:

El producto de dos matrices A y B está definido si el número de columnas de A es igual al número de filas de B.

Matriz Transpuesta:

La transpuesta de una matriz se obtiene intercambiando sus filas por columnas.

Determinante:

El determinante de una matriz cuadrada es un número escalar que proporciona información sobre las propiedades de la matriz.

Inversa:

Una matriz cuadrada A tiene una matriz inversa A^{-1} si el producto de A y A^{-1} es igual a la matriz identidad.

Submatriz:

Una submatriz es una matriz obtenida eliminando algunas filas y/o columnas de una matriz más grande.

Matemáticamente, una matriz es una colección rectangular de números, símbolos o expresiones dispuestos en filas y columnas.

Para que una colección sea considerada una matriz, debe cumplir con ciertas características fundamentales:

Estructura Bidimensional:

Una matriz es una estructura bidimensional, lo que significa que tiene filas y columnas. Cada elemento en la matriz se identifica por su posición en una fila y una columna específicas.

Elementos Numerados:

Los elementos de la matriz deben ser números o expresiones matemáticas. Cada elemento está etiquetado por dos índices, uno para la fila y otro para la columna.

Filas de Longitud Uniforme:

Todas las filas de la matriz deben tener la misma longitud. En otras palabras, el número de elementos en



cada fila es constante.

Propiedades de Dimensión:

La dimensión de una matriz se describe mediante dos números: el número de filas (m) y el número de columnas (n). Una matriz m x n tiene m filas y n columnas.

Ejemplo Notacional:

Si A es una matriz, sus elementos se denotan por a_{ij} , donde i es el índice de la fila y j es el índice de la columna.

Operaciones Definidas:

Deben estar definidas operaciones matriciales básicas, como la suma y resta de matrices, el producto de una matriz por un escalar, el producto de matrices, entre otras.

Ejemplo de una matriz matemática 3x3:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Ejemplo de una matriz programación 3x3:

$$A = \begin{bmatrix} [A_{00}, A_{01}, A_{02}], \\ [A_{10}, A_{11}, A_{12}], \\ [A_{20}, A_{21}, A_{22}] \end{bmatrix}$$

En resumen, para que una colección sea considerada matemáticamente como una matriz, debe ser bidimensional, contener elementos numerados dispuestos en filas y columnas, tener filas de longitud uniforme y admitir operaciones matriciales básicas.

```

lista_anidada = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
          o
lista_anidada = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

print(lista_anidada[0]) # Salida: [1, 2, 3]
print(lista_anidada[1][2]) # Salida: 6

```

Diferencias entre Listas Anidadas y Matrices Aritméticas:

Operaciones Aritméticas:

Las matrices aritméticas suelen estar diseñadas para realizar operaciones matriciales directamente, como la multiplicación de matrices. En cambio, las listas anidadas en Python requieren programación adicional para realizar operaciones matriciales.

Tipos de Datos:

En una matriz aritmética, los elementos suelen ser números y las operaciones se realizan de acuerdo con las reglas matriciales. En listas anidadas, los elementos pueden ser de cualquier tipo, y las operaciones no siguen automáticamente las reglas de las matrices.

Dimensionalidad Implícita:

En las matrices aritméticas, la dimensionalidad se define de manera explícita, por ejemplo, una matriz 3x3. En listas anidadas, la dimensionalidad puede variar; una fila puede tener una longitud diferente a otra.

Acceso a Elementos:



El acceso a los elementos en las listas anidadas se realiza mediante índices dobles. En una matriz aritmética, la notación de índices puede no ser aplicable, ya que las matrices suelen ser representadas por un único índice.

Librerías Específicas:

Para operaciones matriciales eficientes, es común utilizar librerías específicas como NumPy en Python para matrices aritméticas. Estas librerías están optimizadas para realizar operaciones matriciales de manera eficiente.

Una matriz anidada NO tiene las características de una Matriz verdadera. Para poder generar un objeto con las propiedades reales de una matriz verdadera tengo que llamar a alguna librería que proporcione los métodos y atributos antes mencionados (ver array, numpy, pandas)

Más adelante, en análisis de datos deberemos usar matrices verdaderas y mientras tanto las listas anidadas en Python ofrecen flexibilidad para representar estructuras de datos multidimensionales, las matrices aritméticas suelen estar diseñadas específicamente para operaciones matriciales eficientes y siguen reglas matriciales más estrictas.

La elección entre ellas depende de los requisitos específicos del problema y las operaciones que se planean realizar.



Conjuntos:

En programación, un conjunto (set) es una colección no ordenada y mutable de elementos únicos. La característica principal de un conjunto es que no puede contener elementos duplicados, y su orden no se garantiza. Los conjuntos son útiles cuando necesitas almacenar elementos sin preocuparte por el orden y la duplicación.

Los conjuntos (sets) en el análisis de datos pueden ser útiles principalmente cuando se necesita realizar operaciones de conjunto para explorar la relación entre distintos conjuntos de datos.

Eliminación de Duplicados:

Los conjuntos son útiles para eliminar duplicados de un conjunto de datos. Al convertir una lista o una columna de un DataFrame a un conjunto, automáticamente se eliminan los duplicados.

Operaciones de Conjuntos para Explorar Relaciones:

Se pueden usar operaciones de conjuntos (unión, intersección, diferencia) para explorar relaciones entre conjuntos de datos, especialmente cuando trabajas con múltiples conjuntos de datos.

```
conjunto_a = {1, 2, 3, 4}
conjunto_b = {3, 4, 5, 6}
union = conjunto_a | conjunto_b # Unión
interseccion = conjunto_a & conjunto_b # Intersección
diferencia = conjunto_a - conjunto_b # Diferencia
```

Comprobación de Pertenencia Rápida:

Los conjuntos son eficientes para verificar la pertenencia de un elemento. Esto puede ser útil cuando necesitas verificar si un elemento específico está presente en un conjunto grande de datos.

Filtrado de Datos:

Puedes utilizar conjuntos para filtrar datos basados en ciertos criterios. Por ejemplo, puedes tener un conjunto de usuarios que cumplen con ciertas condiciones y filtrar un conjunto de datos para incluir solo esos usuarios.

```
usuarios_interesantes = {usuario.id for usuario in usuarios if usuario.condicion}
conjunto_filtrado = {dato for dato in conjunto_datos if dato.id in usuarios_interesantes}
```

Comprobación de Unicidad en Columnas de DataFrame:

Cuando trabajas con DataFrames en bibliotecas como Pandas, puedes aprovechar los conjuntos para verificar la unicidad de los valores en una columna.

```
import pandas as pd
df = pd.DataFrame({'columna': [1, 2, 3, 2, 4, 5, 1]})
es_unico = len(set(df['columna'])) == len(df['columna'])
```

Tratamiento de Categorías Únicas:

Los conjuntos pueden ser útiles al trabajar con variables categóricas para identificar todas las categorías únicas presentes en un conjunto de datos.

```
categorias_unicas = set(conjunto_datos['categoria'])
```

Consideraciones Importantes:

Los conjuntos son mutables, por lo que si necesitas inmutabilidad, debes convertirlos a frozenset.

Los conjuntos pueden ser más eficientes que las listas cuando se trata de operaciones de pertenencia, especialmente en conjuntos de datos grandes.

Métodos y Atributos:

(Teniendo en cuenta lo que es la tupla a la lista es el frozenset al set)

Sets: add, clear, copy, difference, difference_update, discard, intersection, intersection_update, isdisjoint, issubset, issuperset, pop, remove, symmetric_difference, symmetric_difference_update, union, update

FrozenSets: copy, difference, intersection, isdisjoint, issubset, issuperset, symmetric_difference, union



Sets algunos Métodos v Atributos:

```
mi_conjunto = { 1,2,3,4,5,6,7,8,9,5,1,2,3,5,7,4,8,6,2,0}
```

salida esperada

```
mi_conjunto={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 987}
```

add(elemento):

Agrega un elemento al conjunto. Si el elemento ya está presente, no se realiza ninguna acción.

```
mi_conjunto.add(987)
```

```
print(f'mi_conjunto={mi_conjunto}'")
```

salida esperada

```
mi_conjunto={0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 987}
```

remove(elemento):

Elimina un elemento específico del conjunto. Si el elemento no está presente, se genera un error.

```
mi_conjunto.remove(0)
```

```
print(f'mi_conjunto={mi_conjunto}'")
```

salida esperada

```
mi_conjunto = { 1,2,3,4,5,6,7,8,9,5,1,2,3,5,7,4,8,6,2,0}
```

discard(elemento):

Elimina un elemento específico del conjunto. Si el elemento no está presente, no se produce un error.

```
mi_conjunto.discard(4)
```

```
print(f'mi_conjunto={mi_conjunto}'")
```

salida esperada

```
mi_conjunto = { 1,2,3,4,5,6,7,8,9,5,1,2,3,5,7,4,8,6,2,0}
```

pop():

Elimina y devuelve un elemento aleatorio del conjunto. Si el conjunto está vacío, se genera un error.

```
mi_conjunto.pop()
```

```
print(f'mi_conjunto={mi_conjunto}'")
```

salida esperada

```
mi_conjunto = { 1,2,3,4,5,6,7,8,9,5,1,2,3,5,7,4,8,6,2,0}
```

clear():

Elimina todos los elementos del conjunto, dejándolo vacío.

```
mi_conjunto.clear()
```

```
print(f'mi_conjunto={mi_conjunto}'")
```

salida esperada

```
mi_conjunto = { }
```

union(otro_conjunto):

Devuelve un nuevo conjunto que contiene todos los elementos de ambos conjuntos (unión)

```
conjunto1 = {1, 2, 3}
```

```
conjunto2 = {3, 4, 5}
```

```
union = conjunto1.union(conjunto2)
```

```
print(f'conjunto union={union}'")
```

salida esperada

```
conjunto union={1, 2, 3, 4, 5}
```



intersection(otro_conjunto):(intersección)

Devuelve un nuevo conjunto que contiene los elementos comunes de ambos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
interseccion = conjunto1.intersection(conjunto2)
print(f'conjunto intersección={interseccion}')

# salida esperada
conjunto intersección={3}
```

difference(otro_conjunto):(diferencia).

Devuelve un nuevo conjunto que contiene los elementos en el primer conjunto pero no en el segundo

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
diferencia = conjunto1.difference(conjunto2)
print(f'mi conjunto diferencia={diferencia}')

# salida esperada
mi conjunto diferencia={1, 2}
```

issubset(otro_conjunto):

Verifica si todos los elementos del conjunto están presentes en otro conjunto.

```
conjunto1 = {1, 2}
conjunto2 = {1, 2, 3, 4}
es_subconjunto = conjunto1.issubset(conjunto2)
print(f'mi conjunto es_subconjunto={es_subconjunto}')

# salida esperada
```

issuperset(otro_conjunto):

Verifica si todos los elementos de otro conjunto están presentes en el conjunto.

```
conjunto1 = {1, 2, 3, 4}
conjunto2 = {1, 2}
es_superconjunto = conjunto1.issuperset(conjunto2)
print(f'mi conjunto es_superconjunto={es_superconjunto}')

# salida esperada
```

```
union = conjunto1 | conjunto2      # Unión
interseccion = conjunto1 & conjunto2 # Intersección
diferencia = conjunto1 - conjunto2 # Diferencia
```

Atributos de Conjunto:

No hay índices:

A diferencia de las listas o tuplas, los conjuntos no admiten el acceso mediante índices, ya que no tienen un orden específico.

No hay elementos duplicados:

Los conjuntos no permiten elementos duplicados. Si intentas agregar un elemento que ya está presente, no se realizará ninguna acción.

Mutabilidad:

Los conjuntos son mutables, lo que significa que puedes agregar, eliminar y actualizar elementos después de haber creado el conjunto.

Operaciones de Conjuntos:

Los conjuntos admiten operaciones matemáticas de conjuntos, como unión, intersección y diferencia, que son implementadas mediante métodos específicos.



Diccionarios:

Un diccionario en programación se define como un objeto o estructura de datos que almacena pares clave-valor. Es una implementación de una tabla hash, que proporciona un acceso eficiente a los valores almacenados a través de claves únicas. Los diccionarios son conocidos por varios nombres en diferentes lenguajes de programación, como "mapas", "tablas de búsqueda" o "asociativos".

Clave y Valor:

Cada elemento en un diccionario consiste en un par clave-valor. La clave es una etiqueta única que se utiliza para acceder al valor asociado. La clave puede ser de cualquier tipo inmutable (como cadenas, números o tuplas). El valor puede ser de cualquier tipo.

Estructura de Diccionario:

A nivel conceptual, un diccionario se representa como una colección no ordenada de pares clave-valor. La implementación subyacente utiliza técnicas de hash para proporcionar un acceso eficiente a los valores a través de las claves.

Operaciones Básicas:

Las operaciones básicas en un diccionario incluyen la inserción de pares clave-valor, la recuperación de un valor mediante su clave, la actualización de valores existentes y la eliminación de pares clave-valor.

Iteración:

Se puede iterar sobre las claves, los valores o los pares clave-valor en un diccionario. Esto permite recorrer todos los elementos almacenados.

Eficiencia:

La eficiencia de acceso en diccionarios es constante en promedio, ya que el uso de técnicas de hash permite la búsqueda rápida de valores asociados a claves.

Dict: 'clear', 'copy', 'fromkeys', 'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values'

```

mi_diccionario = {
    'nombre': 'Ariel',
    'apellido': 'García',
    'edad': 50,
    'ciudad': 'Buenos Aires',
    'pais': 'Argentina'
}
# Acceder a un valor mediante clave
print(f'mi_diccionario["nombre"]={mi_diccionario["nombre"]}')
#
# Modificar un valor
mi_diccionario['edad'] = 26
#
# Iterar sobre las claves y valores
for clave, valor in mi_diccionario.items():
    print(f'\t{clave}: {valor}')
mi_diccionario['nombre']=Ariel
    clave='nombre': valor='Ariel'
    clave='apellido': valor='García'
    clave='edad': valor=26
    clave='ciudad': valor='Buenos Aires'
    clave='pais': valor='Argentina'

```



Diccionarios Métodos y Atributos:

```
mi_diccionario = {
    'nombre': 'Ariel',
    'apellido': 'García',
    'edad': 50,
    'ciudad': 'Buenos Aires',
    'pais': 'Argentina'
}
```

clear():

Elimina todos los elementos del diccionario.

```
mi_diccionario.clear()
print(f"mi_diccionario={mi_diccionario}")

mi_diccionario={}
```

copy():

Devuelve una copia superficial (shallow copy) del diccionario.

```
copia_diccionario = mi_diccionario.copy()
print(f"copia_diccionario={copia_diccionario}")

copia_diccionario={'nombre': 'Ariel', 'apellido': 'García', 'edad': 50, 'ciudad': 'Buenos Aires', 'pais': 'Argentina'}
```

get(key, default=None):

Devuelve el valor asociado con la clave proporcionada. Si la clave no existe, devuelve el valor predeterminado (que es None si no se proporciona).

```
valor = mi_diccionario.get('nombre', 'No encontrado')
print(f"valor de la clave 'nombre'={valor}")
valor = mi_diccionario.get('name', 'No encontrado')
print(f"valor de la clave 'name'={valor}")

valor de la clave 'nombre'=Ariel
valor de la clave 'name'=No encontrado
```

pop(key, default=None):

Elimina y devuelve el valor asociado con la clave proporcionada. Si la clave no existe y se proporciona un valor predeterminado, devuelve ese valor; de lo contrario, genera un error.

```
valor = mi_diccionario.pop('nombre', 'No encontrado')
print(f"mi_diccionario={mi_diccionario}")

mi_diccionario={'apellido': 'García', 'edad': 50, 'ciudad': 'Buenos Aires', 'pais': 'Argentina'}
```

popitem():

Elimina y devuelve el último par clave-valor insertado.

```
par = mi_diccionario.popitem()
print(f"par eliminado={par}")
print(f"mi_diccionario={mi_diccionario}")

par eliminado=('pais', 'Argentina')
mi_diccionario={'apellido': 'García', 'edad': 50, 'ciudad': 'Buenos Aires'}
```



update(iterable):

Actualiza el diccionario con elementos de otro diccionario o de un iterable de pares clave-valor.

```
otro_diccionario = {'documento': '23 456 789'}
mi_diccionario.update(otro_diccionario)
print(f'mi_diccionario={mi_diccionario}'")
```

```
mi_diccionario={'apellido': 'García', 'edad': 50, 'ciudad': 'Buenos Aires', 'documento': '23 456 789'}
```

keys():

Devuelve una vista de las claves en el diccionario.

```
claves = mi_diccionario.keys()
print(f'mi_diccionario.keys()={claves}')
mi_diccionario.keys()=dict_keys(['apellido', 'edad', 'ciudad', 'documento'])
```

values():

Devuelve una vista de los valores en el diccionario.

```
valores = mi_diccionario.values()
print(f'mi_diccionario.values ()={valores}')
mi_diccionario.values ()=dict_values(['García', 50, 'Buenos Aires', '23 456 789'])
```

items():

Devuelve una vista de pares clave-valor como tuplas.

```
pares = mi_diccionario.items()
print(f'{pares=}')

for clave, valor in mi_diccionario.items():
    print(f'{clave}: {valor}')

pares=dict_items([('apellido', 'García'), ('edad', 50), ('ciudad', 'Buenos Aires'), ('documento', '23 456 789')])
apellido: García
edad: 50
ciudad: Buenos Aires
documento: 23 456 789
```



Módulo 6 – modificación de flujo:

bucle for:

	<ul style="list-style-type: none"> • for dato in colección: # bloque # de iteración
	<ul style="list-style-type: none"> • break • continue
	<ul style="list-style-type: none"> • anidación
	<ul style="list-style-type: none"> • enumerate • zip

For:

El modificador de flujo for en Python se utiliza para realizar iteraciones sobre elementos de una secuencia o cualquier objeto iterable. Este bucle es una construcción fundamental en el lenguaje y se utiliza para **recorrer** elementos en listas, tuplas, string, diccionarios y otros tipos de datos iterables.

Colecciones:

Una colección en Python es un tipo de objeto que puede contener múltiples elementos u objetos. Estas colecciones son utilizadas para almacenar, organizar y manipular conjuntos de datos de manera eficiente.

Hay diferentes tipos de colecciones, unas integradas en su biblioteca estándar (que son ampliamente utilizadas en programación) y otras a través de.

Iterables:

Un iterable en programación se refiere a un objeto capaz de proporcionar una secuencia de elementos de uno en uno. Estos elementos pueden ser accedidos en un orden específico, y el objeto iterable facilita la iteración sobre sus elementos. En muchos lenguajes de programación, la iteración se logra mediante el uso de bucles, como el bucle for.

En el contexto de Python, un iterable es cualquier objeto sobre el cual se puede iterar. Para que un objeto sea iterable, debe implementar el método especial `__iter__()` que devuelve un objeto iterador. Un iterador, a su vez, debe implementar el método `__next__()` que proporciona el siguiente elemento de la secuencia.

En resumen: un iterable es un objeto que se puede recorrer o iterar elemento por elemento.

Básicamente, cualquier objeto que pueda devolver uno a la vez sus elementos cuando se utiliza en un bucle for se considera un iterable.

Toda colección es un iterable pero no todo iterable es una colección:

Toda colección es un iterable:

Una colección, como una lista, tupla, conjunto o diccionario en Python, es un tipo de estructura de datos que agrupa elementos bajo un mismo nombre o identificador.

Todas las colecciones en Python son iterables, lo que significa que se pueden recorrer elemento por



elemento utilizando bucles for u otras construcciones de iteración.

No todo iterable es una colección:

Aunque todas las colecciones son iterables, existen otros tipos de objetos que también son iterables, pero no se consideran colecciones en el sentido tradicional.

Ejemplos de iterables que no son colecciones incluyen rangos, cadenas de texto, generadores, y archivos.

Un rango (range), que es un iterable, no es una colección porque no almacena todos los elementos en la memoria de una vez, sino que los genera a medida que son necesarios.

Un string (str) - una cadena de texto es iterable, no se considera una colección, pero es inmutable, lo que significa que no se pueden modificar sus elementos.

Estructura for range

Inicialización:

Antes de que el bucle comience, se inicializa la variable de control. En Python, esta inicialización se realiza mediante la función range() o utilizando una secuencia iterable (como una lista o tupla).

Iteración:

El bucle for itera sobre los elementos de la secuencia. En cada iteración, la variable de control toma el valor del siguiente elemento de la secuencia.

Cuerpo del Bucle:

El cuerpo del bucle contiene las instrucciones que se ejecutan en cada iteración. Estas instrucciones son las que se repiten en cada ciclo del bucle.

Actualización de la Variable de Control:

Después de ejecutar el cuerpo del bucle, la variable de control se actualiza automáticamente al siguiente elemento de la secuencia. Este proceso se repite hasta que todos los elementos de la secuencia han sido procesados.

Finalización:

Una vez que todos los elementos de la secuencia han sido procesados, el bucle for se completa y la ejecución del programa continúa con la siguiente instrucción después del bucle.

Modificación de flujo Bucles: for:

```
for dato in colección:  
    # bloque  
    # de iteración  
    • break  
    • continue  
    anidación  
    • enumerate  
    • zip
```

Ejemplo de bucle for que imprime números del 1 al 5

```
for numero in range(1, 6):  
    print(f"el valor de {numero} =")
```

```
el valor de numero =1  
el valor de numero =2  
el valor de numero =3  
el valor de numero =4  
el valor de numero =5
```

Lista = [9,5,1,7,4]

```
for numero in lista:
```



```
print(f"el valor de {numero =}")
```

```
el valor de numero =9
```

```
el valor de numero =5
```

```
el valor de numero =1
```

```
el valor de numero =4
```



break

Cuando se encuentra la instrucción **break** dentro de un bucle (for o while), el bucle se interrumpe inmediatamente, y la ejecución del programa continúa con la siguiente instrucción después del bucle.

```
lista=[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
for numero in lista:
    print(f"el valor de {numero} =")
    if numero == 6:
        print("break")
        break

# salida por consola
    el valor de numero =1
    el valor de numero =2
    el valor de numero =3
    el valor de numero =4
    el valor de numero =5
    el valor de numero =6
break
```

continue

Continue se utiliza dentro de bucles (for o while) para omitir el resto del código en una iteración actual y pasar a la siguiente iteración del bucle. Cuando se encuentra la instrucción continue, el flujo de control salta inmediatamente a la siguiente iteración del bucle, ignorando cualquier código que quede en la iteración actual.

```
lista=[1, 2, 3, 4, 5, 6]
for numero in lista:
    print(f"el valor de {numero} =")
    if numero%2 == 0:
        print("\t\tcontinue x par")
        continue
    print("\timpair")

# salida por consola
el valor de numero =1
    impair
el valor de numero =2
    continue x par
el valor de numero =3
    impair
el valor de numero =4
    continue x par
el valor de numero =5
    impair
el valor de numero =6
    continue x par
```

.



Anidación

La anidación en programación se refiere a la práctica de incluir una estructura dentro de otra. Esto puede aplicarse a varios elementos, como bucles dentro de bucles. La anidación permite organizar y estructurar el código de manera jerárquica, facilitando la comprensión y mantenimiento del código

```
rango_ext=range(0,10,1)
rango_int=range(0,10,1)
for externo in rango_ext:
    print(f"\t\t", end=" | ")
    for interno in rango_int:
        print(f"\t{externo} - {interno}", end=" | ")
    print()
```

```
0 - 0 | 0 - 1 | 0 - 2 | 0 - 3 | 0 - 4 | 0 - 5 | 0 - 6 | 0 - 7 | 0 - 8 | 0 - 9 |
1 - 0 | 1 - 1 | 1 - 2 | 1 - 3 | 1 - 4 | 1 - 5 | 1 - 6 | 1 - 7 | 1 - 8 | 1 - 9 |
2 - 0 | 2 - 1 | 2 - 2 | 2 - 3 | 2 - 4 | 2 - 5 | 2 - 6 | 2 - 7 | 2 - 8 | 2 - 9 |
3 - 0 | 3 - 1 | 3 - 2 | 3 - 3 | 3 - 4 | 3 - 5 | 3 - 6 | 3 - 7 | 3 - 8 | 3 - 9 |
4 - 0 | 4 - 1 | 4 - 2 | 4 - 3 | 4 - 4 | 4 - 5 | 4 - 6 | 4 - 7 | 4 - 8 | 4 - 9 |
5 - 0 | 5 - 1 | 5 - 2 | 5 - 3 | 5 - 4 | 5 - 5 | 5 - 6 | 5 - 7 | 5 - 8 | 5 - 9 |
6 - 0 | 6 - 1 | 6 - 2 | 6 - 3 | 6 - 4 | 6 - 5 | 6 - 6 | 6 - 7 | 6 - 8 | 6 - 9 |
7 - 0 | 7 - 1 | 7 - 2 | 7 - 3 | 7 - 4 | 7 - 5 | 7 - 6 | 7 - 7 | 7 - 8 | 7 - 9 |
8 - 0 | 8 - 1 | 8 - 2 | 8 - 3 | 8 - 4 | 8 - 5 | 8 - 6 | 8 - 7 | 8 - 8 | 8 - 9 |
9 - 0 | 9 - 1 | 9 - 2 | 9 - 3 | 9 - 4 | 9 - 5 | 9 - 6 | 9 - 7 | 9 - 8 | 9 - 9 |
```

.Enumerate

Enumerate es una función que se utiliza para iterar sobre una secuencia (como una lista, una tupla o una cadena) al mismo tiempo que se obtienen tanto el valor como su índice correspondiente. Proporciona una forma conveniente de realizar un seguimiento del índice mientras se recorre una secuencia.

```
lista = ["A","B","C","D","E","F"]
for index, caracter in enumerate(lista):
    print(f"el valor de {caracter} se encuentra en el index {index}")
```

```
el valor de caracter ='A' se encuentra en el index 0
el valor de caracter ='B' se encuentra en el index 1
el valor de caracter ='C' se encuentra en el index 2
el valor de caracter ='D' se encuentra en el index 3
el valor de caracter ='E' se encuentra en el index 4
el valor de caracter ='F' se encuentra en el index 5
```

Zip

Zip es una función que se utiliza para combinar elementos de varias secuencias en una sola secuencia de tuplas. Toma como argumentos una o más secuencias y devuelve un iterador que produce tuplas donde cada tupla contiene los elementos correspondientes de las secuencias de entrada.

```
lista_c= ["A","B","C","D","E","F"]
lista_n= [1, 2, 3, 4, 5, 6]
for caracter, numero in zip(lista_c,lista_n):
```



```
print(f"{{caracter =} en la 'lista_c' se encuentra en la misma posición (index) que {numero} en la 'lista_n'")  
caracter ='A' en la 'lista_c' se encuentra en la misma posición (index) que 1 en la 'lista_n'  
caracter ='B' en la 'lista_c' se encuentra en la misma posición (index) que 2 en la 'lista_n'  
caracter ='C' en la 'lista_c' se encuentra en la misma posición (index) que 3 en la 'lista_n'  
caracter ='D' en la 'lista_c' se encuentra en la misma posición (index) que 4 en la 'lista_n'  
caracter ='E' en la 'lista_c' se encuentra en la misma posición (index) que 5 en la 'lista_n'  
caracter ='F' en la 'lista_c' se encuentra en la misma posición (index) que 6 en la 'lista_n'
```



Módulo 7 –Operadores:

- Operadores Aritméticos:
- Operadores Aritméticos sobre strings:
- Operadores de asignación:
 - Operador Walrus:
- Operadores de comparación:
- Operadores lógicos o booleanos:
- Operadores de identidad:
- Operadores de membresía o pertenencia:
- Operadores de asignación a nivel de bits:



Operadores:

Un operador es un símbolo especial que invoca métodos de los objetos. Los operandos son los valores (objetos) sobre los cuales actúa el operador. Los operadores son fundamentales para llevar a cabo diversas funciones, desde operaciones aritméticas hasta comparaciones y asignaciones.

Es crucial señalar que estos métodos especiales deben estar definidos en las clases de los objetos a los cuales se aplican estos operadores para que funcionen de manera adecuada.

Operadores Aritméticos:

operando (objetos numéricos)	
+	Suma - adiciona dos operandos.
-	Resta - sustracción al valor del operando de la izquierda el valor del de la derecha. Cambia el signo sobre un único operador.
/	Divide el operando de la izquierda por el de la derecha. La salida siempre es un float (genera un casting)
//	división entera se obtiene el cociente 'entero' de dividir el operando de la izquierda por el de la derecha.
&	Módulo es el residuo que se obtiene el resto de dividir el entero del operando de la izquierda por el de la derecha.
*	Producto - Multiplicación de dos operandos.
**x	Potencia - Exponenciación eleva el operando de la izquierda a la potencia del operador del de la derecha.
**(1/x)	Radicación eleva el operando de la izquierda a la potencia de 1 (uno) dividido el valor operador del de la derecha.
operador aritmético entre objetos numéricos Python	

Operadores Aritméticos:

- El operador + llama al método `__add__`.
- `resultado = int.__add__(numero1, numero2)`
- El operador - llama al método `__sub__`.
- El operador * llama al método `__mul__`.
- El operador / llama al método `__truediv__`.
- El operador // llama al método `__floordiv__`.
- El operador ** llama al método `__pow__`.
- El operador & (operador de bits AND) llama al método `__and__`.
- Es importante destacar que estos métodos especiales deben estar definidos en las clases de los objetos sobre los cuales se aplican estos operadores para que funcionen correctamente.

Operadores Aritméticos sobre strings:

operador aritmético entre objetos strings de Python	
+	Concatena dos Strings.
*	Replica un string una cantidad entera de veces.



Operadores de asignación:

= asignación		
= asignación de string	a = "Hola"	
= asignación de float	a = 3.14159	
= asignación de int	a = 8	
+ Suma	a += b	a = a + b
- Resta	a -= b	a = a - b
/ División regular	a /= b	a = a / b
// División entera	a //= b	a = a // b
% Módulo	a &= b	a = a % b
* Multiplicación	a *= b	a = a * b
** Exponenciación	a **= b	a = a ** b

asignación de un dato a un objeto Python - variables

# Sin operador Walrus	# Con operador Walrus
<pre>numero = 5 cuadrado = numero * numero if cuadrado > 10: print(cuadrado)</pre>	<pre>numero = 5 if (cuadrado := numero * numero) > 10: print(cuadrado)</pre>

En este ejemplo, la expresión (cuadrado := numero * numero) asigna el cuadrado de numero a la variable cuadrado y, al mismo tiempo, evalúa si el cuadrado es mayor que 10

Operador Walrus:

El "operador Walrus" en Python, u "operador de asignación de expresiones" u "operador de asignación de expresiones de fusión". Este operador se introdujo en la versión 3.8 de Python y se representa con:=.

Este operador permite asignar un valor a una variable como parte de una expresión. Es útil cuando deseas asignar un valor y utilizarlo en la misma expresión. Un caso de uso común es en bucles while o if para evitar duplicar cálculos.

Este operador es especialmente útil cuando se trabaja con bucles, ya que permite realizar asignaciones y comprobaciones en una sola línea de código. Ten en cuenta que el uso excesivo del operador Walrus puede hacer que el código sea menos legible, así que úsalo con moderación y en situaciones donde mejore la claridad del código.



Operadores de comparación:

== condición Igual	a == b
!= condición Diferente	a != b
> mayor que	a > b
>= mayor igual que	a >= b
< menor que	a < b
<= menor igual que	a <= b
comparación entre objetos de un dato - variables	

Ejemplos de métodos a los que llaman los operadores

- El operador > llama al método `_gt_` (greater than).
- El operador >= llama al método `_ge_` (greater than or equal).
- El operador == llama al método `_eq_` (equal).
- El operador < llama al método `_lt_` (less than).
- El operador <= llama al método `_le_` (less than or equal).
- El operador != llama al método `_ne_` (not equal).

Operadores lógicos o booleanos:

and	True and True	True
	True and False	False
	False and True	False
	False and False	False
True si todos los segmentos de la condición son True False si al menos un segmento es False.		

or	True or True	True
	True or False	True
	False or True	True
	False or False	False
True al menos un segmento de la condición es True False si todos los segmentos son False.		

not	not True	False
	not False	True
Not niegan la condición		



Operadores de identidad:

is True	is	True si ambos operandos hacen referencia al mismo objeto. False en caso contrario.
is False	is not	True si ambos operandos NO hacen referencia al mismo objeto. False en caso contrario.
is None		

Operadores de membresía o pertenencia:

En iterables	in	True si el valor del objeto está presente en la conjunto, colección o string. False en caso contrario.
	is in	True si el valor del objeto NO está presente en la conjunto, colección o string. False en caso contrario.

Operador de membresía o pertenencia

str.__contains__(objeto)
list.__contains__(objeto)
tuple.__contains__(objeto)
dict.__contains__(elemento)

Es importante destacar que estos métodos especiales deben estar definidos en las clases de los objetos sobre los cuales se aplican estos operadores para que funcionen correctamente.
Si importo pandas y genero un DataFrame, este tendrá el método __contains__

Operadores de asignación a nivel de bits:

Los operadores a nivel de bits actúan sobre los operandos como si fueran una cadena de dígitos binarios.
Como su nombre indica, actúan sobre los operandos bit a bit.

Nivel byte	
x y	or bit a bit de x e y.
x ^ y	or exclusivo bit a bit de x e y.
x & y	and bit a bit de x e y.
x << y	Desplaza x n bits a la izquierda.
x >> y	Desplaza x n bits a la derecha.
~x	Obtiene los bits de x invertidos.

Nivel byte

&=	a &= b	a = a & b
=	a = b	a = a b
^=	a ^= b	a = a ^ b
>>=	a >>= b	a = a >> b
<<=	a <<= b	a = << b



Módulo 8 - Objetos - Funciones varias built-in (integradas):

Objetos - Funciones varias built-in (integradas):

En Python todo son objetos. Las funciones también

Veremos varias funciones built-in (incorporadas) en Python

En este módulo partimos de que ya has usado:

funciones built-in: [print, if, else, elif, input, while, for, list, tuple, dict, bool, set, frozenset, type, dir]

operadores: [+,-,/,,%,*,**],[<,<=,>,>=,==,!=],[=,+,-,=/,*=,...,:=], [not, or, and], [in, not in], [is, not is]

Hay palabras reservadas (fuertes) que no deben ser usadas como identificadores o nombres de variables y otras (débiles) que no se aconseja su uso. (<https://docs.python.org/3/library/functions.html>)

<https://docs.python.org/es/3.12/library/functions.html>

Reservadas fuertes:

and, del, from, **not**, **while**, as, **elif**, global, **or**, with, assert, else, if, pass, yield, break, except, import, print, class, exec, in, raise, continue, finally, is, return, def, for, lambda, try

Reservadas débiles:

all, any, input, elif, int, float, str, list, tuple, dict, set, frozenset, except, open, close, read, write, nombres de librerías / bibliotecas que estés usando, etc.

A	E	L	R
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
any()			round()
anext()			
ascii()			
B	F	M	S
bin()	filter()	map()	set()
bool()	float()	max()	setattr()
breakpoint()	format()	memoryview()	slice()
bytarray()	frozenset()	min()	sorted()
bytes()			staticmethod()
C	G	N	str()
callable()	getattr()	next()	sum()
chr()	globals()		super()
classmethod()			
compile()			
complex()			
D	H	O	T
delattr()	hasattr()	object()	tuple()
dict()	hash()	oct()	type()
dir()	help()	open()	
divmod()	hex()	ord()	
I	P	V	Z
	id()	pow()	vars()
	input()	print()	zip()
	int()	property()	
	instance()		
	issubclass()		
	iter()		
			-
			import _()



abs()

Devuelve el valor absoluto de un número. El argumento puede ser un número entero, un número de punto flotante o un objeto que implemente `__abs__()`. Si el argumento es un número complejo, se devuelve su magnitud.

```
abs(objeto numérico)
print (f"abs(-9)={} <---> {abs(9)=}"')
#Salida esperada abs(-9)=9 <---> abs(9)=9
```

all(iterable)

Retorna True si todos los elementos del iterable son verdaderos (o si el iterable está vacío)

```
dato_1=dato_2=dato_3=dato_4= True
if all([dato_1,dato_2,dato_3,dato_4]):
    print ("Todos los datos son True--->",dato_1,dato_2,dato_3,dato_4)
else:
    print ("algún datos no es True --->",dato_1,dato_2,dato_3,dato_4)
```

#Salida esperada Todos los datos son True---> True True True True

```
dato_1=dato_3=dato_4= True
dato_2= False
if all([dato_1,dato_2,dato_3,dato_4]):
    print ("Todos los datos son True--->",dato_1,dato_2,dato_3,dato_4)
else:
    print ("algún datos no es True --->",dato_1,dato_2,dato_3,dato_4)
```

#Salida esperada algún datos no es True ---> True False True True

Devuelve verdadero si todos los elementos o condiciones son verdaderos es verdadero.

- Devuelve False si algún elemento o condición es falsos.
- Cualquiera puede considerarse como una secuencia de operaciones OR en los iterables proporcionados.
- `print (all([True, False, False, False]))` Falso
- `print (all([True, True, True, True]))` Verdadero

```
a=1;b=2;c=3;d=4;e=5;f=0
if all([(a>b),( b>c),( c>d ),(d>e)]):
    print("Todos son True")
    print([(a>b),( b>c),( c>d ),(d>e)])
else:
    print ("alguno no es True")
    print([(a>b),( b>c),( c>d ),( d<e )])
```

any(iterable)

Retorna True si un elemento cualquiera del iterable es verdadero. Si el iterable está vacío, retorna False.

```
dato_1=dato_2=dato_3=dato_4= False
if any([dato_1,dato_2,dato_3,dato_4]):
```



```
print ("algún datos es True --->",dato_1,dato_2,dato_3,dato_4)
else:
```

```
    print ("ningún dato es True--->",dato_1,dato_2,dato_3,dato_4)
```

```
#Salida esperada ningún dato es True---> False False False
```

```
dato_1=dato_3=dato_4= False
```

```
dato_2= True
```

```
if any([dato_1,dato_2,dato_3,dato_4]):
```

```
    print ("algún datos es True --->",dato_1,dato_2,dato_3,dato_4)
```

```
else:
```

```
    print ("ningún dato es True--->",dato_1,dato_2,dato_3,dato_4)
```

```
#Salida esperada algún datos es True ---> False True False False
```

Devuelve verdadero si alguno de los elementos o condiciones es verdadero.

- Devuelve False si está vacío o si todos son falsos.
- Cualquiera puede considerarse como una secuencia de operaciones OR en los iterables proporcionados.
- print (any([False, False, False, False])) Falso
- print (any([False, True, False, False])) verdadero

```
a=1;b=2;c=3;d=4;e=5;f=0
```

```
if any([(a>b),( b<c),( c>d ),(d<e)]):
```

```
    print("Alguno es True")
```

```
    print([(a>b),( b<c),( c>d ),(d<e)])
```

```
else:
```

```
    print ("ninguno True")
```

```
print([(a>b),( b<c),( c>d ),( d<e )])
```

ascii(objecto)

Retorna una cadena que contiene una representación imprimible de un objeto. Esto genera una cadena

```
print (ascii("¡¡¡¡Python es genial!!!!! ¿?"))
```

```
#Salida esperada '\xa1\xab\xab\xab\xabPython es genial!!!! \xbf?'
```

```
#          \\\no ascii\\//                                \\_no ascii\\//
```

bin(objecto entero)

Convierte un número entero a una cadena binaria con prefijo «0b».

```
print(f"el binario de 3 es {bin(3)}")
```

```
#Salida esperada 'el binario de 3 es 0b11'
```

```
print(f"el binario de -10 es {bin(-10)}")
```

```
#Salida esperada 'el binario de -10 es -0b1010'
```

bytearray(source=b'')

```
bytearray(source=b'')
```

```
bytearray(source, encoding)
```



`bytearray(source, encoding, errors)`

Retorna un nuevo array de bytes. La clase `bytearray` es una secuencia mutable de enteros en el rango $0 \leq x < 256$. Tiene la mayoría de los métodos comunes en las secuencias mutables, descritos en Tipos de secuencia mutables, así como la mayoría de los métodos que la clase `bytes` tiene, véase Operaciones de bytes y `bytearray`.)

El parámetro opcional `source` puede ser empleado para inicializar el vector (array) de varias maneras distintas:

- Si es una string, debes proporcionar también el parámetro `encoding` (y opcionalmente, `errors`; entonces `bytearray()` convierte la cadena a bytes empleando `str.encode()`).
- Si es un integer, el array tendrá ese tamaño y será inicializado con bytes nulos.
- Si es un objeto conforme a interfaz de búfer, se utilizará un búfer de solo lectura del objeto para inicializar el arreglo de bytes.
- Si es un iterable, debe ser un iterable de enteros en el rango $0 \leq x < 256$, que son usados como los contenidos iniciales del array.
- Sin argumento, se crea un array de tamaño 0.

```
# Crear un bytearray a partir de una cadena de bytes
string_org = b'hola, mundo IT'
byte_array = bytearray(string_org)
```

```
# Modificar el contenido del bytearray
byte_array[0] = 72 # ASCII para 'A'
byte_array.extend(b' 2024 !!!!')
print(f'{string_org}')
print(f'{byte_array}')
```

```
#salida esperada
#      hola, mundo IT
#      bytearray(b'Hola, mundo IT 2024 !!!')
```

`bytes(source=b'')`

```
class bytes(source=b'')
class bytes(source, encoding)
class bytes(source, encoding, errors)
```

Retorna un nuevo objeto «`bytes`», que es una secuencia inmutable de enteros en el rango $0 \leq x < 256$.

`bytes` es una versión inmutable de `bytearray` – tiene los mismos métodos no mutables y el mismo comportamiento en términos de indexación y segmentación.

En consecuencia, los argumentos del constructor son interpretados como los de `bytearray()`.

Los objetos de `bytes` también pueden ser creados con literales.

```
# Crear un objeto bytes directamente
byte_string = b'hola, mundo IT'
# Acceder a elementos individuales
primer_byte = byte_string[0] # Obtiene el primer byte
print(f'{primer_byte=}')

#salida esperada
#      primer_byte=72
#NOTA:(ASCII para 'H')
```



```
# Concatenar bytes
nuevo_byte_string = byte_string + b' 2024 !!!!'
print(f'{nuevo_byte_string}')
#salida esperada
#      b'Hola, mundo IT 2024 !!!'
```

chr(objeto int)

Retorna la cadena que representa un carácter cuyo código Unicode es el objeto entero. Por ejemplo, `chr(97)` retorna la cadena 'a', mientras que `chr(8364)` retorna la cadena '€'. Esta función es la inversa de `ord()`. El rango válido para el argumento `i` es desde 0 a 1,114,111 (0x10FFFF en base 16). Se lanzará una excepción `ValueError` si `i` está fuera de ese rango.

```
print (f"el carácter en la posición 97 es '{chr(97)}'")
#Salida esperada "el carácter en la posición 97 es 'a' "
print (f"el carácter en la posición 8364 es {chr(8364)}")
#Salida esperada "el carácter en la posición 8364 es '€' "
```

dir() dir(objecto)

Sin argumentos, retorna la lista de nombres en el ámbito local.

```
print (dir())
#Salida esperada ['__annotations__', '__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']

Con un argumento, intenta retornar una lista de atributos válidos para ese objeto.

print (dir(int()))#int() como ejemplo
#Salida esperada ['__abs__', ..., '__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'is_integer', 'numerator', 'real', 'to_bytes']
```

divmod(a, b)

Toma dos números (no complejos) como argumentos y retorna un par de números consistentes en su cociente y su resto al emplear división de enteros. Con operandos de tipos diferentes, se aplican las reglas de los operadores aritméticos binarios. Para enteros, el resultado es el mismo que $(a // b, a \% b)$. Para números de punto flotante el resultado es $(q, a \% b)$, donde q normalmente es $\text{math.floor}(a / b)$ pero puede ser 1 menos que eso. En cualquier caso $q * b + a \% b$ es muy cercano a a , si $a \% b$ es distinto de cero y tiene el mismo signo que b , y $0 \leq a \% b < \text{abs}(b)$.

```
print (f"divmod(9, 3) es '{divmod(9, 3)}'")
#Salida esperada divmod(9, 3) es '(3, 0)'
print (f"divmod(10, 3) es '{divmod(10, 3)}'")
#Salida esperada divmod(10, 3) es '(3, 1)'
```



enumerate(iterable, start=0)

Retorna un objeto enumerador. iterable tiene que ser una secuencia, un iterator, o algún otro objeto que soporte la iteración. Retorna una tupla que contiene un contador (desde start, cuyo valor por defecto es 0) y los valores obtenidos al iterar sobre iterable.

```
estaciones = ['Primavera', 'Verano', 'Otoño', 'Invierno']
```

```
print(f"list(enumerate(estaciones))={}")
```

```
#Salida esperada list(enumerate(estaciones))=[(0, 'Primavera'), (1, 'Verano'), (2, 'Otoño'), (3, 'Invierno')]
```

```
print(f"list(enumerate(estaciones, start=1))={}")
```

```
#Salida esperada list(enumerate(estaciones, start=1))=[(1, 'Primavera'), (2, 'Verano'), (3, 'Otoño'), (4, 'Invierno')]
```

.



eval(expression, globals=None, locals=None)

Esta función también puede ser utilizada para ejecutar objetos de código arbitrario (como los que crea la función `compile()`). En este caso, se pasa un objeto de código en vez de una cadena de caracteres. Si el objeto de código ha sido compilado usando 'exec' como el argumento mode, el valor que retornará `eval()` será `None`.

Los argumentos son una cadena y opcionalmente, globales y locales. Si se introduce, `globals` tiene que ser un diccionario, y `locals` puede ser cualquier objeto de mapeo.

El argumento `expression` se analiza y evalúa como una expresión de Python (técticamente hablando, una lista de condiciones), usando los diccionarios `globals` y `locals` como espacios de nombres globales y locales. Si el diccionario `globals` está presente y no contiene un valor para la clave `__builtins__`, se inserta bajo esa clave una referencia al diccionario del módulo incorporado `builtins` antes de que la `expression` sea analizada. De esa forma se puede controlar qué módulos incorporados están disponibles para el código ejecutado insertando un diccionario propio `__builtins__` dentro de `globals` antes de pasarlo a la `eval()`. Si el diccionario `locals` es omitido entonces su valor por defecto es el diccionario `globals`. Si ambos diccionarios son omitidos, la expresión es ejecutada con las `globals` y `locals` del entorno en el que `eval()` es llamada. Tener en cuenta que `eval()` no tiene acceso al nested scopes (no locales) en el entorno que lo contiene.

El valor que retorna es el resultado de la expresión evaluada. Los errores de sintaxis son reportados como excepciones.

```
x = 12
print(f"eval(f{x}+10**2+8/4)=\")")
#Salida esperada eval(f{x}+10**2+8/4)=114.0
```

format(value, format_spec='')

Convierte `value` a su representación «con formato», de forma controlada por `format_spec`. La interpretación de `format_spec` dependerá del tipo del argumento `value`. Sin embargo, hay una sintaxis estándar de formato que emplean la mayoría de los tipos incorporados: Especificación de formato Mini-Lenguaje.

El `format_spec` por defecto es una cadena vacía que normalmente produce el mismo efecto que llamar a `str(value)`.

```
# Definir variables
nombre = "Ariel"
edad = 50
ciudad = "Caba"
# Crear una plantilla de cadena con llaves placeholders {}
plantilla = "Hola, mi nombre es {} y tengo {} años. Vivo en {}."
mensaje_formateado='Hola, mi nombre es Ariel y tengo 50 años. Vivo en Caba.'
# Usar format para reemplazar los placeholders con los valores de las variables
mensaje_formateado = plantilla.format(nombre, edad, ciudad)
print(f"mensaje_formateado=\")")
#Salida esperada eval(f{x}+10**2+8/4)=114.0
```



```
mensaje = "Hola {1}, mi nombre es {0}."  
mensaje_formateado = mensaje.format("Ariel", "Juan")  
print(f'{mensaje_formateado}')
```

#Salida esperada mensaje_formateado='Hola Juan, mi nombre es Ariel.'

```
mensaje = "Hola, mi nombre es {nombre} y tengo {edad} años."  
mensaje_formateado = mensaje.format(nombre="Ariel", edad=50)  
print(f'{mensaje_formateado}')
```

#Salida esperada mensaje_formateado='Hola, mi nombre es Ariel y tengo 50 años.'

```
precio = 19.999999999  
mensaje = "El precio es: ${:.2f}"  
mensaje_formateado = mensaje.format(precio)  
print(f'{mensaje_formateado}')
```

#Salida esperada mensaje_formateado='El precio es: \$19.99'

```
mensaje_formateado = "Hola soy, {:>50}!".format(nombre)  
print(f'{mensaje_formateado}')
```

#Salida esperada mensaje_formateado='Hola soy, Ariel!'

```
mensaje_formateado = "Hola soy, {:^50}!".format(nombre)  
print(f'{mensaje_formateado}')
```

#Salida esperada mensaje_formateado='Hola soy, Ariel !'

```
mensaje_formateado = "Hola soy, {:<50}!".format(nombre)  
print(f'{mensaje_formateado}')
```

#Salida esperada mensaje_formateado='Hola soy, Ariel !'

hex(x)

Convierte un número entero a una cadena hexadecimal de minúsculas con el prefijo «0x». Si x no es un objeto de la clase Python int, tiene que definir un método `__index__()` que retorne un entero.

```
entero = 123654  
hexadecimal = hex(entero)  
print(f'{entero} {hexadecimal}')  
#Salida esperada entero=123654 hexadecimal='0x1e306'
```

isinstance(object, classinfo)

Retorna True si el argumento object es una instancia del argumento classinfo, o de una subclase (directa, indirecta o virtual) del mismo. Si object no es un objeto del tipo indicado, esta función siempre retorna False. Si classinfo es una tupla de objetos de tipo (o recursivamente, otras tuplas lo son) o Tipo de conversión de múltiples tipos, retorna True si object es una instancia de alguno de los tipos. Si classinfo no es un tipo, una tupla de tipos, o una tupla de tuplas de tipos, una excepción TypeError es lanzada. Es posible que TypeError no se genere para un tipo no válido si una verificación anterior tiene éxito.

```
Objeto = 123#-----modificar a "hola" 3.14159, True  
print (f'{isinstance(objeto,str)} -- {isinstance(objeto,float)} -- {isinstance(objeto,int)} --
```



```
{isinstance(objeto,bool)=}"}
print (f"{isinstance(objeto,(float,int))=}"--> Nota: más de una opción como tupla ")
```

#Salida esperada

isinstance(objeto,str)=False -- isinstance(objeto,float)=False -- isinstance(objeto,int)=True --

isinstance(objeto,bool)=False

isinstance(objeto,(float,int))=True--> Nota: más de una opción como tupla

len(s)

Retorna el tamaño (el número de elementos) de un objeto. El argumento puede ser una secuencia (como una cadena, un objeto byte, una tupla, lista o rango) o una colección (como un diccionario, un set o un frozen set).

```
lista = [9,1,7,3,5]
string = "Hola Mundo IT"
print (f"{lista}      {len(lista)=}"")
print (f"{string}      {len(string)=}"")
```

#Salida esperada

[9, 1, 7, 3, 5] len(lista)=5
Hola Mundo IT len(string)=13

max(iterable, *, key=None)

Retorna el elemento mayor en un iterable o el mayor de dos o más argumentos.

Si un argumento posicional es dado, debe ser un iterable. El elemento mayor en el iterable es retornado. Si dos o más argumentos posicionales son indicados, el mayor de los argumentos posicionales será retornado.

Hay dos argumentos de solo palabra clave que son opcionales. El argumento key especifica una función de ordenación de un sólo argumento, como la usada para list.sort(). El argumento default especifica un objeto a retornar si el iterable proporcionado está vacío. Si el iterable está vacío y default no ha sido indicado, se lanza un ValueError.

```
print("max(iterable, *, default, key=None)")
print("max(arg1, arg2, *args, key=None)")
```

```
lista_int = [9,1,7,3,5]
lista_str = ['Primavera', 'Verano', 'Otoño', 'Invierno']
print (f"{lista_int}      {max(lista_int)=}"")
print (f"{lista_str}      {max(lista_str)=}"")
print (f"{lista_str}      {max(lista_str, key=len)=}"")
```

#Salida esperada

[9, 1, 7, 3, 5] max(lista_int)=9
['Primavera', 'Verano', 'Otoño', 'Invierno'] max(lista_str)=Verano'
['Primavera', 'Verano', 'Otoño', 'Invierno'] max(lista_str, key=len)=Primavera'

```
dic = {"C":1,"A":7,"B":3,"D":2 }
```

```
print (f"{dic}      {max(dic)=}"")
```

```
print (f"{dic}      clave del valor máximo {max(dic, key=dic.get)=}      valor máximo
{dic[max(dic, key=dic.get)]=}"")
```



```
#Salida esperada
```

```
{'C': 1, 'A': 7, 'B': 3, 'D': 2}      min(dic)='D'
```

```
{'C': 1, 'A': 7, 'B': 3, 'D': 2}      clave del valor máximo min(dic, key=dic.get)='A'    valor máximo
```

```
dic[min(dic, key=dic.get)]=7
```

min(iterable, *, key=None)

Retorna el menor elemento en un iterable o el menor de dos o más argumentos.

Si se le indica un argumento posicional, debe ser un iterable. El menor elemento del iterable es returned. Si dos o más argumentos posicionales son indicados, el menor de los argumentos posicionales es returned.

Hay dos argumentos de solo palabra clave que son opcionales. El argumento key especifica una función de ordenación de un sólo argumento, como la usada para list.sort(). El argumento default especifica un objeto a retornar si el iterable proporcionado está vacío. Si el iterable está vacío y default no ha sido indicado, se lanza un ValueError.

Si hay múltiples elementos con el valor mínimo, la función retorna el primero que encuentra. Esto es consistente con otras herramientas que preservan la estabilidad de la ordenación como sorted(iterable, key=keyfunc)[0] y heapq.nsmallest(1, iterable, key=keyfunc).

```
min(iterable, *, default, key=None)
```

```
min(arg1, arg2, *args, key=None)
```

```
lista_int = [9,1,7,3,5]
```

```
lista_str = ['Primavera', 'Verano', 'Otoño', 'Invierno']
```

```
print(f" {lista_int}      {min(lista_int)}")
```

```
print(f" {lista_str}      {min(lista_str)}")
```

```
print(f" {lista_str}      {min(lista_str, key=len)}")
```

```
#Salida esperada
```

```
# [9, 1, 7, 3, 5]      min(lista_int)=1
```

```
# ['Primavera', 'Verano', 'Otoño', 'Invierno']      min(lista_str)='Invierno'
```

```
# ['Primavera', 'Verano', 'Otoño', 'Invierno']      min(lista_str, key=len)='Otoño'
```

```
dic = {"C":1,"A":7,"B":3,"D":2}
```

```
print(f" {dic}      {min(dic)}")
```

```
print(f" {dic}      clave del valor mínimo {min(dic, key=dic.get)}      valor mínimo
```

```
{dic[min(dic, key=dic.get)]}")
```

```
#Salida esperada
```

```
# {'C': 1, 'A': 7, 'B': 3, 'D': 2}      min(dic)='A'
```

```
# {'C': 1, 'A': 7, 'B': 3, 'D': 2}      clave del valor mínimo min(dic, key=dic.get)='C'      valor
```

```
mínimo dic[min(dic, key=dic.get)]=1
```

oct(x)

Convierte un número entero a una cadena octal con prefijo «0o». El resultado es una expresión válida de Python. Si x no es un objeto de la clase Python int, tiene que definir un método __index__() que retorne un entero.

```
entero = 8
```

```
octal = oct(entero)
```

```
print(f" {entero}      {octal}")
```



```
#Salida esperada entero=8  octal='0o10'
```

```
entero = 1
octal = oct(entero)
print(f"entero={}  {octal=}")
```

```
#Salida esperada entero=1  octal='0o1'
```

```
entero = 123654
octal = oct(entero)
print(f"entero={}  {octal=}")
```

```
#Salida esperada entero=123654  octal='0o361406'
```

ord(c)

Al proporcionarle una cadena representando un carácter Unicode, retorna un entero que representa el código Unicode de ese carácter. Por ejemplo, `ord('a')` retorna el entero 97 y `ord('€')` (símbolo del Euro) retorna 8364. Esta es la función inversa de `chr()`.

```
string = "Ñ"
ordinal = ord(string)
print(f"string={}  {ordinal=}")
```

```
#Salida esperada string='Ñ'  ordinal=209
```

```
string = "A"
ordinal = ord(string)
print(f"string={}  {ordinal=}")
```

```
#Salida esperada string='A'  ordinal=65
```

```
string = "😊"
ordinal = ord(string)
print(f"string={}  {ordinal=}")
```

```
#Salida esperada string='😊'  ordinal=128522
```

pow(base, exp, mod=None)

Retorna `base` elevado a `exp`; si `mod` está presente, retorna `base` elevado a `exp`, módulo `mod` (calculado de manera más eficiente que `pow(base, exp) % mod`). La forma con dos argumentos `pow(base, exp)` es equivalente a usar el operador potencia: `base**exp`.

Los argumentos deben ser de tipo numérico. Si hay tipos mixtos de operandos, aplican las reglas de coerción para operadores binarios aritméticos. Para operandos de la clase `int`, el resultado tiene el mismo tipo que los operandos (después de la coerción) a menos que el segundo argumento sea negativo; en tal caso, todos los argumentos son convertidos a punto flotante y un resultado de punto flotante es retornado. Por ejemplo, `pow(10, 2)` retorna 100, pero `pow(10, -2)` retorna 0.01. Para una base negativa de tipo `int` o `float` y un exponente no integral, se obtiene un resultado complejo. Por ejemplo, `pow(-9, 0.5)` retorna un valor cercano a `3j`.

```
base= 3
exponente = 3
print(f"Salida {pow(base,exponente)=}")
```



Salida esperada Salida pow(base,exponente)=27

```
print (f"Salida {pow(2, 3, 5)}=} Calcula (2^3) % 5")
```

Salida esperada Salida esperada Salida pow(2, 3, 5)=3 Calcula (2^3) % 5

reversed(seq)

Devuelve un iterador inverso. seq debe ser un objeto que tenga un método `__reversed__()` o que admita el protocolo de secuencia (el método `__len__()` y el método `__getitem__(0)` con argumentos enteros que comienzan en 0).

```
lista_int = [9,1,7,3,5]
```

```
print (f"Salida {lista_int=}           {list(reversed(lista_int))=}") Nota, devuelve un objeto reversed  
que hay que convertir en lista para ver en print ")
```

Salida esperada

Salida lista_int=[9, 1, 7, 3, 5]

```
list(reversed(lista_int))=[5, 3, 7, 1, 9]
```

Nota, devuelve un objeto reversed que hay que convertir en lista para ver en print

```
lista_str = ['Primavera', 'Verano', 'Otoño', 'Invierno']
```

```
print (f"Salida {lista_str=}           {list(reversed(lista_str))=}") Nota, devuelve un objeto reversed  
que hay que convertir en lista para ver en print ")
```

Salida esperada

Salida lista_str=['Primavera', 'Verano', 'Otoño', 'Invierno']

```
list(reversed(lista_str))=['Invierno', 'Otoño', 'Verano', 'Primavera']
```

Nota, devuelve un objeto reversed que hay que convertir en lista para ver en print

round(number, ndigits=None)

Retorna number redondeado a ndigits de precisión después del punto decimal. Si ndigits es omitido o es None, retorna el entero más cercano a su entrada.

Para los tipos integrados que admiten `round()`, los valores se redondean al múltiplo más cercano de 10 a la potencia menos ndigits; si dos múltiplos están igualmente cerca, el redondeo se realiza hacia la opción par (así, por ejemplo, tanto `round(0.5)` como `round(-0.5)` son 0, y `round(1.5)` es 2). Cualquier valor entero es válido para ndigits (positivo, cero o negativo). El valor returned es un entero si se omite ndigits o None. De lo contrario, el valor returned tiene el mismo tipo que number.

Para un objeto number general de Python, `round` delega a `number.__round__`.

```
print (f"{{round (3.14159)=}      {round (3.14159,0)=}}")
```

```
print (f"{{round (3.14159,2)=}}")
```

```
print (f"{{round (3.14159,4)=}}")
```

round (3.14159)=3

round (3.14159,0)=3.0

round (3.14159,2)=3.14

round (3.14159,4)=3.1416

slice(inicio, parada, paso=Ninguno)



Devuelve un objeto de sector que representa el conjunto de índices especificados por rango (inicio, parada, paso). Los argumentos de inicio y paso están predeterminados en Ninguno.

Los objetos Slice tienen atributos de datos de solo lectura de inicio, parada y paso que simplemente devuelven los valores de los argumentos (o sus valores predeterminados). No tienen ninguna otra funcionalidad explícita; sin embargo, NumPy y otros paquetes de terceros los utilizan.

Los objetos de segmento también se generan cuando se utiliza la sintaxis de indexación extendida. Por ejemplo: `a[start:stop:step]` o `a[start:stop, i]`. Consulte `itertools.islice()` para obtener una versión alternativa que devuelve un iterador.

```
lista_int=[9,5,1,7,4,3,6,8,2,5,0,1,4,7,8,9,6,3,2,1,0,5]
inicio = 2
limite_final=14
step_paso = 2
print (f"Salida {lista_int}")
print (f"{lista_int[slice(inicio,limite_final,step_paso)]}")
print ("Nota, devuelve un objeto slice que hay que convertir en lista para ver en print ")
```

Salida esperada

```
Salida [9,5,1,7,4,3,6,8,2,5,0,1,4,7,8,9,6,3,2,1,0,5] =
lista_int[slice(inicio,limite_final,step_paso)] =
Nota, devuelve un objeto slice que hay que convertir en lista para ver en print
```

sorted(iterable, /, *, key=None, reverse=False)

Retorna una nueva lista ordenada a partir de los elementos en iterable.

Tiene dos argumentosopcionales que deben ser especificados como argumentos de palabra clave.

key especifica una función de un argumento que es empleada para extraer una clave de comparación de cada elemento en iterable (por ejemplo, `key=str.lower`). El valor por defecto es `None` (compara los elementos directamente).

`reverse` es un valor boleado. Si está puesto a `True`, entonces la lista de elementos se ordena como si cada comparación fuera reversa.

Puedes usar `functools.cmp_to_key()` para convertir las funciones `cmp` a la antigua usanza en funciones `key`.

La función built-in `sorted()` está garantizada en cuanto a su estabilidad. Un ordenamiento es estable si garantiza que no cambia el orden relativo de elementos que resultan igual en la comparación — esto es de gran ayuda para ordenar en múltiples pasos (por ejemplo, ordenar por departamento, después por el escalafón de salario).

El algoritmo de ordenación utiliza sólo comparaciones de `<` entre items. Mientras que al definir un método `__lt__()` será suficiente para ordenar, PEP 8 recomienda que las seis comparaciones rich comparisons se implementen. Esto ayudará a evitar errores al usar los mismos datos con otras herramientas de ordenado como `max()` que se basan en un método subyacente diferente. La implementación de las seis comparaciones también ayuda a evitar confusiones por comparaciones de tipos mixtos que pueden llamar reflejado al método `__gt__()`. Si hay múltiples elementos con el valor máximo, la función retorna el primero que ha encontrado. Esto es consistente con otras herramientas para preservar la estabilidad de la ordenación como `sorted(iterable, key=keyfunc, reverse=True)[0]` y `heapq.nlargest(1, iterable, key=keyfunc)`.

```
lista_int = [9,1,7,3,5]
print (f"Salida {lista_int}")
print (f"{list(sorted(lista_int))}")
print ("Nota, devuelve un objeto sorted que hay que convertir en lista para ver en print ")
```

#Salida esperada

```
# Salida lista_int=[9, 1, 7, 3, 5]
# list(sorted(lista_int))=[1, 3, 5, 7, 9]
```



```
# Nota, devuelve un objeto sorted que hay que convertir en lista para ver en print
```

```
lista_str = ['Primavera', 'Verano', 'Otoño', 'Invierno']
print(f"Salida {lista_str} ")
print(f"list(sorted(lista_str))={}")
print ("Nota, devuelve un objeto sorted que hay que convertir en lista para ver en print ")
```

#Salida esperada

```
# Salida lista_str=['Primavera', 'Verano', 'Otoño', 'Invierno']
# list(sorted([lista_str])=['Invierno', 'Otoño', 'Primavera', 'Verano']
# Nota, devuelve un objeto sorted que hay que convertir en lista para ver en print
```

sum(iterable, /, start=0)

Suma start y los elementos de un iterable de izquierda a derecha y Retorna el total. Los elementos del iterable son normalmente números, y el valor start no puede ser una cadena.

Para algunos casos de uso, hay buenas alternativas a sum(). La manera preferente y más rápida de concatenar secuencias de cadenas es llamado a `".join(sequence)`. Para añadir valores de punto flotante con precisión extendida, ver `math.fsum()`. Para concatenar series de iterables, considera usar `itertools.chain()`.

```
lista_int=[9, 5, 1, 7, 4, 3, 6, 8, 2, 5, 0, 1, 4, 7, 8, 9, 6, 3, 2, 1, 0, 5]
print(f"la suma de la lista es {sum(lista_int)}")
```

#Salida esperada

```
# la suma de la lista es 96
```

```
dic = {"C":1,"A":7,"B":3,"D":2 }
print (f"la suma del diccionario es {sum(dic.values())}")
```

#Salida esperada

```
# la suma del diccionario es 13
```

zip(*iterables, strict=False)

Iterar sobre varios iterables en paralelo, generando tuplas con un item para cada una.

Más formalmente: zip() retorna un iterador de tuplas, donde la tupla i-ésima contiene el elemento i-ésimo de cada uno de los argumentos iterables.

```
for item in zip([1, 2, 3 ,4], ['Primavera', 'Verano', 'Otoño', 'Invierno']):
    print(item)
```

Salida esperada

```
(1, 'Primavera')
(2, 'Verano')
(3, 'Otoño')
(4, 'Invierno')
```

Otra forma de pensar en zip() es que convierte las filas en columnas y las columnas en filas. Esto es similar a transponer una matriz.

zip() es perezoso: los elementos no se procesarán hasta que el iterable sea iterado, por ejemplo, mediante un bucle for o envolviendo en un list.

Una cosa a considerar es que los iterables pasados a zip() podrían tener diferentes longitudes; a veces por diseño, y a veces por un error en el código que preparó estos iterables. Python ofrece tres enfoques diferentes para tratar este problema:

De forma predeterminada, zip() se detiene cuando se agota el iterable más corto. Ignorará los elementos



restantes en las iteraciones más largas, cortando el resultado a la longitud del iterable más corto:

```
print(f"{{list(zip(range(1,5), ['Primavera', 'Verano', 'Otoño', 'Invierno']))}}")
```

#Salida esperada

```
# list(zip(range(1,5), ['Primavera', 'Verano', 'Otoño', 'Invierno']))=[(1, 'Primavera'), (2, 'Verano'), (3, 'Otoño'), (4, 'Invierno')]
```

`zip()` se usa a menudo en los casos en que se supone que los iterables son de igual longitud. En tales casos, se recomienda usar la opción `strict=True`. Su salida es la misma que la normal `zip()`:

```
print("list(zip('a', 'b', 'c'), (1, 2, 3), strict=True))")
```

```
print(f"{{list(zip(range(1,5), ['Primavera', 'Verano', 'Otoño', 'Invierno']), strict=True)}}")
```

#Salida esperada

```
# list(zip(range(1,5), ['Primavera', 'Verano', 'Otoño', 'Invierno'], strict=True))=[(1, 'Primavera'), (2, 'Verano'), (3, 'Otoño'), (4, 'Invierno')]
```

try:

```
    print("A diferencia del comportamiento predeterminado, lanza una excepción ValueError si un iterable se agota antes que los demás:")
```

```
    print(f"{{list(zip(range(1,50), ['Primavera', 'Verano', 'Otoño', 'Invierno']), strict=True)}}")
```

except Exception as error:

```
    print(f"{{error=}}")
```

#Salida esperada

```
# ValueError: zip() argument 2 is longer than argument 1")
```

Sin el argumento `strict=True`, cualquier error que resulte en iterables de diferentes longitudes será silenciado, posiblemente manifestándose como un error difícil de encontrar en otra parte del programa.

Las iteraciones más cortas se pueden acolchar con un valor constante para que todas las iteraciones tengan la misma longitud. Esto lo hace `itertools.zip_longest()`.

Casos extremos: con un único argumento iterable, `zip()` retorna un iterador de 1-tuplas. Sin argumentos, retorna un iterador vacío.

Consejos y trucos:

La evaluación de izquierda a derecha de los iterables está garantizada. Esto permite emplear una expresión idiomática para agregar una serie de datos en grupos de tamaño n usando `zip(*[iter(s)]*n, strict=True)`. Esto repite el mismo iterador n veces de forma que cada tupla de salida tiene el resultado de n llamadas al iterador. Esto tiene el efecto de dividir la entrada en trozos de longitud n.

`zip()` en conjunción con el operador * puede usar para descomprimir (unzip) una lista:

```
x = [1, 2, 3]
```

```
y = [4, 5, 6]
```

```
print(f"{{list(zip(x, y))}}")
```

#salida esperada list(zip(x, y))=[(1, 4), (2, 5), (3, 6)]

```
x2, y2 = zip(*zip(x, y))
```

```
print(f"{{x == list(x2) and y == list(y2)}}")
```

#salida esperada x == list(x2) and y == list(y2)=True



Módulo 9 –funciones:

¿Qué es una función?

En Python, una función es un objeto.

- Python tiene un paquete de funciones pre-construidas '**builtin_function_or_method**'.

`print ()` tiene por trabajo enviar a consola el argumento o parámetro que se le envían.

`print ('Salida')` #envía el argumento '**Salida**' a consola.

`dato = 'por parametro'`

`print (dato)` o `print (f'{dato}')` #envía mediante el parámetro dato '**por parámetro**' a consola.

`valor = int(cadena)` #mediante la función `int` asigna el contenido de cadena a valor (si es posible) como entero

A abs() aiter() all() anext() any() ascii()	D delattr() dict() dir() divmod()	H hasattr() hash() help() hex()	N next()	S set() setattr() slice() sorted() staticmethod() str() sum() super()
B bin() bool() breakpoint() bytearray() bytes()	E enumerate() eval() exec()	I id() input() int() isinstance() issubclass() iter()	O object() oct() open() ord()	P pow() print() property()
C callable() chr() classmethod() compile() complex()	F format() frozenset()	G len() list() locals()	L range() repr() reversed() round()	R vars()
			M map() max() memoryview() min()	Z zip()
				– import ()

- **funciones propias (def return)**

- `def return`
- Argumentos posicionales y por omisión
 - Los argumentos posicionales se asignan por posición.
 - Los argumentos por omisión tienen valores predefinidos.
- Parámetros arbitrarios en Python
 - Los argumentos posicionales se asignan por posición.
 - Los argumentos por omisión tienen valores predefinidos.
- funciones con parámetros y operadores
- Orden de los argumentos
 - posicionales
 - `*args` para recibir un número variable de argumentos posicionales.
 - `**kwargs` para recibir un número variable de argumentos clave-valor.
 - por omisión
- Recursividad
 - Una función puede llamarse a sí misma.



- Anotaciones en funciones.
 - Puedes agregar anotaciones a los parámetros y al valor de retorno.
- Docstrings
 - """ Descripción de la función para documentación. """
- Retorno Empaquetados y desempaquetados.
- Ámbitos de los distintos objetos.
- Anidación de funciones
 - Puedes definir funciones dentro de otras funciones.
- Llamado a funciones desde/como un string.

función es un objeto con nombre propio con un bloque de código reutilizable.

Este bloque se utiliza para realizar una tarea específica y permiten dividir el código en fragmentos más pequeños y lógicos,

Estos bloques o funciones permiten la organización, la reutilización y el mantenimiento del código.

Se puede asignar a un encargado (persona grupo) a cada función.

Para crear una función se utiliza la palabra clave **def**, seguida del nombre de la función (en snake case, sin usar palabras claves de Python),

Luego del nombre paréntesis que pueden contener (o no) los objetos parámetros para recibir datos desde afuera de la función a dentro de esta.

Cuando se llama a una función, se utiliza el nombre de la función seguido de paréntesis. Los argumentos pasados en la llamada a la función se asignan a los parámetros definidos en la declaración de la función.

En Python no es totalmente cierto que se debe proveer el mismo número de argumentos como haya parámetros definidos. Veremos más adelante parámetros con argumentos por omisión (donde algunos parámetros no se envían a la función, se omiten)

Se utiliza la función termina por indentación o por encontrar **return** y vuelve al lugar que se la llamo.

Características de las funciones teóricos
Las funciones en Python son bloques de código reutilizables que realizan tareas específicas. Tienen varias características que las hacen poderosas y flexibles en la programación. A continuación, se presentan algunas de las características clave de las funciones en Python:
Modularidad: Las funciones permiten dividir el código en módulos más pequeños y manejables. Esto mejora la legibilidad, el mantenimiento y la reutilización del código, ya que una función puede ser llamada en múltiples lugares dentro de un programa.
Reutilización de código: Las funciones permiten escribir una vez y utilizar muchas veces. Puedes definir una función y llamarla en diferentes partes de tu programa sin tener que volver a escribir el mismo código una y otra vez.
Encapsulación: Las funciones encapsulan la lógica y las operaciones en un solo lugar. Puedes agrupar un conjunto de instrucciones relacionadas en una función, lo que facilita el seguimiento y la comprensión del flujo del programa.
Parámetros: Las funciones pueden aceptar parámetros, que son valores que se pasan a la función para que los utilice en sus operaciones. Los parámetros permiten que una función sea más genérica y flexible, ya que pueden tomar diferentes valores en cada llamada.
Retorno de valores: Las funciones pueden devolver un valor o resultado después de realizar sus operaciones. Esto permite que el resultado de una función se utilice en otras partes del programa o se asigne a una variable para su posterior procesamiento.
Ámbito (scope) de las variables: Las variables definidas dentro de una función tienen un ámbito local, lo que significa que solo son accesibles dentro de la función en la que se definen. Esto evita conflictos de nombres con variables fuera de la función y ayuda a mantener la claridad y la integridad del código.
funciones como objetos de primera clase: En Python, las funciones son objetos de primera clase, lo que



significa que se pueden asignar a variables, pasar como argumentos a otras funciones y devolver como resultados de otras funciones. Esto permite un alto grado de flexibilidad en la programación y la implementación de conceptos como funciones de orden superior y programación funcional.

Estas son algunas de las características más importantes de las funciones en Python. Con el uso adecuado de las funciones, puedes modularizar tu código, mejorar su mantenibilidad y reutilización, y escribir programas más estructurados y legibles.

Por flujo de información las funciones deben estar antes de ser llamadas.

Son objetos y deben cumplir con las normas de nombres de Python interprete y comunidad.

Verificar el ámbito de:

Variables para lectura y escritura en la función

Colecciones para lectura y escritura en la función

Empaquetado de entrada y salida

Es un bloque de código reutilizable que realiza una tarea específica que tienen una etiqueta para poder acceder a esta (llamarla).

Puedes definir funciones para organizar tu código en unidades más pequeñas y lógicas.

Las funciones son bloques de código reutilizables que realizan una tarea específica.

```
def nombre_de_la_función (parametros_de_entrada):
    # Cuerpo de la función
    # Realizar ingresos, análisis, operaciones, cálculos, etc.
    return parametros_de_salida #opcional, por default devuelve None
```

Definición de la función (def): Comienza con la palabra clave def, seguida del nombre de la función y paréntesis que pueden contener los parámetros de la función.

Parámetros de entrada: Son valores que puedes pasar a la función para que los utilice durante su ejecución. Los parámetros son opcionales y se especifican entre los paréntesis de la definición de la función. Python acepta variables de un solo dato, listas, tuplas (*), diccionarios (**), valores por omisión, y opciones que desarrollaremos más adelante.

Cuerpo de la función: Es el bloque de código indentado que se ejecutará cuando llames a la función. Aquí es donde se realiza el trabajo principal de la función.

Return: La palabra clave return se utiliza para devolver un resultado desde la función. Una función puede devolver un parámetro de salida. Si no se declara con return o si se declara sin un parámetro o argumento, la función devolverá None de manera implícita. En caso contrario, devolverá el valor del parámetro o el argumento.

Parámetros de salida: Son valores que puedes devolver desde la función como resultado de su ejecución. El parámetro de salida por defecto es None, u opcionalmente se especifica un objeto para transferir información al lugar donde se llamó a la función. Un objeto puede ser una variable de un solo dato o una lista, tupla, diccionario, etc.

```
def saludar(nombre):
    mensaje = f'Hola, {nombre}!'
    print(mensaje)
saludar("Pepe")
```

```
#Salida esperada por consola
Hola, Pepe!
```



El argumento “Pepe” se envía a la función `saludar` donde ingresa mediante el objeto nombre se concatena en mensaje y lo imprime.

El orden de los argumentos posicionales es importante.

Además de incluir el número correcto de argumentos, es importante notar el orden en el cual se indican los argumentos.

Los argumentos necesitan ser escritos en el orden exacto, en el que se han declarado los parámetros en la definición de la función.

Los parámetros sólo existen dentro de las funciones en donde han sido definidos, y el único lugar donde un parámetro puede ser definido es entre los paréntesis después del nombre de la función, donde se encuentra la palabra reservada `def`.

La asignación de un valor a un parámetro de una función se hace en el momento en que la función es invocada, especificando el argumento

```
def sumar(dato_1,dato_2):
    salida= dato_1+dato_2
    return salida
-----
dato_primer=9
dato_segundo=2
entrada= sumar(dato_primer,dato_segundo)
print (f"sumar {dato_primer} + {dato_segundo} = {entrada}")
```

#Salida esperada por consola
sumar 9 + 2 = 11

En este caso, la función `sumar()` toma dos parámetros `dato_1` Y `dato_2`, realiza la suma y retorna el resultado y se imprime en la pantalla.

```
def sumar(dato_1,dato_2):
    salida= dato_1+dato_2
    return salida

def restar(dato_1,dato_2):
    salida= dato_1-dato_2
    return salida

def dividir(dato_1,dato_2):
    salida= dato_1/dato_2
    return salida

def multiplicar(dato_1,dato_2):
    salida= dato_1*dato_2
    return salida

def potenciar(dato_1,dato_2):
    salida= dato_1**dato_2
```



```
return salida

def radicar(dato_1,dato_2):
    salida= dato_1**(1/dato_2)
    return salida
#-----
dato_primer=9
dato_segundo=2

entrada= sumar(dato_primer,dato_segundo)
print (f"sumar {dato_primer} + {dato_segundo} = {entrada}")

entrada= restar(dato_primer,dato_segundo)
print (f"restar {dato_primer} - {dato_segundo} = {entrada}")

entrada= dividir(dato_primer,dato_segundo)
print (f"dividir {dato_primer} / {dato_segundo} = {entrada}")

entrada= multiplicar(dato_primer,dato_segundo)
print (f"multiplicar {dato_primer} * {dato_segundo} = {entrada}")

entrada= potenciar(dato_primer,dato_segundo)
print (f"potenciar {dato_primer} ** {dato_segundo} = {entrada}")

entrada= radicar(dato_primer,dato_segundo)
print (f"radicar {dato_primer} ** (1/{dato_segundo}) = {entrada}")

#Salida esperada por consola
sumar 9 + 2 = 11
restar 9 - 2 = 7
dividir 9 / 2 = 4.5
multiplicar 9 * 2 = 18
potenciar 9 ** 2 = 81
radicar 9 ** (1/2) = 3.0
```

En estas funciones dos enteros ingresados por el programador

```
dato_primer=9
dato_segundo=2
```

ya hemos visto como ingresar un objeto string con input, validarla y hacer un casting de string a entero.
Ahora generaremos en una función este ingreso, validación y casting



```

def ingresar_validar_cambiar(tipo_de_salida, texto_de_consulta):
    salida=""
    while salida == "":
        ingreso = input(f" {texto_de_consulta} :")
        if tipo_de_salida==str:
            salida = ingreso.title()
        elif tipo_de_salida==int and ingreso.isdecimal() is True:
            salida = int(ingreso)
        elif tipo_de_salida==float and ingreso.replace(".", "", 1).isdecimal() is True:
            salida = float(ingreso)
        else:
            continue
    return salida
#-----
desde_función = ingresar_validar_cambiar(str, "ingrese un texto")
print(f'{desde_función=}' clase:{type(desde_función)}")"

desde_función = ingresar_validar_cambiar(int, "ingrese un entero")
print(f'{desde_función=}' clase:{type(desde_función)}")"

desde_función = ingresar_validar_cambiar(float, "ingrese un flotante")
print(f'{desde_función=}' clase:{type(desde_función)}")"

```

#Salida esperada por consola

```

ingrese un texto :pYTHON eS gENIAL#<-----ingreso de la altura del usuario
desde función='Python Es Genial' clase:<class 'str'>
ingrese un entero :8#<-----ingreso de la altura del usuario
desde_función=8 clase:<class 'int'>
ingrese un flotante :3.14159#<-----ingreso de la altura del usuario
desde_función=3.14159 clase:<class 'float'>

```

Las funciones pueden tener parámetros opcionales con valores predeterminados, lo que permite llamar a la función sin proporcionar todos los argumentos.

Argumentos posicionales y por omisión (palabras clave):

Los argumentos posicionales se pasan a una función según el orden en el que se definen en la declaración de la función.

La sintaxis básica de argumentos posicionales ordenados es la siguiente:

```

def mi_función (parametro1, parametro2):
    # parametro1 tendrá el valor de 8
    # parametro2 tendrá el valor de 9
    return
mi_función (8,9)

```

Los argumentos de palabras clave se pasan a una función utilizando el nombre del parámetro al que se desea asignar el valor. Esto permite pasar los argumentos en cualquier orden.

La sintaxis de argumentos posicionales no ordenados es la siguiente:

```

def mi_función (parametro1, parametro2):
    # parametro1 tendrá el valor de 8
    # parametro2 tendrá el valor de 9
    return

```



```
#-----  
mi_función (parametro2=9, parametro1=8)
```

Parámetros arbitrarios en Python

Hay dos tipos de parámetros arbitrarios * y **:

Un asterisco * y recopilan todos los argumentos posicionales en una tupla.

```
def sumar(*numeros):  
    total = sum(numeros)  
    print("La suma es:", total)  
#-----  
sumar(1, 2, 3, 4, 5)
```

Dos asteriscos ** y recopilan todos los argumentos de palabras clave en un diccionario.

```
def multiplicar(**diccionario):  
    for clave, valor in diccionario.items()  
        print(f'{clave} - {valor}')  
#-----  
multiplicar( **{"1ro": 1, "2do": 2, "3ro": 3, "4to": 4, "5to": 5})
```

funciones con parámetros y operadores

En Python, el símbolo de asterisco * se utiliza para desempaquetar argumentos de una secuencia (como una lista o una tupla) y el símbolo de barra diagonal / se utiliza para separar los argumentos posicionales de los argumentos de palabras clave en la definición de una función. Veamos cómo se usan en cada caso:

Desempaquetar argumentos con *:

Si tienes una secuencia de valores y deseas pasarlos como argumentos individuales a una función, puedes utilizar el operador * para desempaquetar la secuencia.

Esto es útil cuando tienes una lista o una tupla y deseas pasar sus elementos como argumentos a una función que espera argumentos separados.

La sintaxis básica de parámetros tuplas con * es la siguiente:

```
def suma(a, b, c):  
    resultado = a + b + c  
    print(f'el resultado de la suma es {resultado}')  
#-----  
numeros = [1, 2, 3]  
suma(*numeros) # Equivalente a suma(1, 2, 3)  
-----  
def suma(*valores):  
    resultado = sum(valores)  
    print(f'el resultado de la suma es {resultado}')  
#-----  
numeros = [1, 2, 3]  
suma(*numeros) # Equivalente a suma(1, 2, 3)
```

Separar argumentos posicionales y de palabras clave con /:

En la definición de una función, el símbolo / se utiliza para marcar el punto en el que terminan los argumentos posicionales y comienzan los argumentos de palabras clave.

Esto es relevante cuando deseas definir una función que tenga una cantidad fija de argumentos posicionales y luego argumentos opcionales que se pueden especificar mediante palabras clave.



La sintaxis básica de parámetros con / es la siguiente:

```
def función(a, b, /, c, d):
    total = a + b + c + d
    print(f"La suma es ({a=}+{b=}+{c=}+{d=}):", total)
#-----
función(1, 2, 3, 4)
        =
función(1, 2, c=3, d=4)
        opciones
función(1, 2, d=4, c=3)
función(1, 2, c=3, d=4)
función(b=2, a=1, d=4, c=3)# no valido
```

En este ejemplo, a y b son argumentos posicionales, mientras que c y d son argumentos de palabras clave. Al marcar la posición de /, se indica que los primeros dos argumentos son posicionales y los últimos dos argumentos deben especificarse mediante palabras clave.

Orden de los argumentos

- posicionales
- *args para recibir un número variable de argumentos posicionales.
- **kwargs para recibir un número variable de argumentos clave-valor.
- por omisión.

Ámbitos (Scope) en Python

En Python, los ámbitos o scopes determinan la accesibilidad y visibilidad de los distintos tipos de objetos en diferentes partes del código.

Un objeto en ciertos ámbitos puede ser modificado reescrito, con funciones o con métodos de los objetos - Accesibilidad total.

Un objeto en ciertos ámbitos solo puede mostrar su contenido – Visibilidad y utilizar solo los métodos y atributos que no modifica el objeto.

- Ámbito Local (Local Scope):
- Ámbito Encerrado (Enclosing Scope)
- Ámbito Global (Global Scope):
- Ámbito de Nombre Integrado (Built-in Scope):

Un objetos con un solo dato creado en la raíz (root - sobre el margen) tipo str, int, float, bool, puede ser leídos y no modificados si no se les marca como globales.

Un objetos múltiple – Colección con un conjunto de datos creado en la raíz (root - sobre el margen) tipo list, set, dict,etc (no tuplas o frozensets), puede ser leídos y modificados sin necesidad de marcarlo como globales.

Si los objetos se crean en una función o método no existen en otros ámbitos - otras funciones o métodos

El siguiente contenido se tomó de bibliografía para los que requieren más información pero no son parte de este curso.

Ámbitos en Python

Ámbito Local (Local Scope):

El ámbito local se refiere al ámbito dentro de una función o método.

Las variables definidas dentro de una función solo son accesibles dentro de esa función.

Si se intenta acceder a una variable local fuera de su función, se producirá un error.

Los cambios realizados a variables locales no afectan a variables con el mismo nombre en otros ámbitos.



Ámbito Encerrado (Enclosing Scope)

- Se refiere al ámbito que se encuentra alrededor de una función anidada.
- Ocurre cuando hay una función definida dentro de otra función.
- Una función puede acceder a las variables de las funciones que la contienen, pero no viceversa.

Ámbito Global (Global Scope):

- El ámbito global se refiere al ámbito fuera de cualquier función o clase.
- Las variables definidas en el ámbito global son accesibles desde cualquier parte del código, incluyendo dentro de funciones y clases.
- Pueden ser modificadas desde funciones, pero se debe indicar explícitamente que se está utilizando la variable global.
- Definido a nivel de módulo o script.

Ámbito de Nombre Integrado (Built-in Scope):

- El ámbito de nombre integrado se refiere a las funciones y variables predefinidas que están disponibles en Python sin necesidad de importar ningún módulo.
- Incluye funciones y objetos integrados como print(), len(), range(), etc.
- Estas funciones y variables están disponibles en cualquier parte del código.
- Es importante tener en cuenta que hay una jerarquía en la resolución de nombres en Python, lo que significa que Python busca una variable en el ámbito local primero, luego en el ámbito encerrado, seguido del ámbito global y finalmente en el ámbito de nombre integrado. Si no se encuentra la variable en ningún ámbito, se genera un error de "NameError".

En este ejemplo, la variable x tiene un ámbito global y es accesible en todas partes del código. La variable y tiene un ámbito local dentro de la función func(), mientras que la variable z tiene un ámbito encerrado dentro de la función nested_func(). Las variables x, y y z son accesibles dentro de sus respectivos ámbitos.

Espero que esto aclare los conceptos de los ámbitos de los distintos objetos en funciones en Python. Si tienes más preguntas, no dudes en hacerlas.

```
# Ámbito global
x = 10
y = "Sin Datos root"
z = "Sin Datos root"
def func():
    # Ámbito local
    y = 20
    def nested_func():
        # Ámbito encerrado
        z = 30
        print(f"dentro de función anidada: {x=} {y=} {z=}") # Accede a las variables x, y y z
    nested_func()
    print(f"dentro de función: {x=} {y=} {z=}") # Accede a las variables x e y
func()
print(f"root: {x=} {y=} {z=}") # Accede a la variable x

dentro de función anidada: x=10          y=20          z=30
dentro de función:      x=10          y=20          z='Sin Datos root'
root:                 x=10          y='Sin Datos root' z='Sin Datos root'
```

Recursividad

Se denomina llamada recursiva (o recursividad), a aquellas funciones que en su algoritmo, hacen referencia a sí misma (se llama dentro de la misma función a sí misma).



Python admite las llamadas recursivas, permitiendo a una función, llamarse a sí misma, de igual forma que lo hace cuando llama a otra función.

La sintaxis básica recursividad en funciones es la siguiente:

```
def función_rec (parametro):
    #contenido de la función
    if condición:
        #cumple con la condicional
        return función_rec (parametro_modificado)
    else:
        #no cumple con la condicional
        return salida_final
regreso= función_rec (argumento)
```

```
def factorial_recurso(valor):
    if valor == 1:
        return 1
    else:
        return valor * factorial_recurso(valor-1)
salida = 10#<-----modificar
regreso = factorial_recurso(salida)

#Salida esperada por consola
el factorial de 10 es 3628800
```

```
def función_password (intento=1):
    respuesta = input("Ingrese su password (1234) :")
    if respuesta != "1234":
        if intento <= 3:
            print (f"\nError {intento}! Inténtalo de nuevo" )
            intento += 1
            función_password(intento) # Llamada recursiva
        else:
            print ("Ingreso no permitido" )
            exit()
    else:
        print ("Ingresaste!!!!!!" )
función_password()
```

```
#Salida esperada por consola - incorrecta
Ingrese su password (1234) :4321#<-----ingreso de la altura del usuario
```

```
Error 1! Inténtalo de nuevo
Ingrese su password (1234) :0000#<-----ingreso de la altura del usuario
```



Error 2! Inténtalo de nuevo

Ingrese su password (1234) :9876#<-----ingreso de la altura del usuario

Error 3! Inténtalo de nuevo

Ingrese su password (1234) :1212#<-----ingreso de la altura del usuario

Ingreso no permitido

#Salida esperada por consola - correcta

Ingrese su password (1234) :1234#<-----ingreso de la altura del usuario

Ingresaste!!!!!!

Anotaciones en funciones

Existe una funcionalidad relativamente reciente llamada function annotation o anotaciones en funciones. Dicha funcionalidad nos permite añadir metadatos a las funciones, indicando los tipos esperados tanto de entrada como de salida.

La sintaxis básica anotación en funciones es la siguiente:

```
def función_anot (ingreso : int)-> float:  
    #contenido de la función  
    #se espera el ingreso como entero  
    #cumple con la condicional  
    return ingreso/2# siempre una división regresa un float  
#-----  
salida = 5  
regreso = función_anot (salida)  
print (f'el e {salida} es {regreso} ")
```

Las anotaciones son muy útiles de cara a la documentación del código, pero no imponen ninguna norma sobre los tipos. Esto significa que se puede llamar a la función con un parámetro que no sea int, y no obtendremos ningún error.

```
def fibonacci_recursivo(valor):  
    if valor == 0:  
        return 0  
    elif valor == 1:  
        return 1  
    else:  
        return fibonacci_recursivo(valor-1) + fibonacci_recursivo(valor-2)  
salida = 10  
regreso = fibonacci_recursivo(salida)  
print (f'el fibonacci de {salida} es {regreso} ")  
  
#Salida esperada por consola  
el fibonacci de 10 es 55
```

Documentación (Docstrings):

```
Help! I need somebody  
Help! Not just anybody  
Help! You know I need someone  
Help!
```



Writer(s): John Lennon, Paul Mccartney

Docstrings son cadenas de texto que se utilizan para documentar la función y explicar su propósito, parámetros, comportamiento y demás información.

La sintaxis básica de docstrings es la siguiente:

```
def mi_función (parametro):
    """
    Este es el docstrings de "mi_función"
    mi_función (parametro):
        requiere un argumentos:
        retorna el argumento *5
        si el argumento es numérico multiplica en valor por 5
        si el argumento es string replica este 5 veces
    """
    return parametro*5
#-----
print(mi_función.__doc__)
help(mi_función)
print(f'{mi_función(5)=}')
print(f'{mi_función('cinco')=}')
```

```
def ingresar_validar_cambiar(tipo_de_salida, texto_de_consulta):
    """
    Este es el docstrings de la función
    la función ingresar_validar_cambiar (tipo_de_salida, texto_de_consulta)
    requiere dos argumentos:
        tipo_de_salida soporta tipo str, int y float
        texto_de_consulta un string con el mensaje para el usuario que explique el dato
        que se espera
    """
    salida=""
    while salida == "":
        ingreso = input(f'{texto_de_consulta} :')
        if tipo_de_salida==str:
            salida = ingreso.title()
        elif tipo_de_salida==int and ingreso.isdecimal() is True:
            salida = int(ingreso)
        elif tipo_de_salida==float and ingreso.replace(".", "", 1).isdecimal() is True:
            salida = float(ingreso)
        else:
            continue
    return salida
print(ingresar_validar_cambiar.__doc__)
help(ingresar_validar_cambiar)
desde_función = ingresar_validar_cambiar(str, "ingrese un texto")
print(f'{desde_función=} clase:{type(desde_función)}')

desde_función = ingresar_validar_cambiar(int, "ingrese un entero")
print(f'{desde_función=} clase:{type(desde_función)}')

desde_función = ingresar_validar_cambiar(float, "ingrese un flotante")
```



```
print(f"desde función={} clase:{type(desde función)}")
```

#Salida esperada por consola
`#print(ingresar_validar_cambiar.__doc__)`

Este es el docstrings de la función
la función ingresar validar cambiar (tipo de salida, texto de consulta)
requiere dos argumentos:
 tipo_de_salida soporta tipo str, int y float
 texto_de_consulta un string con el mensaje para el usuario que explique el dato
que se espera
`#help(ingresar_validar_cambiar)`
Help on function ingresar_validar_cambiar in module __main__:
ingresar_validar_cambiar(tipo_de_salida, texto_de_consulta)

Este es el docstrings de la función
la función ingresar validar cambiar (tipo de salida, texto de consulta)
requiere dos argumentos:
 tipo_de_salida soporta tipo str, int y float

 texto_de_consulta un string con el mensaje para el usuario que explique el dato
que se espera
ingrese un texto :.....

Retorno

1) Retorno único:

En algunos lenguajes de programación cada función tiene solamente un return. En Python no es así.
Aunque la comunidad prefiere que solo se coloque uno.
Por otra parte en breve con try except veremos que el return puede ser interceptado y descartado.

La sintaxis básica de retorno único o múltiple:

```
def mi_función (parametro):
    if parametro.isdecimal():
        return int(parametro)
    else:
        return False
#-----
print (f"mi_función(5)={}")
print (f"mi_función(' cinco ')={}")
```

2) Empaquetados y desempaquetados.

Cuando es necesario retornar más de un objeto al llamamiento de la función se retorna separados por comas, esto genera una tupla, no obstante, podemos retornar diccionarios y otros objetos

Nota:

Recuerden que dos o más objetos separados por coma forman una tupla, estén o no entre paréntesis ()

La sintaxis básica de empaquetados es la siguiente:

```
def mi_función (parametro):
    if isinstance(parametro, str):
        if parametro.isdigit() and parametro < 18:
            return int(parametro), "Ud no puede ingresar al bar"
```



```

    elif parametro.isdigit() and parametro >= 18:
        return int(parametro), "Bienvenido, Ud puede beber alcohol"
    elif isinstance(parametro, (int,float)):
        if parametro < 18:
            return parametro, "Ud no puede ingresar al bar"
        elif parametro >= 18:
            return parametro, "Bienvenido, Ud puede beber alcohol"
    else:
        return None
#-----
print (f"mi_función(30)={}")
print (f"mi_función(15)={}")
print (f"mi_función(' quince ')={")
```

La sintaxis básica desempaquetados es la siguiente:

```

def mi_función():
    return "Python","es genial"
#-----
regreso = mi_función()
print (f"regreso={}")
regreso_0, regreso_1 = mi_función()
print (f"primer valor de regreso {regreso_0}=")
print (f"segundo valor de regreso {regreso_1}=")
```

Ámbitos de los distintos objetos. Scope

El ámbito (scope) en Python se refiere a la región del programa donde una variable es accesible y puede ser utilizada. En Python, el ámbito de una variable puede ser local, global o integrado (built-in).

Objetos inmutables

Ámbito Local:

Dentro de una función, una variable string definida dentro de esa función tiene ámbito local y solo es accesible dentro de esa función.

```

def ejemplo():
    mensaje_local = "Hola, mundo Python"
    print(mensaje)
ejemplo()
print(f"mensaje_local={}")# No puedes acceder a 'mensaje_local' fuera de la función ya que se creo dentro
```

Ámbito Global:

Una variable string definida fuera de cualquier función tiene ámbito global y es accesible en todo el programa.

```

mensaje_global = "Hola, mundo Python"
def ejemplo():
    print(f"mensaje_global={}") # Solo se puede leer el contenido de mensaje_global
    mensaje_global += " ¡2024!"# No se puede modificar (escribir) el contenido de mensaje_global
ejemplo()
```

Puedes modificar una variable string definida globalmente dentro de una función.

```

mensaje_global = "Hola, mundo Python"
def modificar_mensaje():
    global mensaje_global#Solo se puede leer el contenido de mensaje_global
```



```
mensaje_global += " ¡2024!" #No se puede modificar (escribir) el contenido de mensaje_global
modificar_mensaje()
print(f" {mensaje_global} ")
```

Objetos mutables

```
#una lista definida dentro de una función tiene ámbito local y solo es accesible dentro de esa función.
def ejemplo_lista():
    lista_numeros = [1, 2, 3]
    print(f" {lista_numeros} ")
ejemplo_lista()
print(f" {lista_numeros}")# No puedes acceder a 'lista_numeros' fuera de la función
```

Una lista definida fuera de cualquier función tiene ámbito global y es accesible en todo el programa.

```
lista_global = [4, 5, 6]
def ejemplo_lista():
    lista_global.append("fin")
    print(f" {lista_global} ")
ejemplo_lista()
print(f" {lista_global} ")
```

Anidación de funciones

Es la posibilidad de definir funciones dentro de otras funciones. Esto permite crear funciones internas que solo son accesibles desde la función que las contiene. La anidación de funciones puede ser útil para modularizar el código y encapsular lógica específica.

```
def función_externa(x):
    def función_interna(y):
        return y * 2
    resultado_interno = función_interna(x)
    return resultado_interno + 5

# Llamada a la función externa
resultado_final = función_externa(3)
print(f" {resultado_final} ")
```

La función interna solo es visible dentro de la función externa. Si intentas llamar a función_interna fuera de función_externa resultaría en un error, ya que no está definida en ese ámbito.

Llamado a funciones desde/como un string.

Puedes guardar el nombre de una función como una cadena (string) y luego usar la función globals() para obtener la referencia a la función por su nombre.

```
def mi_función():
    print("¡Hola desde la función!")

# Guardar el nombre de la función como string
nombre_de_función = 'mi_función'
# Obtener la referencia a la función por su nombre
función = globals()[nombre_de_función]

# Invocar la función
función()
```

En este ejemplo, globals() devuelve un diccionario que contiene todos los nombres de las variables y funciones



globales en tu script, y puedes acceder a la función por su nombre como una clave en ese diccionario.

Ten en cuenta que este enfoque funciona solo para funciones globales, no para funciones dentro de clases u otros ámbitos más complejos. Si necesitas algo más avanzado, podrías explorar el módulo `getattr()` o usar un diccionario para mapear nombres de funciones a funciones directamente.

En este caso, `función_con_parametros` acepta dos parámetros, y al guardar y luego invocar la función por su nombre, también proporcionamos los valores necesarios para esos parámetros, luego pasar los valores necesarios al invocar la función.

Este enfoque también funciona si tienes una función que acepta un número variable de argumentos utilizando `*args` y `**kwargs`.

```
def función_con_parametros(parametro1, parametro2):
    print(f"Parámetro 1: {parametro1}")
    print(f"Parámetro 2: {parametro2}")

# Guardar el nombre de la función como string
nombre_de_función = 'función_con_parametros'

# Obtener la referencia a la función por su nombre
función = globals()[nombre de función]

# Definir los valores de los parámetros
valor_parametro1 = "Hola"
valor_parametro2 = "Mundo"

# Invocar la función con los parámetros
función(valor_parametro1, valor_parametro2)
```



Módulo 10 – Funciones lambda, orden superior y Lazy evaluation:

Funciones lambda:

- argumentos:
- expresión:

Funciones de orden superior.

- map
- Filter
- Reduce

Lazy evaluation



Funciones lambda:

Las funciones lambda, también conocidas como funciones anónimas, son funciones pequeñas y de una sola línea que se definen sin un nombre utilizando la palabra clave lambda.

En realidad, si bien la comunidad recomienda no usar nombres la mayoría de los ejemplos lo tienen

Son útiles cuando necesitas una función simple de una línea sin tener que definirla formalmente utilizando def.

La sintaxis básica de empaquetados y desempaquetados en Python es la siguiente:

lambda argumentos: expresión

argumentos:

Si los hay son los argumentos de la función separados por comas.

expresión:

Es una expresión que define el cuerpo de la función lambda.

Esta expresión se puede evaluar.

Se devuelve como resultado de la función.

```
"""
def suma(a, b):
    return a + b
"""

suma = lambda a, b: a + b
print(f'{suma(3,6)=}')
```

Salida esperada por consola
suma(3,6)=9

```
"""
def salida(a, b):
    return a + b if a>b else a - b
"""

salida = lambda a, b: a + b if a>b else a - b
print(f'{salida(3,6)=}')
print(f'{salida(6,3)=}')

# Salida esperada por consola
salida(3,6)=-3
salida(6,3)=9
```



Funciones de orden superior.

Funciones de orden superior.

En Python todos son objetos, las funciones también.

Las funciones de orden superior en Python son aquellas funciones que pueden aceptar otras funciones como argumentos de entrada, manipularlas de manera flexible dentro de otras funciones y al final devolver funciones como resultado.

En Python, las funciones de orden superior se benefician del hecho de que las funciones son ciudadanos de primera clase, lo que significa que se pueden tratar como cualquier otro objeto, como enteros, cadenas o listas. Esto permite realizar operaciones avanzadas con funciones, como pasarlas como argumentos a otras funciones, asignarlas a variables, almacenarlas en estructuras de datos, retornarlas como resultado de una función, etc.

```
def aplicar_operacion(funcion_entrada, numeros):
    resultado = []
    for numero in numeros:
        resultado.append(funcion_entrada(numero))
    return resultado

def cuadrado(x):
    return x ** 2

def cubo(x):
    return x ** 3
#-----
numeros = [1, 2, 3, 4, 5]

resultado_cuadrado = aplicar_operacion(cuadrado, numeros)
print(f"resultado_cuadrado={resultado_cuadrado}")
resultado_cubo = aplicar_operacion(cubo, numeros)
print(f"resultado_cubo={resultado_cubo}")

# Salida esperada por consola
resultado_cuadrado=[1, 4, 9, 16, 25]
resultado_cubo=[1, 8, 27, 64, 125]
```

```
def mi_funcion(operacion):
    if operacion == "suma":
        def suma(a, b):
            return a + b
        return suma
    elif operacion == "resta":
        def resta(a, b):
            return a - b
        return resta
```



```
return a - b
return resta
-----
operacion = mi_funcion("suma")
resultado = operacion(3, 4)
print(f"suma=",resultado) # Salida: 7
operacion = mi_funcion("resta")
resultado = operacion(8, 4)
print(f"resta=",resultado) # Salida: 4
resultado = mi_funcion("suma")(8, 2)
print(f"suma=",resultado) # Salida: 6

# Salida esperada por consola
suma= 7
resta= 4
suma= 10
```

map

La función map() en Python es una función de orden superior que se utiliza para aplicar una función original dada a cada elemento de un iterable (como una lista, tupla o set) y devuelve un objeto map que contiene los resultados. Es muy útil cuando necesitas aplicar una función a múltiples elementos de una colección iterable de manera eficiente y elegante de procesar datos en Python.

La sintaxis básica de la función map() es la siguiente:

```
map(función, iterable)
```

Esto genera un objeto tipo map, por lo que se suele ver como
list(map(función, iterable))

```
def cuadrado(cada_numero):
    resultado = cada_numero ** 2
    return resultado
-----
numeros = [1, 2, 3, 4, 5]
regreso = map(cuadrado, numeros)
print (f'{regreso=}')
print (f'{type(regreso)=}')
print (f'{list(regreso)=}')

# Salida esperada por consola
regreso=<map object at 0x000001B673BD5E70>
type(regreso)=<class 'map'>
list(regreso)=[1, 4, 9, 16, 25]
```



Se define la función estándar con el nombre cuadrado(parámetro) con un return parámetro al cuadrado (**2)

Map aplica esta función a cada elemento de la lista números. Los regresos se coleccionan en un objeto map. Para ver su contenido hacemos un list(objeto map) o un for de cada dato map

```
palabras = ["Python", "es", "genial"]
# len como función original
longitudes = map(len, palabras)
print(f"longitudes={}")
print(f"type(longitudes)={}")
for palab, longi in zip(palabras, longitudes):
    print(f"palabra:{palab} longitud-len= {longi}")

# Salida esperada por consola
longitudes=<map object at 0x00000221058A5D20>
type(longitudes)=<class 'map'>
palabra:'Python' longitud-len= 6
palabra:'es' longitud-len= 2
palabra:'genial' longitud-len= 6
```

Len es una función built-in incorporada para calcular la longitud de un objeto iterable, en este caso un string.

Map() se aplica a la función len() y se genera un iterable con las longitudes correspondientes.

Es importante tener en cuenta que map() devuelve un objeto map que es un iterable. Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.



Filter

filter() es una función de orden superior que se utiliza para filtrar elementos de un iterable (como una lista, tupla o set) según una función original de filtrado dada.

Es importante tener en cuenta que filter() devuelve un objeto filter que es un iterable. Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.

La sintaxis básica de la función filter() es la siguiente:

```
filter(función, iterable)
```

Esto genera un objeto tipo filter, por lo que se suele ver como
`list(filter(función, iterable))`

En este caso la función debe ser un filtro que solo regrese un True o un False.

Filter aplica a cada elemento del iterable de la colección y True se incluirá en el objeto filter mientras que los False serán excluido.

iterable: Es el objeto iterable (como una lista, tupla, conjunto) del que se desean filtrar los elementos.

La función filter() es útil cuando necesitas seleccionar elementos específicos de un iterable según una condición. Proporciona una forma concisa y eficiente de filtrar datos en Python.

```
def es_par(numero):# original
    return numero % 2 == 0
#
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

solo_pares = filter(es_par, numeros)
print (f"{'solo_pares='}")
print (f"{'type(solo_pares)=='}")
for cada_numero in (solo_pares):
    print (f"\t\t{número} {cada_numero=}")

# Salida esperada por consola
solo_pares=<filter object at 0x000001C3A6DC5D20>
type(solo_pares)=<class 'filter'>
    número cada_numero=2
    número cada_numero=4
    número cada_numero=6
    número cada_numero=8
    número cada_numero=10
```



Reduce

Es necesario recordar que reduce se encuentra en el Módulo functools y requiere ser importada antes de su uso.

reduce() es una función de orden superior que se utiliza para aplicar una función acumulativa original a los elementos de un iterable y reducirlos a un solo valor.

La sintaxis básica de la función reduce() es la siguiente:

```
from functools import reduce  
reduce(función, iterable)  
reduce(función, iterable, [initializer])
```

Al ser una función acumulativa que se aplicará a los elementos del iterable. Esta función original debe tomar dos argumentos y devolver el resultado de la operación acumulativa.

iterable: Es el objeto iterable (como una lista, tupla, conjunto) que se desea reducir a un solo valor.

initializer (opcional): Es un valor inicial que se utiliza como el primer argumento en la primera llamada a la función acumulativa. Si no se proporciona, el primer elemento del iterable se utiliza como valor inicial.

Es importante tener en cuenta que reduce() requiere al menos dos elementos en el iterable. Si el iterable está vacío y no se proporciona un valor inicial (initializer), se generará un error.

La función reduce() es útil cuando necesitas realizar una operación acumulativa en los elementos de un iterable y reducirlos a un solo valor. Algunos ejemplos comunes de uso incluyen la suma de todos los elementos, el cálculo del producto de todos los elementos, la concatenación de cadenas, entre otros.

Espero que esta explicación aclare cómo funciona la función reduce() como una función de orden superior en Python. Si tienes más preguntas, no dudes en hacerlas.

```
from functools import reduce  
def sumar(acumulador, valor_nuevo ):  
    print(f"\t{acumulador} {valor_nuevo}")  
    return valor_nuevo + acumulador  
#-----  
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
resultado = reduce(sumar, numeros)  
print (f"resultado={resultado}")  
print (f"type(resultado)={type(resultado)})")  
  
# Salida esperada por consola  
acumulador=1      valor_nuevo=2  
acumulador=3      valor_nuevo=3  
acumulador=6      valor_nuevo=4  
acumulador=10     valor_nuevo=5  
acumulador=15     valor_nuevo=6  
acumulador=21     valor_nuevo=7  
acumulador=28     valor_nuevo=8  
acumulador=36     valor_nuevo=9
```



```
acumulador=45      valor_nuevo=10
resultado=55
type(resultado)=<class 'int'>
```

En este ejemplo, se define la función `sumar(acumulador, valor_nuevo)` que toma dos argumentos y devuelve su suma `return valor_nuevo + acumulador`.

Esto se itera para cada elemento de la colección y acumula sucesivamente los elementos del return
Pueden comparar con `print(sum(numeros))`

La sintaxis de las funciones lambda + funciones de orden superior es la siguiente:

```
lambda <parámetro> :expresión
Funciones anónimas que en Python pueden llevar nombre :(

✓ lambda + map
✓ lambda + filter
✓ lambda + reduce (from functools import reduce)
```

```
string = "Python es el mejor lenguaje de programación"
palabras = string.split()
print(f"palabras={}")
longitud_mayor_a_5 = filter(lambda palabra: len(palabra) >= 5, palabras)
print(f"list(longitud_mayor_a_5)={}")

# Salida esperada por consola
palabras=['Python', 'es', 'el', 'mejor', 'lenguaje', 'de', 'programación']
list(longitud_mayor_a_5)=['Python', 'mejor', 'lenguaje', 'programación']
```

En este ejemplo, se utiliza una función lambda para verificar si la longitud de una cadena es mayor o igual a 5. `filter()` se aplica a la función lambda y a cada elemento de la lista de palabras, y se generan objeto tipo filter que es un iterable con las palabras con longitud ≥ 5 .

Para ver los resultados, debes convertirlo a una lista o recorrerlo en un bucle.

```
todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
resultado = filter(lambda x: x % 2 == 0, todos)

print(f" la salida de filter es {resultado}")
print(f" type {type(resultado)}")
print(f" la lista de la salida de filter es {list(resultado)}")

# Salida esperada por consola
la salida de filter es <filter object at 0x000001E57FA8F790>
type <class 'filter'>
```



la lista de la salida de filter es [2, 4, 6, 8, 0]

La función filter() es útil cuando necesitas seleccionar elementos específicos de un iterable según una condición de filtrado. Puedes utilizar una función definida por el usuario o una función lambda para especificar la condición de filtrado. Esto te permite realizar operaciones más complejas de selección y filtrado en tus datos.

```
todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
pares = [2, 4, 6, 8, 10]

resultado = filter(lambda xy: xy[0] in pares and xy[1] in pares, zip(todos, pares))
print(f" la salida de filter es {resultado}")
print(f" type {type(resultado)}")
print(f" la lista de la salida de filter es {list(resultado)}")

# Salida esperada por consola
la salida de filter es <filter object at 0x000001A994805720>
type <class 'filter'>
la lista de la salida de filter es [(2, 4), (4, 8)]
```

Una función lambda para verificar si los elementos de números y pares están presentes en la lista pares. filter() se aplica a la función lambda y a los iterables numeros y pares, y se generan los elementos que cumplen con la condición.



Lazy evaluation (Evaluación perezosa):

La evaluación perezosa es una estrategia de evaluación utilizada en algunos lenguajes de programación que consiste en posponer la evaluación de una expresión hasta que sea necesaria o se requiera su resultado. En lugar de calcular todos los valores de una expresión de forma inmediata, la evaluación perezosa permite evaluar solo los valores necesarios en un momento dado.

La evaluación perezosa es especialmente útil cuando se trabaja con estructuras de datos potencialmente grandes o cuando se tienen expresiones complejas donde no todos los valores son necesarios en todos los casos. Al posponer la evaluación, se puede evitar el cálculo innecesario de valores que no se utilizarán.

En Python, la evaluación perezosa se puede lograr utilizando generadores, que son objetos que generan una secuencia de valores bajo demanda. Los generadores se definen utilizando la sintaxis de comprensión de listas pero con paréntesis en lugar de corchetes.

```
todos = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

cuadrados_en_reserva = (x ** 2 for x in todos)

# los primeros 3 cuadrados
print(f"primer valor:",next(cuadrados_en_reserva))
print(f"segundo valor:",next(cuadrados_en_reserva))
print(f"tercer valor:",next(cuadrados_en_reserva))

primer valor: 1
segundo valor: 4
tercer valor: 9
```

En este ejemplo, en lugar de generar una lista completa con los cuadrados de los números, se crea un generador cuadrados_en_reserva que produce los cuadrados bajo demanda. Al llamar a la función next(cuadrados_en_reserva), se obtiene el siguiente valor del generador.

La evaluación perezosa tiene la ventaja de ahorrar recursos y tiempo de ejecución al evitar cálculos innecesarios. Sin embargo, es importante tener en cuenta que los generadores son un tipo de iteradores y solo pueden ser recorridos una vez. Una vez que se agotan los valores del generador, no se pueden obtener más elementos.

En resumen, es una estrategia que permite posponer la evaluación de expresiones hasta que sea necesario, lo que puede ser beneficioso en términos de eficiencia y rendimiento en ciertos escenarios



Módulo 11 – Manejo de excepciones:

- try
- except
- else
- finally
- raiser

Es muy importante aclarar que el flujo del programa se altera con try except y finally



Manejo de excepciones

En un principio se trabajaba con manejo de errores. Estos fueron mejorados hasta que no tengan que ser solo errores, sino que cualquier tipo de excepción o eventos que ocurren durante la ejecución y que interrumpen el flujo normal del programa.

El manejo de excepciones es una técnica utilizada para controlar y gestionar estas situaciones excepcionales que pueden ocurrir durante la ejecución de un programa.

En Python, el manejo de excepciones se realiza utilizando bloques

- ✓ try
 - ✓ except
 - ✓ else
 - ✓ finally
- y la función
- ✓ raiser

El bloque try se utiliza para encapsular el código que puede generar una excepción,

El bloque except se utiliza para manejar las excepciones capturadas.

El bloque else se utiliza tras un evento que no ingresa por excepción.

El bloque finally se utiliza como último bloque. Esto es muy importante en una función con return

```
try:
    resultado = 10 / 0 #<----- Dividimos sobre cero
except Exception as Error:
    print(f"Error encontrado: {Error}")

# Salida esperada por consola
Error encontrado: division by zero
```

```
try:
    resultado = 2 * x #<----- o x no existe
except Exception as Error:
    print(f"Error encontrado: {Error}")

# Salida esperada por consola
Error encontrado: name 'x' is not defined
```

Es posible utilizar varios bloques except para capturar diferentes tipos de excepciones y manejarlas de manera adecuada. Además, se puede utilizar un bloque else para especificar código que se ejecutará solo si no se generó ninguna excepción en el bloque try. También se puede utilizar un bloque finally para especificar código que se ejecutará siempre, independientemente de si se generó una excepción o no.



```
try:  
    ingreso = int(input("Ingrese un número: "))  
    resultado = 10 / ingreso  
    print(f'resultado = {resultado}')  
except ValueError:  
    print("Error: Ingrese un número válido")#< ----- el usuario no ingreso un entero  
except ZeroDivisionError:  
    print("Error: División entre cero")#< ----- el usuario ingreso 0 cero  
  
#-----  
else:  
    print("¡No se produjo ninguna excepción!")  
finally:  
    print ("the end..... :)")
```

Salida esperada por consola

Ingrese un número: **5**#<-----ingreso de la altura del usuario
resultado = 2.0
¡No se produjo ninguna excepción!
the end..... :)

Sin errores y al final el bloque finally

Salida esperada por consola

Ingrese un número: **dos**#<-----ingreso de la altura del usuario
Error: Ingrese un número válido
the end..... :)

En este ejemplo, se intenta realizar una casting de str a int no valido, lo cual generará una excepción ValueError. El bloque except captura esta excepción y ejecuta el código especificado dentro de él, en este caso, imprime un mensaje de error.

```
#-----  
Ingrese un número: 0#<-----ingreso de la altura del usuario  
Error: División entre cero  
the end..... :)
```

En este ejemplo, se intenta realizar una división entre cero, lo cual generará una excepción ZeroDivisionError. El bloque except captura esta excepción y ejecuta el código especificado dentro de él, en este caso, imprime un mensaje de error.

Python no se rompe. El manejo de excepciones es una técnica para detectar y responder adecuadamente a situaciones excepcionales sin que el script se detenga abruptamente.

"raiser" es una declaración para generar manualmente no un error, sino una "excepción" programada, lanzar explícitamente en un lugar o tiempo específico del código.



La sintaxis básica de la declaración raise es la siguiente:

```
raise TipoDeExcepcion("Mensaje de error opcional")
```

Donde TipoDeExcepcion es el tipo de excepción que se desea lanzar, como ValueError, TypeError, FileNotFoundError, o un error opcional que se mostrará cuando se capture la excepción.

```
def dividir(a, b):
    if b == 0:
        raise ValueError("Nunca dividirás por cero")
    return a / b
try:
    resultado = dividir(10, 0)
except ValueError as Error:
    print(f'Error encontrado: {Error}')
# Salida esperada por consola
Error encontrado: Nunca dividirás por cero
```

La declaración raise se lanza cuando el programador encuentra un evento por el cual genera una excepción propia en este caso un ValueError con el mensaje "Nunca dividirás por cero".

Luego, se captura la excepción en el bloque except y se imprime el mensaje.

No hay error, no hay excepción por bloqueo de script. Raise permite controlar el flujo del programa y señalar condiciones excepcionales de acuerdo con tus necesidades.

Finally:

```
def mi_funcion_dividir(valor_1,valor_2):
    try:
        return valor_1 / valor_2
    except Exception as Error:
        return f'Error encontrado: {Error}'
    finally:
        return "Este es el retorno final"
print (f'{mi_funcion_dividir(10,2)=}')
print (f'{mi_funcion_dividir(10,0)=}')
# Salida esperada por consola
mi_funcion_dividir(10,2) = Este es el retorno final
# Salida esperada por consola
mi_funcion_dividir(10,0) = Este es el retorno final
```

No importa que return se ejecute primero. Siempre el que vale es el de finally



Módulo 12 – Manejo de información en archivos externos y directorios:

El manejo de archivos es una tarea común y útil para leer y escribir información en files.

Primero veremos el manejo del archivo local, en la misma carpeta donde tenemos nuestro archivo Python.

Luego veremos los sistemas de carpetas y discos que depende en mucho del sistema operativo del equipo.



Antes se usaba simplemente open para abrir el archivo externo y luego close para cerrarlo. Hoy en día se recomienda el uso de with que permite una apertura y cierre casi inmediato.

En principio veremos solamente dos argumentos para open el archivo o **file** y el **modo**

El nombre del archivo debe tener una extensión que corresponda al tipo de escritura que contiene.

Texto plano	txt
Excel	xls, xlsx o xlsm
Comma Separated Values	csv
JavaScript Object Notation.	json
Pickle	pkl
eXtensible Markup Language	xml
etc.	

La primera diferencia es si se graba como texto plano (el que podemos leer con un block de notas, vim, etc.) y los que se escriben en binario.

Los textos planos usan la t de text o la omiten, los binarios deben usarse con la b de binary

Se pueden escribir (write) , leer (read), agregar al final (append) y x para crearlo sin contenido

Modo	Modo	Admite	Descripción de la acción
------	------	--------	--------------------------



texto	binario	+	
r / rt (por defecto)	rb	+ Si lectura y escritura	Read Abre un archivo de solo lectura (texto o binario). (Debe existir) El puntero de archivo se coloca al principio del archivo.
w / wt	wb	+ Si escritura y lectura	Write Abre un archivo (nuevo si no existe) solo para escritura (texto o binario). Sobrescribe el archivo si existe.
a / at	ab	+ Si agregar y leer	Append Abre un archivo para agregar información en el (texto o binario). Si el archivo no existe, crea un nuevo archivo para escribir.
x			Crea un archivo vacío. Genera un error si ya existe

Hay muchos más argumentos que no son parte del curso.

buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=None

```
https://es.wikipedia.org/wiki/Windows-1251
https://es.wikipedia.org/wiki/Windows-1252
https://es.wikipedia.org/wiki/UTF-8
https://es.wikipedia.org/wiki/UTF-16
https://es.wikipedia.org/wiki/Unicode
```

Para decodificar ASCII, CP1251 / 52 o cualquier otra codificación a UTF-8 o Unicode en Python, puedes usar el método decode de las cadenas de texto o el Módulo codecs

```
.encode y .decode debe darse al menos como idea, principalmente porque Windows tiene cp1251 y UTF8 - Unicode en Python y veremos que hay mucho material que requiere este concepto
cp1251_text = b'\xd1\xf2\xe0\xee\xf2\xe8\xf0\xe0'
utf8_text = cp1251_text.decode('cp1252').encode('utf-8')
print(utf8_text.decode('utf-8'))

# Decoding from ASCII to UTF-8
ascii_text = b'Hello, World!'
utf8_text = ascii_text.decode('ascii').encode('utf-8')
print(utf8_text.decode('utf-8'))
```

Aquí decodificamos el texto codificado desde la codificación especificada (CP1251 o ASCII) usando el método decode. Luego, lo codificamos en UTF-8 usando el método encode. Finalmente, decodificamos el texto UTF-8 para obtener la cadena Unicode.



En Python, los objetos de cadena de texto (str) utilizan el formato de codificación Unicode por defecto. Unicode es un estándar que asigna un número único (llamado punto de código) a cada carácter utilizado en la mayoría de los sistemas de escritura del mundo.

Cuando se trata de codificación de bytes, Python utiliza el formato UTF-8 como codificación predeterminada. UTF-8 es una forma de codificar los puntos de código Unicode en secuencias de bytes variables, lo que permite representar de manera eficiente caracteres de múltiples idiomas.

En resumen, Python utiliza objetos de cadena de texto Unicode (str) internamente y utiliza UTF-8 como codificación predeterminada al trabajar con bytes.

```
salida="""En Python, los objetos de cadena de texto (str) utilizan el formato de codificación Unicode por defecto.
```

```
Cuando se trata de codificación de bytes, Python utiliza el formato UTF-8 como codificación predeterminada. """
```

```
objeto_io = open ("borrar.txt",mode="w")
objeto_io.write(salida)
objeto_io.close()
```

```
contenido_objeto_python='En Python, los objetos de cadena de texto (str) utilizan el formato de codificación Unicode por defecto.\nCuando se trata de codificación de bytes, Python utiliza el formato UTF-8 como codificación predeterminada. '
```

Escritura:

Formato antiguo

```
salida="""En Python, los objetos de cadena de texto (str) utilizan el formato de codificación Unicode por defecto.
```

```
Cuando se trata de codificación de bytes, Python utiliza el formato UTF-8 como codificación predeterminada. """
```

```
objeto_io = open ("borrar.txt",mode="w")
objeto_io.write(salida)
objeto_io.close()
```

Formato moderno

```
with open ("borrar.txt",mode="w") as objeto_io:
    objeto_io.write(salida)
```

Ir a consola al directorio donde están nuestros archivos y veremos que encontraremos el archivo **borrar.txt** con fecha y hora del momento que corriste



Lectura:

Para leer el contenido de un archivo, se utilizan los métodos `read()`, `readline()` o `readlines()`

Formato antiguo

```
objeto_InOut = open("borrar.txt", "r")
contenido_objeto_python = objeto_InOut.read()
print(f'{contenido_objeto_python}')
objeto_InOut.close()
```

Formato moderno

```
with open("borrar.txt", "r") as objeto_InOut:
    contenido_objeto_python = objeto_InOut.read()
print(f'{contenido_objeto_python}')
```

ambos me darán el mismo resultado

contenido_objeto_python='En Python, los objetos de cadena de texto (str) utilizan el formato de codificación Unicode por defecto.\nCuando se trata de codificación de bytes, Python utiliza el formato UTF-8 como codificación predeterminada. '

En realidad, hay diferencias muy importantes.

No requiere cerrar (`close()`) ya que al salir del bloque `with` esto se hace automáticamente.

Por otro lado, el objeto de intercambio no existe fuera del `with` y ocupa mucha menos memoria

Lee una línea del archivo

```
with open("borrar.txt", "r") as objeto_InOut:
    línea = objeto_InOut.readline() # Lee una línea del archivo
print(f'{línea}')
```

#salida esperada por consola

línea='En Python, los objetos de cadena de texto (str) utilizan el formato de codificación Unicode por defecto.\n'

Se puede trabajar en un bucle `for` leyendo línea a línea

Lee la totalidad de líneas del archivo y guarda cada línea como objeto de una lista

```
with open("borrar.txt", "r") as objeto_InOut:
    líneas = objeto_InOut.readlines() # Lee una línea del archivo
print(f'{líneas}')
```

#salida esperada por consola

líneas=['En Python, los objetos de cadena de texto (str) utilizan el formato de codificación Unicode por defecto.\n', 'Cuando se trata de codificación de bytes, Python utiliza el formato UTF-8 como codificación predeterminada. ']

La serialización de archivos es el proceso de convertir una estructura de datos o un objeto en una secuencia de bytes para que pueda ser almacenado en un archivo o transmitido a través de una red. La serialización permite guardar el estado de un objeto de manera persistente, lo que significa que se puede recuperar posteriormente y restaurar en su forma original.



La serialización es útil cuando necesitas almacenar objetos complejos en disco, transferir datos entre diferentes plataformas o comunicarte con sistemas distribuidos. Al serializar un objeto, se guarda su estado actual, incluyendo los valores de sus atributos y la estructura de los datos.

La serialización puede realizarse en diferentes formatos, como JSON (JavaScript Object Notation), XML (eXtensible Markup Language), protocol buffers, entre otros. Estos formatos definen reglas y convenciones para convertir los datos en una representación legible por máquina.

En Python, el Módulo por excelencia es **pickle** proporciona funcionalidad para serializar y deserializar objetos en formato binario. También existen otras bibliotecas populares de serialización como json y xml.etree.ElementTree que permiten la serialización en formatos JSON y XML, respectivamente.

Serialización:

Consiste en convertir un objeto de Python (normalmente una lista o diccionario) en un string.

Deserialización:

Consiste en convertir un string en un objeto de Python (normalmente una lista o diccionario).

```
import pickle
obj_diccionario={"uno":{"inglés":"One","francés":"Un"},
                  "dos":{"inglés":"Two","francés":"Deux"},
                  "tres":{"inglés":"Three","francés":"Trois"},
                  "cuatro":{"inglés":"Four","francés":"Quatre"},
                  "cinco":{"inglés":"Five","francés":"Cinq"}
                  }
with open ("borrar.pkl" , mode="wb") as objeto_InOut:
    pickle.dump(obj_diccionario,objeto_InOut)

#-----
with open ("borrar.pkl" , mode="rb") as objeto_InOut:
    nuevo=pickle.load(objeto_InOut)
print (f"nuevo={nuevo}")

#salida esperada por consola
nuevo={'uno': {'inglés': 'One', 'francés': 'Un'}, 'dos': {'inglés': 'Two', 'francés': 'Deux'}, 'tres': {'inglés': 'Three', 'francés': 'Trois'}, 'cuatro': {'inglés': 'Four', 'francés': 'Quatre'}, 'cinco': {'inglés': 'Five', 'francés': 'Cinq'}}
```

DUMP: Graba un objeto Python a un archivo JSON

DUMPS: Convierte un objeto Python a uno JSON (objetos json se envían por puertos a otras apps)

LOAD: Lee un archivo JSON a un objeto Python

LOADS: Convierte un objeto JSON a uno Python (objetos json se envían por puertos a otras apps)

Objetos Python	Objetos JSON
list, tuple	array



str	string
int, float, int & float-derived Enums	number
True	true
False	false
dict	object
None	null

indent define el número de indents
sort_keys ordena el resultado
separators=(". ", " = ") Cambia el separador default

Syntax:

```
json.dumps(
    obj,
    *,
    skipkeys=False,
    ensure_ascii=True,
    check_circular=True,
    allow_nan=True,
    cls=None,
    indent=None,
    separators=None,
    default=None,
    sort_keys=False,
    **kw
)
```

Parameters:

obj: serializar obj como una secuencia con formato JSON

skipkeys:

Si skipkeys es verdadero (predeterminado: Falso), las teclas de dictado que no sean de un tipo básico (str, int, float, bool, None) se omitirán en lugar de generar un TypeError.

ensure_ascii:

Si asegurar_ascii es verdadero (el valor predeterminado), se garantiza que la salida tendrá todos los caracteres que no sean ASCII entrantes escapados. si asegurar_ascii es falso, estos caracteres se mostrarán tal cual.

check_circular :

Si check_circular es falso (predeterminado: verdadero), se omitirá la verificación de referencia circular para los tipos de contenedor y una referencia circular dará como resultado un OverflowError (o algo peor).

allow_nan :

Si allow_nan es falso (predeterminado: verdadero), entonces será un ValueError para serializar valores flotantes fuera de rango (nan, inf, -inf) en estricto cumplimiento de la especificación JSON. Si allow_nan es verdadero, se utilizarán sus equivalentes de JavaScript (NaN, Infinity, -Infinity).

Indent:



Si la sangría - indentación - es un entero no negativo o una cadena, los elementos de la matriz JSON y los miembros del objeto se imprimirán con ese nivel de sangría. Un nivel de sangría de 0, negativo o "" solo insertará líneas nuevas. Ninguno (el valor predeterminado) selecciona la representación más compacta. El uso de una sangría de entero positivo sangra tantos espacios por nivel. Si la sangría es una cadena (como "\\t"), esa cadena se usa para sangrar cada nivel.

Separators :

Si se especifica, los separadores deben ser una tupla (item_separator, key_separator). El valor predeterminado es ("","",":") si la sangría es Ninguno y ("","",":") de lo contrario. Para obtener la representación JSON más compacta, debe especificar que se eliminen los espacios en blanco.

Default:

Si se especifica, el valor predeterminado debe ser una función que se llame para objetos que de otro modo no se pueden serializar. Debería devolver una versión codificable JSON del objeto o generar un TypeError. Si no se especifica, se genera TypeError.

sort_keys :

Si sort_keys es verdadero (predeterminado: falso), la salida de los diccionarios se ordenará por clave.

```
import json
obj_diccionario={"uno":{"inglés":"One","francés":"Un"},
                 "dos":{"inglés":"Two","francés":"Deux"},
                 "tres":{"inglés":"Three","francés":"Trois"},
                 "cuatro":{"inglés":"Four","francés":"Quatre"},
                 "cinco":{"inglés":"Five","francés":"Cinq"}
                }
with open ("borrar.json" , mode="w") as objeto_InOut:
    json.dump(obj_diccionario,objeto_InOut)

#-----
with open ("borrar.json" , mode="r") as objeto_InOut:
    nuevo=json.load(objeto_InOut)
print (f'{nuevo=}')

# salida esperada por consola
nuevo={'uno': {'inglés': 'One', 'francés': 'Un'}, 'dos': {'inglés': 'Two', 'francés': 'Deux'}, 'tres': {'inglés': 'Three', 'francés': 'Trois'}, 'cuatro': {'inglés': 'Four', 'francés': 'Quatre'}, 'cinco': {'inglés': 'Five', 'francés': 'Cinq'}}
```

Los archivos json no son binarios y se pueden abrir con muchas aplicaciones, entre ellas el navegador web

```
import json
obj_diccionario={"uno":{"inglés":"One","francés":"Un"},
                 "dos":{"inglés":"Two","francés":"Deux"},
                 "tres":{"inglés":"Three","francés":"Trois"},
```



```
"cuatro": {"inglés": "Four", "francés": "Quatre"},  
    "cinco": {"inglés": "Five", "francés": "Cinq"}  
}  
with open(f"borrar.json", mode="w",encoding='utf-8') as objeto_InOut:  
    json.dump(obj_diccionario,objeto_InOut ,ensure_ascii=False, indent=4)  
#-----  
with open ("borrar.json" , mode="r" ,encoding='utf-8') as objeto_InOut:  
    nuevo=json.load(objeto_InOut)  
print (f" {nuevo} ")  
  
# salida esperada por consola  
nuevo={'uno': {'inglés': 'One', 'francés': 'Un'}, 'dos': {'inglés': 'Two', 'francés': 'Deux'}, 'tres': {'inglés': 'Three', 'francés': 'Trois'}, 'cuatro': {'inglés': 'Four', 'francés': 'Quatre'}, 'cinco': {'inglés': 'Five', 'francés': 'Cinq'}}  
  
#en un lector de texto, block de notas o navegador web  
{  
    "uno": {  
        "inglés": "One",  
        "francés": "Un"  
    },  
    "dos": {  
        "inglés": "Two",  
        "francés": "Deux"  
    },  
    "tres": {  
        "inglés": "Three",  
        "francés": "Trois"  
    },  
    "cuatro": {  
        "inglés": "Four",  
        "francés": "Quatre"  
    },  
    "cinco": {  
        "inglés": "Five",  
        "francés": "Cinq"  
    }  
}  
.  
.
```

Tener en cuenta que, al utilizar la serialización, es recomendable tomar precauciones de seguridad, especialmente cuando se deserializan datos provenientes de fuentes no confiables. La deserialización de datos maliciosos podría conducir a vulnerabilidades de seguridad, como la ejecución de código no autorizado.



Manejo de archivos y directorios:

El Módulo os nos proporciona funcionalidades para interactuar con archivos y directorios mediante cualquier sistema operativo (Windows, Linux o MacOS).

```
import os
    ✓ os.path.isfile() ----->booleana
    ✓ os.path.isdir() ----->booleana

    ✓ os.mkdir ("carpeta_nueva")
    ✓ os.rename ("carpeta_anteriores","carpeta_despues")
    ✓ os.rmdir ("carpeta_adios")
    ✓ path_actual = os.getcwd()
    ✓ listado=os.listdir("path_actual")

    ✓ os.chdir (f"\{directorio}\\"{archivo}")
                (f"\{directorio}/{archivo}")
    ✓ os.remove (archivo)
```

if os.path.isfile("borrar.json"):

devuelve True si el archivo "borrar.json" está en el mismo directorio en donde está el archivo python que estamos ejecutando

if os.path.isdir ("imagenes"):

devuelve True si el directorio o carpeta "imagenes" está dentro del directorio en donde está el archivo python que estamos ejecutando

El curso no contempla ver toda la estructura de directorios y alias en nombre de carpetas Windows. Tenga en cuenta que el directorio usuarios no existe en Windows sino que se escriben en inglés

Directorios reales	Alias Windows
C:\Program Files	C:\Archivos de programa
C:\Program Files (x86)	C:\Archivos de programa (x86)
C:\Users	C:\Usuarios

```
os.mkdir ("carpeta_nueva")
make directory (construir carpeta)
os.rmdir ("carpeta_adios")
remove directory (remover eliminar carpeta)
```

En windows

```
path_actual='J:\\\\carpeta_principal\\\\sub_carpeta\\\\sub_sub_carpeta'
el MacOS o Linux
path_actual='home/sub_carpeta/sub_sub_carpeta'
```

La doble barra invertida \\ es debido a los caracteres de escape que se vieron anteriormente, la



primera permite que la segunda sea leída.

Tener en cuenta que a partir de las nuevas versiones de Windows empieza a ser compatible la barra invertida \ en reemplazo de la estándar /

En Linux y MacOS no hay disco.

```
os.listdir(path_actual)
```

Windows

ATTRIB [+R | -R] [+A | -A] [+S | -S] [+H | -H] [[drive:] [path] filename] [/S [/D]]

+	Activa un atributo.
-	Desactiva un atributo.
R	Atributo de sólo lectura.
A	Atributo de archivo.
S	Atributo de sistema.
H	Atributo de archivo oculto.
/S	Procesa todos los archivos en todos los directorios de una ruta específica.
/D	Procesa los directorios también.

Linux

Mencionar los atributos de archivos (ocultos o de sistema) para acceso o visualización 777,755, 655, 644, etc.

- 0: Sin permisos
 - 1: Ejecución
 - 2: Escritura
 - 3: Lectura y escritura
 - 4: Lectura
 - 5: Lectura y ejecución
 - 6: Lectura y escritura
 - 7: Lectura, escritura y ejecución
- 777

1er valor: Propietario

Es la persona que ha creado el archivo o la carpeta

2do valor: Grupo

Dentro de este parámetro definiremos el grupo de usuarios que tendrán acceso al archivo o carpeta

3er valor: Otros

Dentro de este parámetro están incluidos los usuarios particulares

MacOS

Read

Read & write

- **Leer y escribir:** Permite a los usuarios abrir la carpeta o archivo y modificarlo.
- **Solo lectura:** Permite a los usuarios abrir la carpeta o archivo, pero no modificar su contenido.
- **Solo escribir (Buzón de Entrega):** Convierte una carpeta en un buzón de entrega. Los



usuarios pueden copiar la carpeta o archivo al buzón de entrega, pero no pueden abrirlo.
Solo el propietario del buzón de entrega puede abrirlo.

- **Sin acceso:** Bloquea totalmente el acceso a la carpeta o archivo.

1er valor: Propietario

Es la persona que ha creado el archivo o la carpeta

2do valor: staff

Dentro de este parámetro definiremos el staff de usuarios que tendrán acceso al archivo o carpeta

3er valor: everyone

Dentro de este parámetro están incluidos los usuarios particulares

```
os.rename ("nombre_anterior","nombre_despues")
rename file or directory (renombrar archivo o carpeta)
path_actual = os.getcwd()
genera un string con el camino desde el disco a la carpeta actual
```

```
path_actual = os.getcwd()
print(f'{path_actual=}')
```

```
listado = os.listdir(path_actual)
genera una lista de strings los nombres de archivos y carpetas del directorio indicado
```

```
listado = os.listdir(path_actual)
print(f'{listado=}')
```

```
os.chdir (f'{directorio}\\"{sub_directorio}"')
(f'{directorio}/'{sub_directorio}")
```

La función os.chdir() permite cambiar el directorio en el que se están realizando las operaciones de archivo y directorio.

```
import os
nuevo_directorio = f'{directorio}/{sub_directorio}_nuevo'
os.chdir(nuevo_directorio)

os.remove (archivo)
```

Eliminar un archivo específico del sistema de archivos.

```
archivo_a_eliminar = "archivo.txt"
os.remove(archivo_a_eliminar)
```

```
import os
eliminar = "borrar.txt"

if os.path.isfile (eliminar):
    os.remove(eliminar)
    print("El archivo ha sido eliminado.")
else:
    print("El archivo no existe.")
```

Se pueda hacer con un try, except



IMAGENES EXCEL

Archivo Edición Ver Favoritos Herramientas Ayuda

Atrás Adelante Arriba Búsqueda Carpetas Vistas

Dirección IMAGENES EXCEL Ir

Carpetas	Nombre	Tamaño	Tipo	Fecha de modificación
My Library	Abrir_Excel.gif	9 KB	Imagen GIF	23/09/00 10:04
Annotations	Acceso_Excel.gif	1 KB	Imagen GIF	23/09/00 10:04
p_AULACIC	Aceptación_Excel.gif	1 KB	Imagen GIF	23/09/00 10:04
Página Web personal_archivos	Agregar_datos_Excel.gif	3 KB	Imagen GIF	23/09/00 10:04
patricia	Altura_fila_Excel.gif	2 KB	Imagen GIF	23/09/00 10:03
imagenes_comprimidas	Anchura_columna_Excel.gif	2 KB	Imagen GIF	23/09/00 10:03
Proyecto	Anchura_estandar_Excel.gif	2 KB	Imagen GIF	23/09/00 10:03
IMAGENES EXCEL	Autocorrección_Excel.gif	8 KB	Imagen GIF	23/09/00 10:03
Proyecto	B_Desplazamiento_Excel.gif	1 KB	Imagen GIF	23/09/00 10:02
pdf	B_Estandar_Excel.gif	2 KB	Imagen GIF	23/09/00 10:02
pruebas_zips	B_Etiquetas_Excel.gif	1 KB	Imagen GIF	23/09/00 10:02
publicidad	B_Formato_Excel.gif	2 KB	Imagen GIF	23/09/00 10:02
REPORT_HTM_archivos	B_Fórmulas_Excel.gif	1 KB	Imagen GIF	23/09/00 10:01
temarios	B_Menu_Excel.gif	1 KB	Imagen GIF	23/09/00 10:01
webtrends	B_Título_Excel.gif	1 KB	Imagen GIF	23/09/00 10:01
Mi PC	Boton_Abrir.gif	1 KB	Imagen GIF	23/09/00 10:01
Mis sitios de red	Boton_Alín_Cent.gif	1 KB	Imagen GIF	23/09/00 10:01
Papelera de reciclaje	Boton_Alín_Columnas.gif	1 KB	Imagen GIF	23/09/00 10:00

5 objeto(s) seleccionados 11,2 KB Mi PC

Windows





/bin/	Comandos binarios esenciales de usuario
/boot/	Archivos estáticos del selector de arranque
/dev/	Archivos de unidades
/etc/	Configuración de sistema de Host específico Directorios requeridos: opt, X11, sgml, xml
/home/	Directorio 'home' de usuario
/lib/	Librerías esenciales compartidas y módulos de Kernel
/media/	Punto de montaje para medios removibles
/mnt/	Punto de montaje temporal para sistemas de archivos
/opt/	Agregados de paquetes de Software y Aplicaciones
/sbin/	Binarios de sistema
/srv/	Datos para los servicios provistos por este sistema
/tmp/	Archivos temporales
/usr/	Utilidades y aplicaciones de (Multi-)usuario Jerarquía secundaria Directorios requeridos: bin, include, lib, local, sbin, share
/var/	Variables de archivo
/root/	Directorio 'home' del usuario root
/proc/	Documentación del sistema de archivos virtual del Kernel y las condiciones de los procesos en archivos de texto

Directorio raíz de toda la jerarquía de archivos del sistema
/ Jerarquía primaria

LINUXCONFIG.ORG
Traducido a español por Franco Ferrari
franco.ferrari@rinconmovil.com
http://eldebianita.hazlo-así.biz/

Linux



Módulo de 13: Manejo de tiempos y fechas:

Python tiene principalmente dos modulos, librerias o bibliotecas destinadas al manejo de tiempos y fechas

El módulo **time** y el módulo **datetime** en Python sirven para propósitos relacionados con la medición y manipulación del tiempo, pero tienen enfoques ligeramente diferentes.

El módulo **time** se centra en la medición del tiempo y proporciona funciones de bajo nivel para trabajar con el tiempo en términos de segundos, mientras que el módulo **datetime** proporciona clases más estructuradas y orientadas a la manipulación de fechas y horas.

La elección entre ellos depende de las necesidades específicas de tu aplicación.

Si solo necesitas medir el tiempo o realizar operaciones básicas de temporización, el módulo **time** puede ser suficiente. Si necesitas trabajar con fechas y horas de manera más detallada, el módulo **datetime** es más adecuado. Aunque el módulo datetime contiene algunas funciones que pueden devolver la hora actual (`datetime.now()`), su enfoque principal es en las clases para representar fechas y horas.

En resumen,

Módulo datetime:

Medición y Representación de Fecha y Hora:

El módulo datetime proporciona clases, principalmente `datetime` y `date`, para representar fechas y horas de una manera más estructurada.

La clase `datetime` es particularmente útil ya que combina información de fecha y hora en un solo objeto.

Manipulación de Fechas y Horas:

Las clases en el módulo datetime permiten realizar operaciones más avanzadas, como sumar o restar intervalos de tiempo, formatear fechas y horas según diferentes formatos, y comparar fechas.

Módulo time:

Medición de Tiempo:

El módulo `time` proporciona funciones para medir el tiempo en términos de segundos desde una fecha de referencia específica (época).

Algunas funciones comunes incluyen `time.time()` que devuelve el tiempo actual en segundos desde la época, y `time.sleep(segundos)` que pausa la ejecución del programa durante un número específico de segundos.

Representación de Tiempo como Tuplas:

Las funciones en el módulo `time` a menudo devuelven resultados como tuplas que representan diferentes componentes de la fecha y hora (año, mes, día, etc.).



Módulo datetime:

Medición y Representación de Fecha y Hora:

En Python, datetime es un módulo que proporciona clases para trabajar con fechas y horas. En particular, la clase datetime dentro de este módulo se utiliza para representar objetos que combinan información de fecha y hora.

Para utilizar la clase datetime, primero necesitas importarla

- datetime
- .year,.month,.day,.hour,.minute,.second:
- strftime
- strptime
- Timedelta
- Time
- Date



Manejo de fechas y horas: datetime, date, time, etc:

Manejo fechas y horas

El Módulo datetime trabaja con fechas y horas, sobre un formato ISO 8601. (International Organization for Standardization). Esto permite enviar y recibir fechas y horarios entre varios programas, lenguajes, bases de datos, páginas web, etc

Datetime genera un objeto datetime con varios métodos y atributos que permiten sacar la diferencia entre dos objetos datetime (timedelta) y transformar números o string a datetime o viceversa. Además permite manipular el formato de entrada y salida ubicando año,mes,día(estándar ISO) o día,mes,año (estándar español) o mes,día,año (estándar inglés)

Métodos y atributos:	
.datetime() .now() .timedelta() .date() .fromtimestamp() .time() .strptime() .utcnow() .today() .utcfromtimestamp()	.tzinfo() .MAXYEAR .strftime() .MINYEAR .tzname() .timezone() .combine() .isoformat() .replace() .html()

```
from datetime import datetime
# Obtener la fecha y hora actual
print(f"Fecha y hora actual: { datetime.now() }")

# Crear un objeto datetime específico
print(f"Fecha y hora personalizada: {datetime(2050, 12, 31, 23, 59, 59)}")

# salida por consola
Fecha y hora actual: 2024-06-15 18:18:18.999999
Fecha y hora personalizada: 2050-12-31 23:59:59
```

ISO (Organización Internacional de Normalización) es una organización internacional independiente que desarrolla y promueve estándares en diferentes áreas. En el contexto de la gestión de fechas y horas, la norma ISO 8601 es un estándar internacional que establece un formato para la representación de fechas, horas y combinaciones de ambas.

El formato ISO 8601 se utiliza ampliamente en todo el mundo y tiene varias ventajas. Algunas de las características clave del formato ISO 8601 son:

1. Orden lógico: La fecha se representa en el formato "año-mes-día", lo cual tiene sentido tanto desde una perspectiva lógica como de ordenamiento.
2. Separadores: Se utilizan guiones (-) como separadores entre los componentes de la fecha y dos puntos (:) para separar las partes de la hora.
3. Precisión: El formato permite representar fechas y horas con diferentes niveles de precisión, desde años y meses hasta segundos y fracciones de segundo.
4. Formato extendido: Además del formato básico, el estándar ISO 8601 también define un



formato extendido que permite incluir información adicional, como la zona horaria.

5. Consistencia internacional: Al ser un estándar internacional ampliamente adoptado, el formato ISO 8601 facilita la comunicación y el intercambio de información entre diferentes sistemas y países.

Un ejemplo de una fecha y hora en formato ISO 8601 sería "2023-06-22T21:30:45.123456", donde "2023-06-22" representa la fecha en formato "año-mes-día" y "21:30:45.123456" representa la hora en formato "hora:minuto:segundo. fracción de segundo".

El uso del formato ISO 8601 es recomendado en diversas aplicaciones y lenguajes de programación, ya que proporciona una representación clara y consistente de las fechas y horas, evitando ambigüedades y facilitando el procesamiento y la interoperatividad de la información temporal.

```
import datetime
# Obtener la fecha y hora actual
fecha_hora_actual = datetime.datetime.now()
#
# Obtener la fecha actual en formato ISO
fecha_actual_iso = fecha_hora_actual.date().isoformat()
print("\tFecha actual (ISO):{ fecha_actual_iso }")
#
# Obtener la hora actual en formato ISO
hora_actual_iso = fecha_hora_actual.time().isoformat()
print("\tHora actual (ISO):{ hora_actual_iso }")
#
# Obtener la fecha y hora actual en formato ISO
fecha_hora_actual_iso = fecha_hora_actual.isoformat()
print("\tFecha y hora actual (ISO):{ fecha_hora_actual_iso }")

#salida por consola
    Fecha actual (ISO):2024-01-27
    Hora actual (ISO):11:41:42.898265
    Fecha y hora actual (ISO):2024-01-27T11:41:42.898265
```

Fecha actual (ISO):	2050-12-31		
YYYY	año x 4 caracteres,	-	guion opcional
MM	mes x 2 caracteres,	-	guion opcional
DD	día x 2 caracteres		

YYYY	año x 4 caracteres,	-	guion opcional
MM	mes x 2 caracteres,	-	guion opcional
DD	día x 2 caracteres		

Hora actual (ISO):	23:59:59.599999		
HH	hora x 4 caracteres,	:	dos puntos
MM	minutos x 2 caracteres,	:	dos puntos
SS	segundos x 2 caracteres	.	punto
segundo	1/1000000 segundos x 6 caracteres		fracción de segundo

HH	hora x 4 caracteres,	:	dos puntos
MM	minutos x 2 caracteres,	:	dos puntos
SS	segundos x 2 caracteres	.	punto
segundo	1/1000000 segundos x 6 caracteres		fracción de segundo

Fecha actual (ISO):	1969.08.20 02: 56:00.000000		
YYYY	año x 4 caracteres,	-	guion opcional
MM	mes x 2 caracteres,	-	guion opcional
DD	día x 2 caracteres		

YYYY	año x 4 caracteres,	-	guion opcional
MM	mes x 2 caracteres,	-	guion opcional
DD	día x 2 caracteres		



HH	hora x 4 caracteres,	T	Carácter T
MM	minutos x 2 caracteres,	:	dos puntos
SS	segundos x 2 caracteres	:	punto
segundo	1/1000000 segundos x 6 caracteres		fracción de segundo

```
Fecha actual (ISO): YYYY-MM-DD
Hora actual (ISO): HH:MM:SS.123456
Fecha y hora actual (ISO): YYYY-MM-DDT HH:MM:SS.123456
```

Con datetime.datetime.now() obtener la fecha y hora actual.

El método date() obtendrás solo la fecha

El método time() obtendrás solo el horario.

Fecha específica utilizando la clase datetime.date y pasando los valores de año, mes y día como argumentos.

Date:

```
import datetime
year = 1492
month = 10
day = 12
# Colón visualiza América (aunque creía que era india)
fecha_especifica = datetime.date(year, month, day)
print("\t Colón visualiza América:", fecha_especifica)

#salida por consola
Colón visualiza América: 1492-10-12
```

Datetime:

```
import datetime
year = 1969
month = 8
day = 20
hour = 2
minute = 56
second = 0
fecha_y_horario_especifico = datetime.datetime(year, month, day, hour, minute, second)
print("""\t El comandante Armstrong fue el primer ser humano que pisó la superficie del satélite
terrestre al sur del Mar de la Tranquilidad (Mare Tranquillitatis) el {fecha_y_horario_especifico }
""")

#salida por consola
El comandante Armstrong fue el primer ser humano que pisó la superficie del satélite
terrestre al sur del Mar de la Tranquilidad (Mare Tranquillitatis) el 1969-08-20 02:56:00
```

Obtener componentes individuales de una fecha/hora:

Puedes acceder a los componentes individuales de una fecha/hora, como el año, mes, día, hora, minuto y segundo utilizando los atributos de los objetos datetime.



.year,month,day,,hour..minute,,second:

```
import datetime #1969.08.20 T 02: 56:00
year = 1969
month = 8
day = 20
hour = 2
minute = 56
second = 0
hora = datetime.datetime (year, month, day ,hour,minute,second)
print("\tFecha desde el primer alunizaje :")
print("\tAño:", fecha_hora.year)
print("\tMes:", fecha_hora.month)
print("\tdía:", fecha_hora.day)
print("\tHora:", fecha_hora.hour)
print("\tMinuto:", fecha_hora.minute)
print("\tSegundo:", fecha_hora.second)
```

#salida por consola

Fecha desde el primer alunizaje :

Año: 1969

Mes: 8

día: 20

Hora: 2

Minuto: 56

Segundo: 0

La siguiente tabla muestra todos los códigos de formato que puede utilizar.

Directiva	Sentido	Ejemplo
%a	Nombre abreviado del día de la semana.	sun, mon,...sat
%A	Nombre completo del día de la semana.	sunday, monday....saturday
%w	día de la semana como un número decimal.	0, 1, ..., 6
%d	día del mes como dos números decimales con ceros.	01, 02, ...,31
%-d	día del mes como un/dos números decimales.	1, 2, ..., 31
%b	Nombre del mes abreviado.	ene, feb, ...,dic
%B	Nombre del mes completo.	Enero febrero, ...
%m	Mes como dos números decimales con ceros.	01, 02, ..., 12
%-m	Mes como un/dos números decimales.	1, 2, ..., 12
%y	Año sin siglo como dos números decimales con ceros.	00, 01, ..., 99
%-y	Año sin siglo como un/dos números decimales.	0, 1, ..., 99
%Y	Año con siglo como cuatro números decimales.	1999,2000,2025
%j	día del año como tres números decimales con ceros.	001, 002, ..., 366
%-j	día del año como uno a tres números decimales.	1, 2, ..., 366
%U	Número de semana del año (domingo como primer día de la semana). Todos los días de un año que preceden al primer domingo se consideran en la semana	0.00, 01, ..., 53
%W	Número de semana del año (lunes como primer día de la	



	semana). Todos los días de un año que preceden al primer lunes se consideran en la semana	0.00, 01, ..., 53
%H	Hora como un número decimal con ceros.	00, 01, ..., 23
%-H	Hora (0/23-24)como un/dos números decimales.	0, 1, ..., 23
%I	Hora (0/12) como un/dos números decimales con ceros.	01, 02, ..., 12
%-I	Hora (reloj de 12 horas) como un número decimal.	1, 2, ... 12
%p	AM o PM de la hora.	AM PM
%M	Minuto como dos números decimales con ceros.	00, 01, ..., 59
%-M	Minuto como un/dos números decimales.	0, 1, ..., 59
%S	Segundo como dos números decimales con ceros.	00, 01, ..., 59
%-S	Segundo como un/dos números decimales.	0, 1, ..., 59
%f	Micro segundo como números decimales, ceros a la izquierda.	000000 – 999999
%z	Desplazamiento UTC en la forma +HHMM o -HHMM.	
%Z	Nombre de la zona horaria.	Lun 30 sep 07:06:05 2013
%c	Representación de fecha y hora adecuada de la configuración regional.	
%x	Representación de fecha adecuada de la configuración reg.	09/30/13
%X	Representación de tiempo adecuada de la configuración reg.	07:06:05

strftime:

Permite que un objeto datetime (fecha/horario) sea cambiado a un objeto string.

Nota: se necesita proporcionar el formato (la estructura y orden) de salida de los argumentos en el string

Tomamos el objeto datetime y lo pasamos con **strftime** a string con el formato "%Y-%m-%d %H:%M:%S":

%Y	año 4 caracteres numéricos	- separador guion
%m	mes 2 caracteres numéricos	- separador guion
%m	día 2 caracteres numéricos	' ' separador espacio
%H	hora 2 caracteres numéricos	:
%M	minuto 2 caracteres numéricos	:
%S	Segundo 2 caracteres numéricos	:
		*separador dos puntos
		(depende de la versión)

import datetime

```
fecha_horario = datetime.datetime.now()
fecha_string = fecha_horario.strftime("%Y-%m-%d %H:%M:%S")
print("\tFecha string formateada: {fecha_string} \n{type(fecha_string)=} " )
#salida por consola
    Fecha string formateada: 2060-06-01 01:01:01
type(fecha_string)=<class 'str'>
```

import datetime

```
# Asigna formato de ejemplo1
```



```

formato1 = '%a %d %b %Y %H:%M:%S'

# Asigna formato de ejemplo2
formato2 = '%d-%m-%y %I:%m %p'
hoy = datetime.date.today() # Asigna fecha-hora

# Muestra fecha-hora según ISO 8601
print("\tFecha en formato ISO 8601:", hoy)

# Aplica formato ejemplo1
cadena1 = hoy.strftime(formato1)

# Aplica formato ejemplo2
cadena2 = hoy.strftime(formato2)

# Muestra fecha-hora según ejemplo1
print("\tFormato1: '%a %d %b %Y %H:%M:%S'", cadena1)

# Muestra fecha-hora según ejemplo2
print("\tFormato2: '%d-%m-%y %I:%m %p'", cadena2)

```

#salida por consola

```

Fecha en formato ISO 8601: 2060-06-01
Formato1: '%a %d %b %Y %H:%M:%S' Fri 01 Jun 2023 01:01:01
Formato2: '%d-%m-%y %I:%m %p' 01-06-50 01:01 AM

```

strptime:

Permite que un objeto string sea cambiado a un objeto datetime (fecha/horario).

Nota: se necesita proporcionar el formato (la estructura y orden) de los argumentos en el string para analizar cadena de caracteres y si es posible convertirla en un datetime

La sintaxis de strptime() para convertir una cadena a objeto datetime es la siguiente:

```

from datetime import datetime
fecha_string = "2050-06-01 12:30:56"
formato = "%Y-%m-%d %H:%M:%S"
objeto_datetime = datetime.strptime(fecha_string, formato)
print("\t{objeto_datetime} %Y-%m-%d %H:%M:%S,\n {type(objeto_datetime)=}")

```

salida por consola

Para convertir una cadena a objeto datetime

```

2050-06-01 12:30:56 %Y-%m-%d %H:%M:%S,
type(objeto_datetime)=<class 'datetime.datetime'>

```

El string (fecha_string) que representa la fecha y hora, y formato es otro string que especifica el formato de la fecha_string (fecha de entrada).

El Módulo datetime ofrece muchas más funcionalidades, como comparar fechas, obtener la



diferencia entre dos fechas, trabajar con zonas horarias, etc.

Timedelta:

Para sumar o restar días, semanas, meses o años con objetos de clase datetime.

```
import datetime
fecha_actual = datetime.date.today()
un_día = datetime.timedelta(days=1)
fecha_mañana = fecha_actual + un_día
fecha_pasada = fecha_actual - un_día

print("\tFecha actual:", fecha_actual)
print("\tFecha mañana:", fecha_mañana)
print("\tFecha pasada:", fecha_pasada)

#salida por consola
    Fecha actual: 2050-01-01
    Fecha mañana: 2050-01-02
    Fecha pasada: 2049-12-31
```

```
import datetime
year = 1969
month = 8
day = 20
fecha_hora = datetime.date (year, month, day )
fecha_actual = datetime.date.today()
delta=(fecha_actual-fecha_hora)
print("\tFecha desde el primer alunizaje a hoy :", delta)

#salida por consola
Fecha desde el primer alunizaje a hoy : 999999 days, 0:00:00
```

```
from datetime import timedelta, date
print("\t{(date.today() - timedelta(10))=}")

#salida por consola
(date.today() - timedelta(10))=datetime.date(2050,06, 20)
```

```
from datetime import date
def days_diff(start, end):
    return (end - start).days
print("\t{date(2000, 1, 1)} - {date(2020, 10, 28)} = {days_diff(date(2020, 1, 1), date(2020, 10, 28))} días"

#salida por consola
2000-01-01 - 2020-10-28 = 301 días
```



```
from datetime import timedelta
t1 = timedelta(weeks = 2, days = 5, hours = 1, seconds = 33)
t2 = timedelta(days = 4, hours = 11, minutes = 4, seconds = 54)
days_diff = t1 - t2
print("\t{days_diff=}")

#salida por consola
    days_diff =datetime.timedelta(days=14, seconds=50139)
```

```
from datetime import timedelta
t1 = timedelta(seconds = 33)
t2 = timedelta(seconds = 54)
t3 = t1 - t2
print("\t{days_diff=}")
print("\t{abs(days_diff)=}")

#salida por consola
    days_diff =datetime.timedelta(days=-1, seconds=86379)
    abs(days_diff) =datetime.timedelta(seconds=21)
```

Time:	Función con descripción
time.altzone	El desplazamiento de la zona horaria DST local, en segundos al oeste de UTC, si se define uno. Esto es negativo si la zona horaria DST local está al este de UTC (como en Europa occidental, incluido el Reino Unido). Solo use esto si la luz del día es distinta de cero.
time.asctime ([tupletime])	Acepta una tupla de tiempo y devuelve una cadena legible de 24 caracteres como 'Tue Dec 11 18:07:14 2008'.
time.clock()	Devuelve el tiempo de CPU actual como un número de segundos de coma flotante. Para medir los costos computacionales de diferentes enfoques, el valor de time.clock es más útil que el de time.time().
time.ctime ([segs])	Como asctime(localtime(secs)) y sin argumentos es como asctime()
time.gmtime ([segs])	Acepta un instante expresado en segundos desde la época y devuelve una tupla de tiempo t con la hora UTC. Nota: t.tm_isdst siempre es 0
time.localtime ([segs])	Acepta un instante expresado en segundos desde la época y devuelve una tupla de tiempo t con la hora local (t.tm_isdst es 0 o 1, dependiendo de si DST se aplica a segundos instantáneos por reglas locales).
time.mktime (tupletime)	Acepta un instante expresado como una tupla de tiempo en hora local y devuelve un valor de coma flotante con el instante expresado en segundos desde la época.
time.sleep (seg)	Suspende el hilo de llamada durante seg segundos.
time.strptime (fmt[,tupletime])	Acepta un instante expresado como una tupla de tiempo en la hora local y devuelve una cadena que representa el instante especificado por la cadena fmt.
time.strptime (str,fmt= "%a %b %d %H:%M:%S %Y")	Analiza str de acuerdo con la cadena de formato fmt y devuelve el instante en formato de tupla de tiempo.



time.tiempo()	Devuelve el instante de tiempo actual, un número de segundos en punto flotante desde la época.
time.tzset()	<p>Restablece las reglas de conversión de tiempo utilizadas por las rutinas de la biblioteca. La variable de entorno TZ especifica cómo se hace esto.</p> <p>0 <= horas < 24 0 <= minutos < 60 0 <= segundos < 60 0 <= microsegundos < 1000000</p>

```
from datetime import time
mintime = time.min
print("\tMin Time supported", mintime)
maxtime = time.max
print("\tMax Time supported", maxtime)

#salida por consola
Min Time supported 00:00:00
Max Time supported 23:59:59.999999
```

```
from datetime import time
hora = time(23,59,59,9999)
print("\tHora:", hora.hour)
print("\tMinutos:", hora.minute)
print("\tSegundos:", hora.second)
print("\tMicrosegundos:", hora.microsecond)

#salida por consola
Hora: 23
Minutos: 59
Segundos: 59
Microsegundos: 9999
```

```
from datetime import time
hora = time(23,59,59,9999)

# convierto la hora de data a string
string = hora.isoformat()
print("\tformato texto iso", string)
print("\t{type(string)=}")

print("\thora Original:", hora)
hora = hora.replace(hour = 13, second = 12)
print("\tNew Time:", hora)

print ("ver archivo strftime y strptime")
Ftime = hora.strftime("%I:%M %p")
```



```
print("\tFormatted time", Ftime)
#salida por consola
formato texto iso 23:59:59.009999
type(string)=<class 'str'>
    hora Original: 23:59:59.009999
    New Time: 13:59:12.009999
ver archivo strftime y strptime
    Formatted time 01:59 PM
```

Date:

```
from datetime import date
mindate = date.min
print("\tMin fecha soportado", mindate)
maxdate = date.max
print("\tMax fecha soportado", maxdate)
#salida por consola
    Min Time supported 00:00:00
    Max Time supported 23:59:59.999999
```

```
from datetime import date
fecha = date(2025, 12, 31)
print("\tAño:", fecha.year)
print("\tMes:", fecha.month)
print("\tdía:", fecha.day)
#salida por consola
    Año: 2025
    Mes: 12
    día: 31
```

```
from datetime import date
hoy = date.today()
print("\thoy", hoy)
# convierto la fecha de date a string
string = date.isoformat(hoy)
print("\tformato texto iso", string)
print("{type(string)=}")
print("\tdía semana usando weekday():", hoy.weekday())
print("\tdía semana usando isoweekday():", hoy.isoweekday())
# proleptic Gregorian ordinal
print("\tproleptic Gregorian ordinal:", hoy.toordinal())
# date from the ordinal
print("\tDate desde ordinal", date.fromordinal(720521))
#salida por consola
    hoy 2023-06-23
```



```

formato texto iso 2023-06-23
type(string)=<class 'str'>
    día semana usando weekday(): 4
    día semana usando isoweekday(): 5
    proleptic Gregorian ordinal: 738694
    Date desde ordinal 1973-09-21

```

Se que existe cierta confusión entre los módulos datetime y time, y también las clases datetime y time dentro del módulo datetime

Módulo time vs. Clase time dentro de datetime:

El módulo time es un módulo estándar en Python que proporciona funciones relacionadas con el tiempo, incluyendo operaciones de temporización y manejo de estructuras de tiempo.

La clase time dentro del módulo datetime es diferente y está diseñada para representar una hora específica del día (sin fecha). Esta clase se encuentra en el módulo datetime y no debe confundirse con el módulo time.

Módulo time vs. Clase time dentro de datetime:

El módulo datetime proporciona clases para trabajar con fechas y horas. Contiene tanto la clase datetime como la clase time (para representar horas específicas).

La clase datetime dentro del módulo datetime se utiliza para representar objetos que combinan información de fecha y hora.

Módulo time:

Medición de Tiempo:

El módulo time en Python proporciona funciones para trabajar con el tiempo, principalmente enfocándose en operaciones de bajo nivel relacionadas con la medición del tiempo y el manejo de estructuras de tiempo.

```

import time
tiempo_actual = time.time()
print(f"Tiempo actual en segundos desde la época: {tiempo_actual}")

#salida por consola
Tiempo actual en segundos desde la época: 1706564947.8344257

```

```

import time
print("Inicio de la pausa.")
time.sleep(3) # Pausa la ejecución del programa durante 3 segundos.
print("Fin de la pausa.")

#salida por consola
Inicio de la pausa.
Fin de la pausa.

```



```
import time
estructura_tiempo_actual = time.localtime()
print(f"Estructura de tiempo actual: {estructura_tiempo_actual}")

#salida por consola
Estructura de tiempo actual: time.struct_time(tm_year=2024, tm_mon=1, tm_mday=29,
tm_hour=18, tm_min=49, tm_sec=10, tm_wday=0, tm_yday=29, tm_isdst=0)
```

```
import time
tiempo_actual = time.localtime()
cadena_formateada = time.strftime("%Y-%m-%d %H:%M:%S", tiempo_actual)
print(f"Tiempo formateado como cadena: {cadena_formateada}")

#salida por consola
Tiempo formateado como cadena: 2024-01-29 18:49:10
```

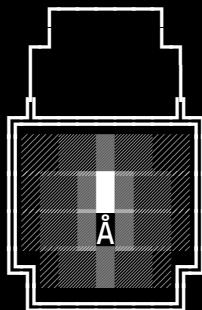


Módulo de 15: Graphical User Interface – GUI:

En Python existen varios frameworks y bibliotecas gráficas para desarrollar interfaces de usuario

Salida esperada por consola "CLI" Command Line Interface.

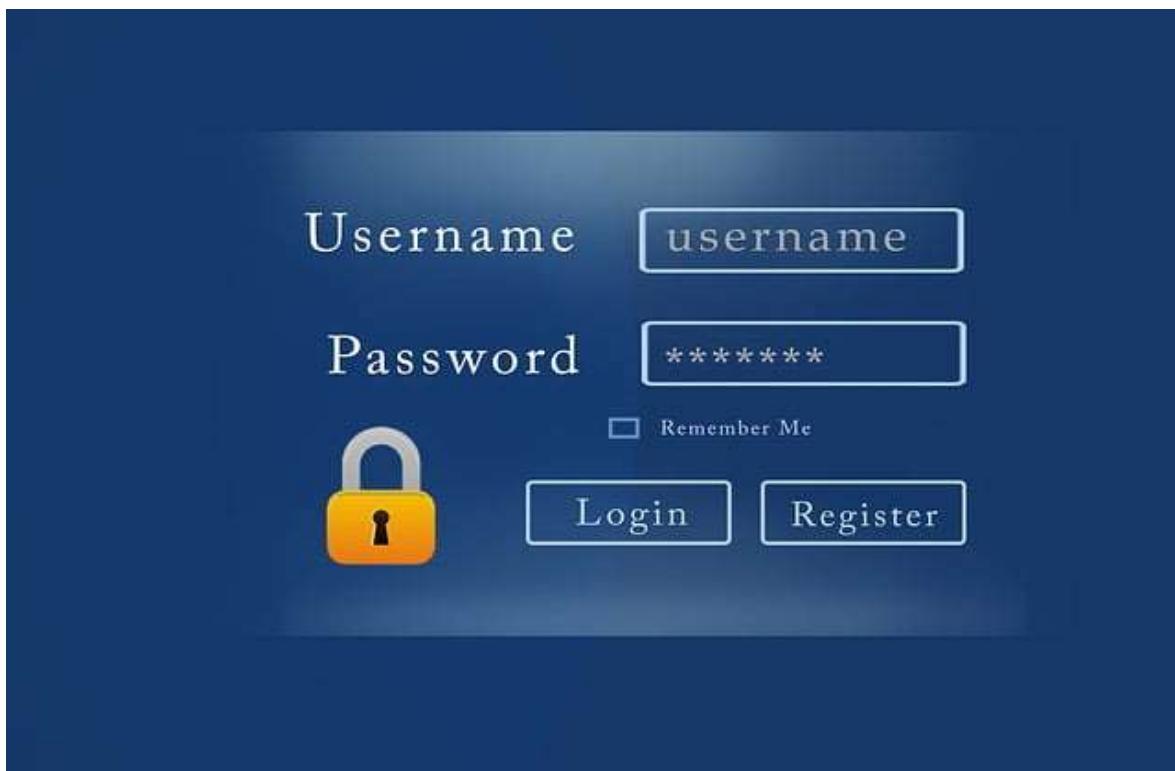
Username:
Password:
Remember Me (T/F):



Select:

- 1) Login
- 2) Register

¿Te parece que hay diferencia?



Salida esperada por "GUI" Graphical user interface.

Algunos ejemplos, y hay muchos más frameworks y bibliotecas que se adaptan a diferentes necesidades y estilos de desarrollo de GUI en Python.

La elección del framework dependerá de los requisitos específicos de tu proyecto y de tus



preferencias personales.

Tkinter es la biblioteca estándar de Python para la creación de interfaces gráficas de usuario. Proporciona una interfaz simple y fácil de usar que incluye una variedad de widgets para construir aplicaciones GUI. Viene incluida en la distribución básica de Python.

PyQt es una biblioteca que proporciona enlaces Python para las bibliotecas Qt. Qt es un conjunto completo de herramientas para el desarrollo de GUI y es conocido por su versatilidad y apariencia nativa en varias plataformas.

Kivy es un framework de código abierto para el desarrollo de aplicaciones multi-touch. Es conocido por ser multiplataforma (Windows, macOS, Linux, Android, iOS).

Multi-touch o dos ratines permiten usar dos dedos en una pantalla táctil como una tablet o celular

wxPython es una interfaz Python para wxWidgets, que es un conjunto de bibliotecas C++ para desarrollo de GUI. Proporciona una interfaz nativa para varias plataformas.

GTK (PyGObject) es una interfaz para GTK (GIMP Toolkit) y proporciona herramientas para crear interfaces gráficas con el popular kit de herramientas GTK.

FLTK (PyFLTK) es una interfaz para FLTK (Fast, Light Toolkit), una biblioteca gráfica C++. Proporciona una interfaz para Python para desarrollar aplicaciones gráficas.

Dear PyGui es una biblioteca gráfica de código abierto y fácil de usar para Python. Está diseñada para ser simple y rápida, y se basa en OpenGL.

Enaml es un framework de interfaz de usuario declarativo para Python. Permite definir interfaces gráficas utilizando una sintaxis declarativa similar a la de QML.

PySide es otro conjunto de bibliotecas para el desarrollo de aplicaciones con Qt. Es una alternativa a PyQt y proporciona una interfaz Python para las bibliotecas Qt.

Remi es un framework de interfaz de usuario para Python que permite crear aplicaciones web interactivas utilizando componentes Python puros.

Conocimientos básicos:

Objetos Gráficos: "Widgets"

- Los elementos gráficos en una GUI se llaman "widgets".

Características de los Widgets:

- Cada tipo de widget tiene diferentes métodos y atributos.
- Los widgets son objetos interactivos que permiten la entrada del usuario, como botones, cuadros de texto y etiquetas.

Posicionamiento de Widgets:

- A diferencia de la consola, en una interfaz gráfica, los widgets deben ser posicionados en la ventana.
- El flujo del programa no dicta necesariamente el orden de la secuencia de los widgets como en la consola.
- Es posible posicionar el último objeto al principio de la ventana

Interactividad y Eventos



- Los widgets responden a eventos del usuario, como clics de ratón o pulsaciones de teclas.
- Pueden asociarse funciones (manejadores de eventos) a estos eventos para realizar acciones específicas.

Estilos y Temas:

- Los widgets pueden personalizarse mediante estilos y temas para adaptarse al diseño general de la aplicación.
- La apariencia y el comportamiento de los widgets pueden modificarse según las necesidades del usuario.

Administración de Diseño en tkinter:

Tkinter ofrece administradores de diseño que facilitan la organización de widgets en la ventana, como pack, grid, y place.

método pack:

pack organiza los widgets en bloques antes de colocarlos en el contenedor.

Los widgets se empaquetan en un lado del contenedor y se expanden según sea necesario.

Uso Típico:

Se utiliza para organizar widgets en diseños sencillos, simples y rápidos.

No proporciona un control detallado sobre la ubicación de los widgets.

Los widgets se colocan uno debajo del otro o uno al lado del otro, según la dirección especificada (vertical u horizontal).

Los widgets se ajustan automáticamente al tamaño del contenido y al tamaño de la ventana.

```
boton1.pack(side="left")
boton2.pack(side="right")
```

método grid:

grid organiza los widgets en una cuadrícula.

Los widgets se colocan en filas y columnas específicas de una cuadrícula.

Uso Típico:

Se utiliza para organizar widgets en una disposición de cuadrícula más compleja. Similar a una tabla (similar a una hoja de cálculo).

Es útil para diseños más complejos y estructurados.

Puedes controlar el ancho y alto de las celdas en la cuadrícula.

Es especialmente útil para diseños más complejos en los que necesitas organizar los widgets en una estructura tabular.

```
etiqueta1.grid(row=0, column=0)
etiqueta2.grid(row=0, column=1)
```

método place:

place permite colocar widgets en ubicaciones específicas utilizando coordenadas x e y.

Uso Típico:

Se utiliza cuando se requiere un control preciso sobre la geometría, (posición y tamaño) de los widgets.



Puedes especificar manualmente la ubicación exacta de un widget utilizando coordenadas X e Y. (una batalla naval)

Puede ser más adecuado para casos en los que se necesita un control total sobre la ubicación de los widgets.

Sin embargo, es importante tener en cuenta que el uso excesivo de place() puede hacer que la interfaz gráfica sea menos flexible y menos adaptable a diferentes tamaños de ventana o cambios en el contenido.

```
boton.place(x=50, y=30)
```

En general, la elección entre pack, grid, y place dependerá de la complejidad del diseño de la interfaz y de los requisitos específicos de posicionamiento y expansión de los widgets en la aplicación. Muchas veces, se combinan estos métodos según las necesidades del diseño.

Graphical User Interface:

No podemos comparar lo que es una presentación gráfica de una en consola. La(s) ventana(s) se conoce como capa gráfica, esta se agrega a la lógica general.

Tengan en cuenta que la ventana GUI necesita iniciarse y al final se coloca un mainloop() que cierra el bucle de refresco de pantalla.

Dentro se configuran los widgets y su ubicación.

Si hay botones generalmente llama mediante command a una función (o método) que es llamado al clickear en el. Estos no admiten parámetros si no se usan lambdas.

En consola usábamos print para enviar información al usuario. Aquí tenemos labels o text dependiendo la cantidad de líneas del string a mostrar.

La entrada de datos en consola a hacemos con input aquí usamos mayormente entry pero tengan en cuenta que en consola se carga el dato cuando se presiona la tecla enter al final del dato introducido por el usuario.

En entry se captura el dato cuando se ejecuta el método .get(), no cuando el usuario suministra la información en la casilla de la GUI. En input se carga con enter, aquí no.

En consola, el orden de muestra según el flujo en que fue escrito.

Si hay un print, un input y otro print. El ultimo solo se verá en cuanto se haya ingresado el dato en el input. En consola no se superponen las salidas ya que una va siempre debajo de la otra.

En consola si tengo un label, un entry y otro label todos los widgets se mostrarán dentro de la ventana normalmente al mismo tiempo.

además cada widget deberá ser ubicado en una posición diferente para que no se superpongan.

Tkinter es una biblioteca estándar de Python que proporciona una interfaz de programación para crear aplicaciones de escritorio multiplataforma con una interfaz gráfica. Tk es una biblioteca gráfica + interface escritas en Tcl (Tool Command Language).

- Tkinter es nativa en la instalación estándar, no se requiere una instalación adicional.
- Es una biblioteca multiplataforma, lo que significa que las aplicaciones creadas con Tkinter pueden ejecutarse en diferentes sistemas operativos, como Windows, macOS y Linux.
- Proporciona una amplia gama de widgets y herramientas para la creación de interfaces gráficas, como botones, etiquetas, campos de entrada, menús desplegables, etc.
- Permite la personalización de la apariencia y el comportamiento de los widgets mediante propiedades y opciones.
- Es altamente integrable con otras bibliotecas y módulos de Python.



Tkinter proporciona una amplia gama de widgets predefinidos que se pueden utilizar para construir la interfaz de usuario de una aplicación.

windows o root (Ventana principal)	windows = Tk.tk() Un contenedor general para agrupar y organizar otros widgets. Creación de la ventana principal: windows = Tk()
windows.mainloop() (Ventana principal)	Es una llamada a un método que inicia el bucle principal de eventos en una aplicación Tkinter. .mainloop() es esencial para iniciar el bucle principal de eventos en una aplicación Tkinter. Este bucle es responsable de manejar la interactividad y mantener actualizada la interfaz gráfica en respuesta a las acciones del usuario. La ejecución de mainloop() mantiene la aplicación en un estado activo hasta que la ventana principal se cierra, momento en el cual el programa sale del bucle y finaliza.
Frame (Marco)	Uno o más contenedores secundarios rectangulares que se utilizan para agrupar y organizar otros widgets dentro de la ventana (windows). Utilización del widget Frame para organizar y estructurar la interfaz gráfica y permite trabajar con un conjunto de widgets a la vez.
Label (Etiqueta)	Un widget que se utiliza para mostrar texto cortos o imágenes en la interfaz. (ten en mente la función print)
Entry (Campo de entrada)	Implementación de campos de entrada para la introducción de datos por parte del usuario. (ten en mente la función input)
Button (Botón)	Incorporación de botones interactivos que se utiliza para realizar una acción al hacer clic en ellos. Normalmente llama a métodos o funciones Permite mostrar texto cortos o imágenes en su interior.
Text (Texto)	Utilización del widget Text para la visualización y manipulación de texto multilínea.
Listbox (Lista)	Un widget que muestra una lista de elementos de donde se puede seleccionar uno o varios.
Checkbutton (Casilla de verificación)	Integración de casillas de verificación mediante el widget Checkbutton que permite al usuario seleccionar una o varias opciones.
Radiobutton (Botón de opción)	Implementación de botones de opción exclusivos que permite al usuario seleccionar solo una opción de un grupo de opciones mutuamente excluyentes.
Menu (Menú)	Un widget que proporciona opciones de menú desplegable en una barra de menú.
Cada widget tiene sus métodos y atributos para personalizar su apariencia y comportamiento en la interfaz de usuario. Alguno de ellos como bg – fondo son utilizados en varios widgets diferentes	

Descripción de los atributos comunes en Tkinter:

x	Ubicación sobre el eje x – horizontal - es un valor en píxeles con respecto al borde izquierdo del campo de texto
y	Ubicación sobre el eje y – vertical
root	Una observación que se debe hacer es que los valores que tome tanto el ancho como el alto de la ventana además de ser únicamente en pixeles deberán de ser números enteros positivos. El significado del signo más y menos indica que si se le pasa un valor positivo este será contado desde la esquina superior izquierda de la pantalla, en cambio si le pasamos un valor negativo este



	<p>será tomado en cuenta partiendo como origen desde la esquina inferior derecha; este es un detalle que hay que tener muy en cuenta cuando deseamos posicionar una ventana.</p> <p>Si no le diéramos argumentos nos devolvería la medida y la posición de la ventana.</p> <p>Argumentos: "wxh+x+y"</p> <p>Donde las letras tienen este significado:</p> <ul style="list-style-type: none"> w: Ancho de la ventana en pixeles h: Alto de la ventana en pixeles x: Posición en el eje X y: Posición en el eje Y
.geometry ("WxH")	Damos el tamaño de la ventana <code>ventana.config("400x300")</code> o <code>ventana.config(width=400, height=300)</code>
.geometry ("WxH+ X+Y")	Damos el tamaño de la ventana y la ubicación de esta en la pantalla <code>ventana.config("400x300"+100+100)</code> o <code>ventana.config(width=400, height=300+100+100)</code>
height	Define la altura en líneas no en pixeles del componente.
width	Indicamos el ancho del control, no en píxeles, sino en caracteres del componente
font	Permite especificar la fuente a utilizar en el texto del componente. Se puede pasar una tupla con el nombre de la fuente, el tamaño y el estilo, por ejemplo: <code>Font("Times New Roman", 24, "bold underline")</code> .
bg	Modifica el color de fondo. Puedes indicar el color en inglés (incluyendo modificadores, como "darkgreen") o su código RGB en hexadecimal ("#aaaaaa" para blanco). En MacOS, no se puede modificar el color de fondo de los botones, pero se puede configurar el <code>highlightbackground</code> , que pinta el fondo alrededor del botón con el color especificado.
fg	Cambia el color del texto.
bd	Modifica el ancho del borde del widget.
cursor	Modifica la forma del cursor. Algunas opciones comunes son "arrow", "crosshair", "hand", "ibeam", etc.
relief	Cambia el estilo del borde del componente. flat: No se muestra ningún borde. solid: Borde sólido sunken: Borde hundido, que provoca que el elemento que encierra parezca que se encuentra por debajo del nivel de la superficie de la pantalla. raised: Borde saliente, que provoca que el elemento que encierra parezca que se encuentra por encima del nivel de la superficie de la pantalla. groove: Borde hundido, que visualmente parece que se encuentra por debajo del nivel de la superficie de la pantalla. ridge: Borde saliente, que visualmente parece que se encuentra por encima del nivel de la superficie de la pantalla. <code>etiqueta = tk.Label(root, text="Hola Mundo!!!", relief="sunken", borderwidth=5)</code>
state	permite deshabilitar el componente.

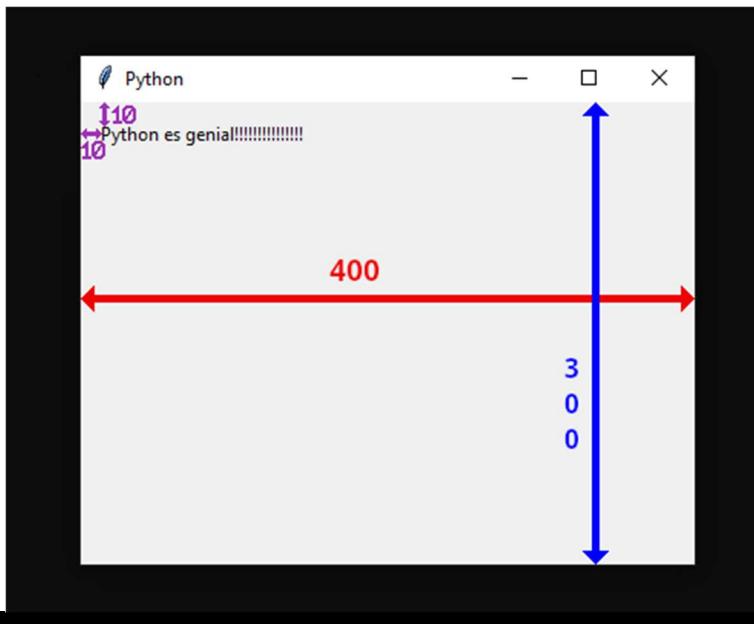


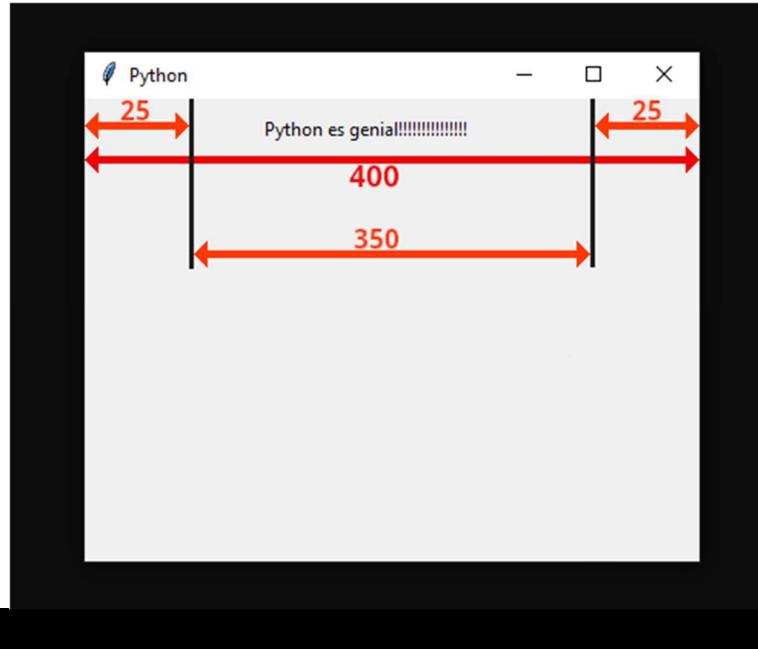
	Puede establecerse como DISABLED, por ejemplo, para una Label en la que no se pueda escribir o un Button que no se pueda hacer clic.
padding	Define el espacio en blanco alrededor del widget en píxeles.
command	Especifica la función que se ejecutará cuando se haga clic en un botón. No acepta parámetros. Para poder enviarlos se requiere una función lambda
justify	etiqueta = tk.Label(root, text="Hola\nMundo!!!", justify="right") justify="center") justify="left")

```
import tkinter as tk
ventana = tk.Tk()
ventana.config(width=400, height=300)
ventana.title("Python")

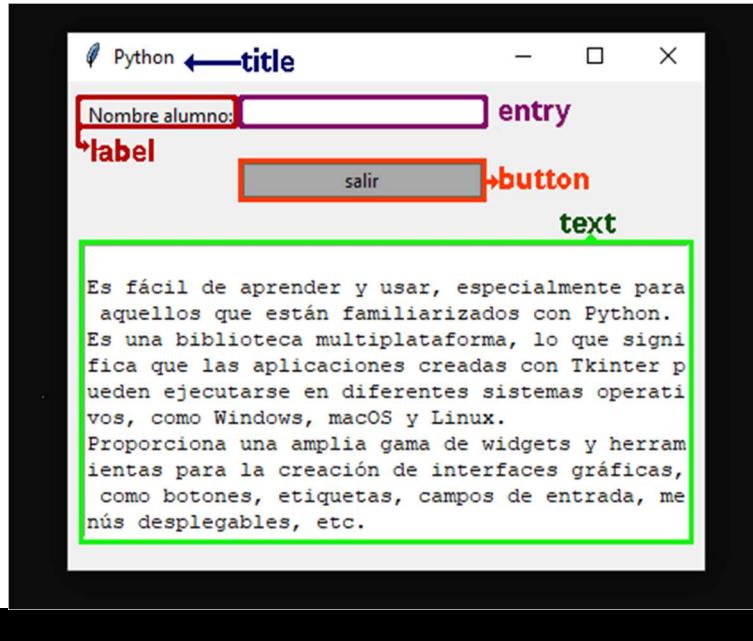
#-----
etiqueta nombre = tk.Label(text="Python es genial!!!!!!!!!!!!!! ")
etiqueta nombre.place(x=10, y=10 , width=350, height=20)

ventana.mainloop()
```





```
def salir():
    exit()
#
ventana = tk.Tk()
ventana.config(width=400, height=300)
ventana.title("Python")
#
etiqueta_nombre = tk.Label(text="Nombre alumno:")
etiqueta_nombre.place(x=10, y=10)
#
caja_nombre = tk.Entry()
caja_nombre.place(x=110, y=10, width=150, height=20)
#
boton_lista = tk.Button(text="salir", bg="#A8A8A8", command=salir)
boton_lista.place(x=110, y=50, width=150, height=20)
#
text_ = tk.Text( )
lista = f"""
Es fácil de aprender y usar, especialmente para aquellos que están familiarizados con Python.
Es una biblioteca multiplataforma, lo que significa que las aplicaciones creadas con Tkinter pueden ejecutarse en diferentes sistemas operativos, como Windows, macOS y Linux.
Proporciona una amplia gama de widgets y herramientas para la creación de interfaces gráficas, como botones, etiquetas, campos de entrada, menús desplegables, etc.
Permite la personalización de la apariencia y el comportamiento de los widgets mediante propiedades y opciones.
Es altamente integrable con otras bibliotecas y módulos de Python."""
text_.insert(tk.END, lista)
text_.place(x=10, y=100, width=380, height=180)
#
ventana.mainloop()
```



Configuración y Modificación de Atributos en Tkinter:

En Tkinter una vez creado un widget, este se puede modificar en cualquier momento.

La configuración y modificación de atributos es esencial para personalizar la apariencia y el comportamiento de los widgets en una interfaz gráfica.

Los atributos son propiedades específicas de un widget que determinan su aspecto y funcionalidad. Cada widget tiene métodos y atributos que si bien comparten nombre con los de otros (polimorfismo), el comportamiento es particular.

Al modificar atributos, especialmente en aplicaciones con muchos widgets, ten en cuenta la eficiencia. Cambios innecesarios y no utilizar actualizaciones específicas generan pantallas lentas

método config:

```
# Configuración de atributos para un Label
import tkintes as tk
label1 = tk.Label(root, text="Hola, Mundo!")#      creación
label1.config(fg="blue", font=("Arial", 12))#      configuración
label1.place(10,10)#                          ubicación
root.mainloop()
```

```
# Configuración de atributos para un Text
import tkintes as tk
texto = tk.Text(root)#                      creación
texto.tag_config("resaltado", background="yellow", foreground="red")#    configuración
texto.place(10,10)#                         ubicación
root.mainloop()
```

Actualización Dinámica de Atributos:

Los atributos pueden actualizarse dinámicamente durante la ejecución del programa en respuesta a eventos o cambios en la aplicación. Utiliza métodos como config o métodos específicos del widget



para actualizar los atributos según sea necesario.

```
import tkintes as tk
# Actualización dinámica de atributos en respuesta a un evento
def cambiar_color():# función
    label.config(fg="red")
boton = Button(root, text="Cambiar Color", command=cambiar_color)# llama a la funcion
#Si se necesita pasar parámetros debería usarse una función lambda
boton.place(10,10)# ubicación
root.mainloop()
```

Uso de Variables de Control:

Algunos atributos, como el estado de una casilla de verificación (Checkbutton), pueden estar vinculados a variables de control (BooleanVar, IntVar, etc.).

```
# Configuración de atributos vinculados a variables de control
import tkintes as tk
var = BooleanVar()
checkbox = tk.Checkbutton(root, text="Aceptar Términos y Condiciones", variable=var)
root.mainloop()
```

Manejo de Estilos y Temas:

Tkinter proporciona la capacidad de aplicar estilos y temas a los widgets para cambiar su apariencia y comportamiento. Utiliza el módulo ttk para acceder a estilos y temas adicionales.

```
import tkinter as tk
from tkinter import ttk
style = ttk.Style()
style.configure("Estilo.TButton", foreground="green", font=("Helvetica", 12))
boton = ttk.Button(root, text="Aceptar", style="Estilo.TButton")
```

La personalización de la apariencia y el comportamiento de los widgets en Tkinter es una tarea clave para crear interfaces gráficas atractivas y adaptadas a las preferencias del usuario. Una poderosa herramienta para lograr esto es a través de la utilización de estilos y temas proporcionados por el módulo ttk (themed Tkinter)ttk es un módulo adicional en Tkinter que proporciona widgets temáticos y estilos avanzados.

Creación de un Objeto de Estilo:

```
# Debes crear un objeto de estilo utilizando ttk.Style().
# Luego lo configuras
import tkinter as tk
from tkinter import ttk
estilo = ttk.Style()
# Utiliza el método configure del objeto de estilo para especificar los atributos deseados.
estilo.configure("EstiloBoton.TButton", foreground="blue", font=("Arial", 12))
```

Aplicación del Estilo a un Widget:

```
# Una vez que has configurado un estilo, puedes aplicarlo a un widget específico usando el argumento style.
```



```
boton = ttk.Button(root, text="Aceptar", style="EstiloBoton.TButton")
```

Estilos Predefinidos:

ttk ofrece estilos predefinidos que puedes aplicar directamente a los widgets.

Ejemplo de aplicación de un estilo predefinido:

```
boton = ttk.Button(root, text="Aceptar", style="TButton")
```

Uso de Temas pre-armados:

Los temas en Tkinter permiten cambiar el aspecto visual global de la interfaz.

Puedes seleccionar un tema predefinido o crear tu propio tema.

- "clam"
- "alt"
- "default"
- "classic"
- "vista"
- "xpnative"

Personalización de Temas:

Si bien puedes crear tu propio estilo, también es posible tomar uno ya existente y personalizar ciertos aspectos de un tema utilizando métodos específicos del objeto de estilo.

Personalización de la apariencia de las barras de desplazamiento en un tema:

```
estilo.theme_use("clam")
```

```
estilo.theme_element_create("miScrollbar", "hscroll", parent="horizontal.TScrollbar")
```

```
estilo.configure("Horizontal.TScrollbar", element="miScrollbar", background="red")
```

Reutilización de Estilos:

Puedes reutilizar estilos y aplicarlos a varios widgets para mantener una apariencia coherente en toda la aplicación.

Ejemplo de reutilización de un estilo:

```
etiqueta = ttk.Label(root, text="Hola, Mundo!", style="EstiloEtiqueta.TLabel")
```

Eliminación de Estilos:

Si decides cambiar o eliminar un estilo, puedes utilizar el método `ttk.Style().configure` para restablecer los atributos o `ttk.Style().theme_reset()` para restablecer el tema.

```
root.mainloop()
```

Consideraciones Adicionales:

Al personalizar estilos y temas, ten en cuenta las preferencias de usabilidad y accesibilidad.

Experimenta con diferentes combinaciones de colores, fuentes y tamaños para lograr el diseño deseado.

La utilización de estilos y temas en Tkinter permite una personalización avanzada de la interfaz gráfica, proporcionando flexibilidad y coherencia en el diseño de aplicaciones. Experimenta con diferentes configuraciones para descubrir la combinación perfecta que se ajuste a tus necesidades y a la estética de tu aplicación.

Ventanas y frames:



En Tkinter, la manipulación de ventanas y frames implica el control de la apertura, cierre y manipulación de ventanas secundarias. Aquí se presentan algunos métodos y conceptos clave para realizar estas operaciones:

Creación de Ventanas Secundarias:

Utiliza la clase Toplevel para crear ventanas secundarias en Tkinter. Estas ventanas son independientes de la ventana principal.

Creación de una ventana secundaria:

```
from tkinter import Tk, Button, Toplevel
```

```
def abrir_ventana():
```

```
    ventana_secundaria = Toplevel(root)
```

```
    ventana_secundaria.title("Ventana Secundaria")
```

```
    etiqueta = Label(ventana_secundaria, text="Contenido de la ventana secundaria")
```

```
    etiqueta.pack()
```

```
root = Tk()
```

```
boton = Button(root, text="Abrir Ventana", command=abrir_ventana)
```

```
boton.pack()
```

Cierre de Ventanas Secundarias:

Utiliza el método destroy() para cerrar una ventana. Este método destruye la ventana y libera los recursos asociados.

cierre de una ventana secundaria:

```
def cerrar_ventana(ventana):
```

```
    ventana.destroy()
```

```
# ...
```

```
boton_cerrar = Button(ventana_secundaria, text="Cerrar Ventana", command=lambda: cerrar_ventana(ventana_secundaria))
```

```
boton_cerrar.pack()
```

Manipulación de Frames en Ventanas Secundarias:

Puedes agregar frames a ventanas secundarias de la misma manera que lo haces en la ventana principal.

agregar un frame a una ventana secundaria:

```
frame_secundario = Frame(ventana_secundaria)
```

```
etiqueta_frame = Label(frame_secundario, text="Contenido del Frame en la ventana secundaria")
```

```
etiqueta_frame.pack()
```

```
frame_secundario.pack()
```

Comunicación entre Ventanas:

Para la comunicación entre ventanas, puedes pasar parámetros a funciones que crean ventanas secundarias o utilizar variables globales.

```
def abrir_ventana_con_parámetro(parámetro):
```

```
    ventana_secundaria = Toplevel(root)
```

```
    etiqueta = Label(ventana_secundaria, text=f"parámetro: {parámetro}")
```

```
    etiqueta.pack()
```

```
boton_parámetro = Button(root, text="Abrir Ventana con parámetro", command=lambda: abrir_ventana_con_parámetro("Hola"))
```

```
boton_parámetro.pack()
```



Restricción de Interacción con la Ventana Principal:

```
# Puedes deshabilitar la interacción con la ventana principal mientras se muestra una ventana secundaria utilizando el método grab_set() y restablecerla con grab_release().  
# La ventana principal no es accesible mientras la ventana modal está abierta.  
def abrir_ventana_modal():  
    ventana_modal = Toplevel(root)  
    ventana_modal.title("Ventana Modal")  
    ventana_modal.grab_set()  
boton_modal = Button(root, text="Abrir Ventana Modal", command=abrir_ventana_modal)  
boton_modal.pack()
```

Personalización de Ventanas Secundarias:

```
# Puedes personalizar la apariencia de las ventanas secundarias utilizando métodos como geometry para establecer dimensiones y attributes para configurar propiedades adicionales.  
# Se establecen dimensiones y la ventana se coloca en la parte superior de otras ventanas.
```

```
ventana_personalizada = Toplevel(root)  
ventana_personalizada.title("Ventana Personalizada")  
ventana_personalizada.geometry("300x200")  
ventana_personalizada.attributes("-topmost", True)
```

Manejo de Cierre de la Ventana Principal:

```
# Maneja el evento de cierre de la ventana principal utilizando el método protocol para realizar acciones específicas antes de cerrar la aplicación.  
#se muestra un cuadro de diálogo de confirmación antes de cerrar la aplicación.  
def cerrar_aplicacion():  
    if messagebox.askokcancel("Cerrar Aplicación", "¿Estás seguro de que quieres salir?"):  
        root.destroy()  
root.protocol("WM_DELETE_WINDOW", cerrar_aplicacion)  
root.mainloop()
```

Manejo de Imágenes y Iconos:

Algunos atributos permiten la manipulación de imágenes, iconos o fondos. Usa métodos específicos o el método config para asignar imágenes a widgets como Label o Button.

```
import tkinter as tk  
from tkinter import PhotoImage  
root = tk.Tk()  
imagen = PhotoImage(file="imagen.gif")  
label = Label(root, image=imagen)  
label.pack()  
root.mainloop()
```



Frame(Marco):

Uno o más contenedores secundarios rectangulares que se utilizan para agrupar y organizar otros widgets dentro de la ventana (windows).

Utilización del widget Frame para organizar y estructurar la interfaz gráfica y permitirte trabajar con un conjunto de widgets a la vez dentro de la ventana principal de manera lógica y ordenada.

Creación de un Frame:

```
from tkinter import tk, Frame  
root = tk.Tk()  
mi_frame = Frame(root)  
mi_frame.pack()
```

Configuración de Atributos:

```
# Puedes configurar atributos específicos del Frame, como su color de fondo, borde y tamaño.  
mi_frame.configure(bg="lightgray", bd=5, relief="ridge")
```

Ubicación del Frame:

```
# Utiliza los métodos de administración de diseño (pack, grid, place) para colocar el Frame en la  
ventana principal.
```

```
mi_frame.pack(side="top", fill="both", expand=True)
```

```
# Agrega widgets y otros elementos dentro del Frame para organizar el contenido de manera más  
detallada.
```

```
from tkinter import Button  
boton = Button(mi_frame, text="Hacer Algo")  
boton.pack()
```

Jerarquía de Frames:

```
# Puedes anidar múltiples Frames para crear una jerarquía y organizar widgets de manera más  
estructurada.
```

```
sub_frame = Frame(mi_frame)  
sub_frame.pack(side="left", padx=10)
```

Estilo y Temas en Frames:

```
# Aplica estilos y temas a los Frames para mantener una coherencia visual en toda la aplicación.
```

```
from tkinter import ttk  
from tkinter import Tk, Frame  
root = Tk()  
estilo = ttk.Style()  
estilo.theme_use("clam")  
mi_frame = ttk.Frame(root)  
mi_frame.pack()
```

Manejo de Eventos en Frames:

```
# Puedes asociar funciones y manejadores de eventos específicos a widgets dentro de un Frame.
```

```
# Estos eventos pueden ser clics de botones, entradas de teclado, entre otros.
```

```
def hacer_algo():  
    print("Botón clicado")
```

```
boton = Button(mi_frame, text="Hacer Algo", command=hacer_algo)  
boton.pack()
```



Personalización de Frames según su Función:

Dependiendo de la función del Frame, puedes personalizar su apariencia y comportamiento de manera específica.

```
from tkinter import Entry, Label, Frame  
root= tk.Tk()  
frame_entrada = Frame(root)  
Label(frame_entrada, text="Ingrese su nombre:").pack()  
Entry(frame_entrada).pack()  
frame_entrada.pack()
```

Reutilización de Frames:

Puedes crear un Frame personalizado y reutilizarlo en diferentes partes de tu aplicación.

```
def crear_frame_personalizado(padre):  
    nuevo_frame = Frame(padre, bg="lightblue", bd=2, relief="groove")  
    Label(nuevo_frame, text="Contenido 2024").pack()  
    Button(nuevo_frame, text="Haz Click").pack()  
    return nuevo_frame  
#....  
frame_1 = crear_frame_personalizado(root)  
frame_2 = crear_frame_personalizado(root)  
frame_1.pack(side="left", padx=10)  
frame_2.pack(side="right", padx=10)  
root.mainloop()
```

Consideraciones de Eficiencia:

- Evita la sobre complejidad en la estructura de Frames. Utiliza la jerarquía de manera lógica y mantén un equilibrio entre la claridad y la eficiencia.
- Ajusta los atributos según las necesidades específicas de cada Frame.

Diseño Responsivo:

- Si estás diseñando una aplicación que debe adaptarse a diferentes tamaños de ventana, considera el diseño responsivo al configurar la ubicación y el tamaño de los Frames.
- La personalización de Frames en Tkinter te proporciona un control preciso sobre la organización y estructura de tus widgets dentro de la interfaz gráfica. Experimenta con diferentes configuraciones y estilos para lograr el diseño deseado y mejorar la experiencia del usuario.



Respuesta a Eventos en Tkinter:

En Tkinter, los widgets responden a eventos del usuario, como clics de ratón o pulsaciones de teclas, mediante la asociación de funciones o manejadores de eventos específicos.

Eventos del Usuario:

Los eventos del usuario son acciones que ocurren durante la interacción con la interfaz gráfica. Algunos ejemplos comunes son clics de ratón, pulsaciones de teclas, movimientos de ratón, entre otros.

Asociación de Funciones a Eventos:

Para que un widget responda a un evento específico, se asocia una función o manejador de eventos a ese evento. La función se ejecutará cuando el evento ocurra.

Se utiliza el método bind para asociar eventos a funciones.

#Ejemplo de asociación de un clic de ratón a una función:

```
from tkinter import Tk, Button  
def manejar_clic(event):  
    print("¡Clic de ratón!")  
root = Tk()  
boton = Button(root, text="Haz clic")  
boton.bind("<Button-1>", manejar_clic)  
boton.pack()  
root.mainloop()
```

Manejo de Eventos por Defecto:

#Al asociar una función a un evento, puedes anular el comportamiento predeterminado del widget para ese evento. Esto permite personalizar la respuesta del widget.

En este caso, la función prevenir_pulsacion se ejecutará en lugar del comportamiento predeterminado de la tecla Enter en un campo de entrada.

```
from tkinter import Tk, Entry  
def prevenir_pulsacion(event):  
    print("Pulsación de tecla preventida")  
root = Tk()  
entrada = Entry(root)  
entrada.bind("<Return>", prevenir_pulsacion)  
entrada.pack()  
root.mainloop()
```

Eventos Específicos:

Cada evento tiene una cadena asociada que lo identifica de manera única. Por ejemplo, <Button-1> representa un clic izquierdo, <Button-3> un clic derecho, <KeyPress> una pulsación de tecla, entre otros.

Puedes consultar la documentación de Tkinter para obtener la sintaxis completa de eventos.

Acceso a la Información del Evento:

La función asociada a un evento puede acceder a información sobre ese evento a través del argumento event. Este objeto proporciona detalles como las coordenadas del ratón, la tecla presionada, etc.

Ejemplo de acceso a la información del evento:

En este ejemplo, la función mostrar_coordenadas imprime las coordenadas del ratón mientras se



arrastra el botón izquierdo en un Canvas.

```
from tkinter import Tk, Canvas
def mostrar_coordenadas(event):
    print(f"Coordenadas del ratón: x={event.x}, y={event.y}")
root = Tk()
canvas = Canvas(root, width=300, height=200)
canvas.bind("<B1-Motion>", mostrar_coordenadas)
canvas.pack()
```

Desvinculación de Eventos:

Puedes desvincular una función de un evento utilizando el método unbind.

Esto evita que la función se ejecute cuando ocurre el evento.

Ejemplo de desvinculación de un evento:

```
from tkinter import Tk, Button
def manejar_clic(event):
    print("¡Clic de ratón!")

root = Tk()
boton = Button(root, text="Haz clic")
boton.bind("<Button-1>", manejar_clic)
boton.pack()
root.mainloop()
```

Desvincular el evento después de un tiempo

En este ejemplo, la función manejar_clic se desvincula del evento después de 3 segundos.

```
root.after(3000, lambda: boton.unbind("<Button-1>", manejar_clic))
```

Al comprender cómo los widgets responden a eventos y cómo asociar funciones a estos eventos, puedes crear interfaces interactivas y personalizadas en Tkinter.

Experimenta con diferentes eventos y funciones para mejorar la experiencia del usuario en tus aplicaciones.

Acceso a la Entrada del Usuario en Tkinter:

En Tkinter, puedes acceder y procesar la entrada del usuario principalmente a través de widgets como Entry y Text. Aquí se proporcionan detalles sobre cómo hacerlo y se compara con la función input de Python:

Creación de un Widget Entry:

Utiliza el widget Entry para permitir que el usuario ingrese texto. Este widget crea un cuadro de entrada de una línea.

La función obtener_entrada obtiene el texto ingresado en el widget Entry cuando se hace clic en el botón.

```
from tkinter import Tk, Entry, Button
def obtener_entrada():
    texto = entrada.get()
    print(f"Texto ingresado: {texto}")
root = Tk()
```



```
entrada = Entry(root)
boton = Button(root, text="Obtener Entrada", command=obtener_entrada)
entrada.pack()
boton.pack()
root.mainloop()
```

Acceso a la Entrada del Usuario con Text:

El widget Text permite la entrada de texto en varias líneas. Puedes especificar el número de filas y columnas.

```
from tkinter import Tk, Text, Button
def obtener_texto():
    texto = texto_widget.get("1.0", "end-1c")
    print(f"Texto ingresado:\n{texto}")
root = Tk()
texto_widget = Text(root, height=5, width=30)
boton = Button(root, text="Obtener Texto", command=obtener_texto)

texto_widget.pack()
boton.pack()
root.mainloop()
```

Comparación con la Función input:

La función input de Python permite obtener una entrada de usuario directamente desde la consola.

```
texto_usuario = input("Ingrese un texto: ")
print(f"Texto ingresado: {texto_usuario}")
```

Comparación: Entry y Text en Tkinter son widgets gráficos que permiten la entrada de usuario en una interfaz gráfica, mientras que input es una función de consola.

Los widgets en Tkinter proporcionan una experiencia más visual y son ideales para interfaces gráficas, mientras que input es simple y directo en entornos de consola.

input en consola:

- Se utiliza en entornos de consola.

- La entrada se realiza directamente en la consola.

- Útil para programas simples de consola.

- El objeto recibe la información del usuario al presionar enter

Entry y Text en Tkinter:

- Se utilizan en interfaces gráficas.

- Proporcionan una experiencia más visual e interactiva.

- Más adecuados para aplicaciones con interfaces gráficas complejas.

En resumen, Entry y Text en Tkinter son ideales para la entrada de usuario en interfaces gráficas, mientras que input es más adecuado para programas simples en la consola. La elección depende del contexto y de la naturaleza de tu aplicación.

El objeto no recibe la información del usuario al presionar enter sino mediante el método get()

Actualización Dinámica de la Interfaz:

métodos para actualizar dinámicamente la interfaz gráfica en respuesta a eventos o cambios en los datos, que es esencial para proporcionar una experiencia interactiva a los usuarios.



Actualización de Texto en Label:

```
# Utiliza el método config para actualizar dinámicamente el texto en un widget Label.  
from tkinter import Tk, Label, Button  
def actualizar_texto():  
    nuevo_texto = "Nuevo Texto"  
    label.config(text=nuevo_texto)  
root = Tk()  
label = Label(root, text="Texto Inicial")  
boton = Button(root, text="Actualizar Texto", command=actualizar_texto)  
  
label.pack()  
boton.pack()  
root.mainloop()
```

Modificación de Contenido en Entry o Text:

```
# Usa el método delete y insert para modificar el contenido de un widget Entry o Text.  
from tkinter import Tk, Entry, Button  
def modificar_contenido():  
    nuevo_contenido = "Nuevo Contenido"  
    entry.delete(0, "end")  
    entry.insert(0, nuevo_contenido)  
root = Tk()  
entry = Entry(root)  
boton = Button(root, text="Modificar Contenido", command=modificar_contenido)  
entry.pack()  
boton.pack()  
root.mainloop()
```

#Actualización de Listbox o Combobox:

```
# Para listas de opciones, utiliza métodos como insert o delete para actualizar dinámicamente los  
elementos en un widget Listbox o Combobox.  
from tkinter import Tk, Listbox, Button  
def actualizar_lista():  
    nueva_lista = ["Uno", "Dos", "Tres"]  
    listbox.delete(0, "end")  
    for item in nueva_lista:  
        listbox.insert("end", item)  
root = Tk()  
listbox = Listbox(root)  
boton = Button(root, text="Actualizar Lista", command=actualizar_lista)  
listbox.pack()  
boton.pack()  
root.mainloop()
```

Cambio de Imagen en un Canvas:

```
# Si estás utilizando un Canvas, puedes cambiar dinámicamente la imagen utilizando el método  
create_image.  
from tkinter import Tk, Canvas, PhotoImage, Button  
def cambiar_imagen():  
    nueva_imagen = PhotoImage(file="nueva_imagen.png")
```



```
canvas.itemconfig(imagen_canvas, image=nueva_imagen)
root = Tk()
canvas = Canvas(root, width=200, height=200)
imagen = PhotoImage(file="imagen_inicial.png")
imagen_canvas = canvas.create_image(100, 100, anchor="center", image=imagen)
boton = Button(root, text="Cambiar Imagen", command=cambiar_imagen)
canvas.pack()
boton.pack()
root.mainloop()
```

Dinámica de Widgets con Variables Tkinter:

Usa variables Tkinter (StringVar, IntVar, etc.) para vincular dinámicamente el contenido de los widgets.

Ejemplo de uso de StringVar:

```
from tkinter import Tk, Label, Button, StringVar
def actualizar_variable():
    nueva_info = "Nueva Información"
    variable.set(nueva_info)
root = Tk()
variable = StringVar()
label = Label(root, textvariable=variable)
boton = Button(root, text="Actualizar Variable", command=actualizar_variable)
label.pack()
boton.pack()
root.mainloop()
```

Estos ejemplos proporcionan métodos básicos para la actualización dinámica de la interfaz gráfica en Tkinter.

Dependiendo de la complejidad de tu aplicación, podrías necesitar métodos más avanzados, como la actualización de widgets dentro de un Frame o la actualización de gráficos en un Canvas de manera más dinámica.

La elección de la técnica dependerá de los requisitos específicos de tu aplicación.

Diseño anclado y no anclado:

En el contexto de diseño de interfaces gráficas, el término "anclado" se refiere a la técnica de fijar o posicionar widgets (elementos gráficos como botones, etiquetas, etc.) en direcciones específicas dentro de su contenedor.

Los dos conceptos clave asociados con el diseño anclado son "anchor" y "sticky".

Ancla widgets en direcciones específicas dentro de su contenedor utilizando los argumentos anchor y sticky.

Esto ayuda a mantener una disposición consistente al cambiar el tamaño de la ventana.

Anchor (Ancla):

Este concepto se refiere a la dirección o punto alrededor del cual el widget está "anclado" dentro de su contenedor.

Por ejemplo, si anclas un widget al "norte" (north), se posicionará en la parte superior del contenedor.

Sticky (Adherente):

Este concepto está relacionado con la capacidad de un widget para "adherirse" a los bordes



del contenedor a medida que este cambia de tamaño. Puedes especificar las direcciones (norte, sur, este, oeste) a las que el widget se adhiere mientras el contenedor se redimensiona.

Con diseño no anclado, nos referirnos cuando los widgets no están fijos a direcciones específicas. Los widgets pueden fluir y ajustarse según el diseño general de la interfaz.

En un diseño no anclado, es posible que los widgets se distribuyan de manera más flexible, ocupando el espacio disponible sin necesariamente estar alineados estrictamente en direcciones específicas. Esto puede ser útil en casos en los que deseas que los elementos se adapten de manera más dinámica a cambios en el tamaño de la ventana o contenedor. (ten en cuenta que los celulares tienen todos pantallas diferentes y que además podes verlos horizontal o verticalmente).

Gestión de Frames en Tkinter:

Los frames en Tkinter son contenedores rectangulares que pueden contener otros widgets y proporcionan una forma eficiente de organizar y estructurar una interfaz gráfica dentro de una ventana.

Importancia de los Frames:

- **Organización Lógica:**

Los frames permiten agrupar widgets relacionados en áreas específicas de la interfaz.

Facilitan la organización lógica de la interfaz, mejorando la legibilidad y mantenimiento del código.

- **Modularidad:**

Al dividir la interfaz en frames, se mejora la modularidad del código.

Cada frame puede contener funcionalidades específicas, lo que facilita la gestión y actualización de la aplicación.

- **Personalización y Estilo:**

Los frames pueden personalizarse con su propio estilo y apariencia.

Se pueden aplicar colores, bordes y otras propiedades específicas a cada frame.

- **Facilita Actualizaciones y Cambios:**

Al tener áreas de la interfaz encapsuladas en frames, es más fácil realizar cambios o actualizaciones sin afectar otras partes de la aplicación.

- **Mejora el Mantenimiento:**

La división de la interfaz en frames mejora la mantenibilidad del código.

Cambios o adiciones a funcionalidades específicas se pueden realizar de manera más sencilla.

Ejemplos de Uso de Frames:

- **Estructuración de Páginas:**

Organiza la interfaz en frames que representan páginas o secciones diferentes de la aplicación.

```
from tkinter import Tk, Frame, Button, Label

def mostrar_pagina(pagina):
    # Oculta todas las páginas y muestra la seleccionada
    for p in paginas:
        paginas[p].pack_forget()
```



```
paginas[pagina].pack()
# Configura el color de fondo después de empaquetar nuevamente
root.config(bg=colores[pagina])

root = Tk()

# Tamaño y posición inicial de la ventana
root.geometry("400x300+100+100") # Ancho x Alto + Posición X + Posición Y

# Definición de Frames con colores de fondo
frame_inicio = Frame(root)
frame_configuración = Frame(root)
frame_ayuda = Frame(root)

# Colores de fondo para cada Frame
colores = {"inicio": "red", "configuracion": "green", "ayuda": "blue"}

# Contenido de cada Frame
Label(frame_inicio, text="Página de Inicio").pack()
Label(frame_configuración, text="Configuración").pack()
Label(frame_ayuda, text="Página de Ayuda").pack()

# Botones para cambiar entre páginas
Button(root, text="Inicio", command=lambda: mostrar_pagina("inicio"))
Button(root, text="Configuración", command=lambda: mostrar_pagina("configuracion"))
Button(root, text="Ayuda", command=lambda: mostrar_pagina("ayuda"))

# Diccionario que mapea nombres de páginas a sus Frames respectivos
paginas = {"inicio": frame_inicio, "configuracion": frame_configuración, "ayuda": frame_ayuda}

# Configura el color de fondo inicial
root.config(bg=colores["inicio"])

root.mainloop()
```

- Organización de Secciones:
Utiliza frames para organizar diferentes secciones de la interfaz, como encabezado, contenido y pie de página.
- Formulario en Frames:
Estructura un formulario utilizando frames para agrupar widgets relacionados.



Aplicación Práctica de Administradores de Diseño en Tkinter:

La aplicación práctica de los administradores de diseño es efectiva y estéticamente agradable.

Creación de un Formulario con Tkinter

```
from tkinter import Tk, Frame
from tkinter import Tk, Frame

def enviar_formulario():
    # lógica para procesar el formulario
    nombre = entry_nombre.get()
    apellido = entry_apellido.get()
    print(f'Nombre: {nombre}, Apellido: {apellido}')

# Creación de la ventana principal
root = Tk()
root.title("Formulario Simple")

# Creación de un Frame para el formulario
frame_formulario = Frame(root, padx=20, pady=20)

# Widgets del formulario
label_nombre = Label(frame_formulario, text="Nombre:")
entry_nombre = Entry(frame_formulario)

label_apellido = Label(frame_formulario, text="Apellido:")
entry_apellido = Entry(frame_formulario)

boton_enviar = Button(frame_formulario, text="Enviar",
                      command=enviar_formulario)

# Organización con pack
label_nombre.pack()
entry_nombre.pack()

label_apellido.pack()
entry_apellido.pack()

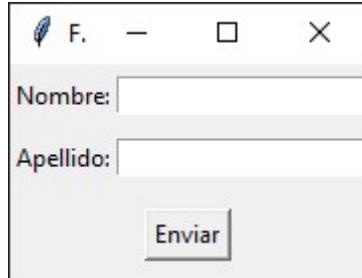
boton_enviar.pack()

# Organización del Frame en la ventana principal
frame_formulario.pack()

# Inicio del bucle principal
root.mainloop()
```



El manejo de ventanas y frames es esencial para crear interfaces de usuario.



Alineación de Widgets con grid:

En este ejemplo, utilizaremos el administrador de diseño grid para alinear widgets de manera más precisa.

```
from tkinter import Tk, Label, Entry, Button
def enviar_formulario():
    # lógica para procesar el formulario
    nombre = entry_nombre.get()
    apellido = entry_apellido.get()
    print(f"Nombre: {nombre}, Apellido: {apellido}")

    # Creación de la ventana principal
    root = Tk()
    root.title("Formulario con Grid")

    # Widgets del formulario
    label_nombre = Label(root, text="Nombre:")
    entry_nombre = Entry(root)

    label_apellido = Label(root, text="Apellido:")
    entry_apellido = Entry(root)

    boton_enviar = Button(root, text="Enviar", command=enviar_formulario)

    # Organización con grid
    label_nombre.grid(row=0, column=0, pady=5)
    entry_nombre.grid(row=0, column=1, pady=5)

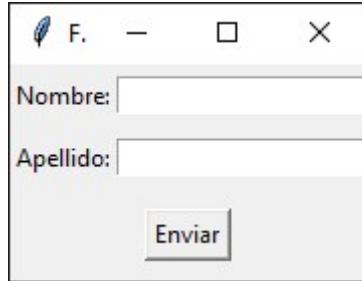
    label_apellido.grid(row=1, column=0, pady=5)
    entry_apellido.grid(row=1, column=1, pady=5)

    boton_enviar.grid(row=2, columnspan=2, pady=10)

    # Inicio del bucle principal
    root.mainloop()
```

Alineación de Widgets con place:

En este ejemplo, utilizaremos el administrador de diseño place para alinear widgets de manera más precisa en ejes x e y.



```
from tkinter import Tk, Label, Entry, Button

def enviar_formulario():
    # lógica para procesar el formulario
    nombre = entry_nombre.get()
    apellido = entry_apellido.get()
    print(f'Nombre: {nombre}, Apellido: {apellido}')

# Creacion de la ventana principal
root = Tk()
root.title("Formulario con Place")

# Widgets del formulario
label_nombre = Label(root, text="Nombre:")
entry_nombre = Entry(root)

label_apellido = Label(root, text="Apellido:")
entry_apellido = Entry(root)

boton_enviar = Button(root, text="Enviar", command=enviar_formulario)

# Organización con place
label_nombre.place(x=50, y=50, anchor="e")
entry_nombre.place(x=60, y=50)

label_apellido.place(x=50, y=80, anchor="e")
entry_apellido.place(x=60, y=80)

boton_enviar.place(x=80, y=120)

# Inicio del bucle principal
root.mainloop()
```

Optimización del Rendimiento en Tkinter:

La optimización del rendimiento es esencial para garantizar que las interfaces gráficas creadas con Tkinter sean eficientes y respondan de manera rápida a las interacciones del usuario.

Optimización el Rendimiento:



a. Uso Racional de Widgets:

- Evita la creación innecesaria de widgets. Cada widget añade carga a la interfaz, así que solo crea los que realmente necesitas.

b. Limita la Profundidad de Frames Anidados:

- Evita la anidación excesiva de frames, ya que esto puede afectar el rendimiento. Utiliza frames de manera estructurada y lógica, sin anidaciones innecesarias.

c. Utilización de grid con Ponderación de Columnas y Filas:

- Si usas grid, asigna pesos a columnas y filas mediante columnconfigure y rowconfigure. Esto permite que los widgets se expandan proporcionalmente al cambiar el tamaño de la ventana.

```
root.columnconfigure(0, weight=1)
root.rowconfigure(0, weight=1)
```

d. Evita el Uso Excesivo de place:

- Aunque place puede ser útil en ciertos casos, su uso excesivo puede afectar el rendimiento. Utiliza pack o grid para la mayoría de los casos.

e. Implementa Actualizaciones Diferidas:

- Si realizas actualizaciones en la interfaz, considera usar el método update_idletasks() para diferir la actualización hasta que la interfaz esté en un estado inactivo.

```
root.update_idletasks()
```

Estrategias para Evitar Problemas Comunes de Diseño:

a. Evitar Widgets Invisibles:

- No mantengas widgets invisibles en la interfaz si no son necesarios. Puedes ocultar y mostrar widgets según sea necesario para reducir la carga.

b. Gestión Eficiente de Eventos:

- Evita la sobrecarga de eventos. Asocia únicamente aquellos eventos necesarios para la funcionalidad de tu aplicación.

c. Limita el Uso de Imágenes Pesadas:

- Si usas imágenes, asegúrate de que sean optimizadas y no excesivamente grandes. Imágenes pesadas pueden afectar significativamente el rendimiento.

d. Limpieza de Recursos no Necesarios:

- Libera recursos no necesarios, como variables y widgets, cuando ya no se utilicen. Esto ayuda a reducir la memoria utilizada.

e. Pruebas en Diferentes Entornos:

- Realiza pruebas en diferentes entornos y dispositivos para asegurarte de que el rendimiento sea aceptable en diversas configuraciones.

f. Considera la Actualización Incremental:

- Si actualizas muchos widgets simultáneamente, considera dividir la actualización en partes más pequeñas y aplicarlas incrementalmente.

g. Uso Cauteloso de la Actualización Continua:

- Evita la actualización continua de la interfaz si no es esencial. Actualizaciones constantes pueden afectar la respuesta y el rendimiento.

Al aplicar estos consejos y estrategias, puedes optimizar el rendimiento de tus interfaces gráficas en Tkinter y garantizar una experiencia fluida para los usuarios. Recuerda que la optimización puede ser un proceso iterativo, y es importante realizar pruebas para evaluar el impacto de los cambios realizados en el rendimiento.



Conceptos Básicos de Eventos en Tkinter:

Introducción a los Eventos del Usuario:

Los eventos del usuario en Tkinter son acciones específicas realizadas por el usuario, como clics de ratón, pulsaciones de teclas o movimientos del ratón. Estos eventos son fundamentales para la interactividad en las interfaces gráficas y permiten que la aplicación responda dinámicamente a las acciones del usuario.

Algunos ejemplos de eventos comunes son:

- Clics de ratón (izquierdo, derecho, medio).
- Pulsaciones de teclas.
- Movimientos del ratón.
- Entrada de teclado en widgets específicos (por ejemplo, Entry).

Cómo Tkinter Detecta y Maneja Eventos:

Tkinter utiliza un modelo de programación basado en eventos para gestionar las interacciones del usuario. Aquí hay una breve explicación de cómo Tkinter detecta y maneja eventos:

a. Ciclo Principal (mainloop):

Tkinter tiene un bucle principal (mainloop) que está en constante espera de eventos. Este bucle se ejecuta continuamente, permitiendo que la aplicación responda a eventos a medida que ocurren.

b. Enlace de Eventos a Funciones:

Los eventos se enlazan a funciones específicas. Cuando ocurre un evento, se ejecuta la función asociada a ese evento.

Por ejemplo, puedes enlazar la pulsación de un botón (<Button-1>) a una función que realiza una acción específica.

```
boton = Button(root, text="Haz clic")
boton.bind("<Button-1>", funcion_clic)
```

c. Funciones de Manejo de Eventos:

Las funciones de manejo de eventos son aquellas que se ejecutan cuando ocurre un evento específico.

Estas funciones toman un argumento (el evento) que contiene información sobre el evento, como la posición del ratón, tecla presionada, etc.

```
def funcion_clic(event):
    print("Se hizo clic en el botón en:", event.x, event.y)
```

d. Eventos Predeterminados:

Tkinter también tiene eventos predeterminados para widgets comunes. Por ejemplo, el evento <Return> se activa cuando se presiona la tecla "Enter" en un Entry.

```
entry = Entry(root)
entry.bind("<Return>", funcion_enter)
```

e. Eventos de Ratón:

Para eventos de ratón, puedes especificar el tipo de botón y otros detalles. Por ejemplo,



<Button-1> representa el clic izquierdo del ratón.

```
def clic_izquierdo(event):
    print("Se hizo clic izquierdo en:", event.x, event.y)
    widget.bind("<Button-1>", clic_izquierdo)
```

Asociación de Funciones a Eventos en Tkinter:

Detalles sobre Cómo Asociar Funciones a Eventos:

Asociar funciones a eventos en Tkinter es esencial para lograr la interactividad deseada en la interfaz gráfica. Aquí están los detalles sobre cómo realizar esta asociación:

a. Utilización del método bind():

El método bind() se utiliza para asociar funciones a eventos específicos de widgets. Se aplica al widget al que se desea vincular el evento.

b. Sintaxis Básica:

La sintaxis básica del método bind() es la siguiente:

```
widget.bind("<NombreEvento>", funcion_manejadora)
```

<NombreEvento> es la cadena que representa el evento específico, como <Button-1> para el clic izquierdo del ratón.

c. Funciones Manejadoras de Eventos:

Las funciones manejarán los eventos y pueden tomar un argumento (generalmente llamado event) que contiene información sobre el evento.

Pueden realizar acciones específicas basadas en el tipo de evento.

d. Desvinculación de Eventos:

Si en algún momento deseas desvincular una función de un evento, puedes utilizar el método unbind().

```
widget.unbind("<NombreEvento>", funcion_manejadora)
```

¿Cómo Responder a Eventos? Ejemplos:

Responder a un Clic de Botón:

```
from tkinter import Tk, Button
def manejar_clic(event):
    print("Se hizo clic en el botón")

# Creación de la ventana principal
root = Tk()

# Creación de un botón
boton = Button(root, text="Haz clic")

# Asociación de la función a un evento de clic
boton.bind("<Button-1>", manejar_clic)

# Colocación del botón en la ventana
```



```
boton.pack()  
  
# Inicio del bucle principal  
root.mainloop()
```

Desvinculación de Evento:

```
from tkinter import Tk, Button  
  
def manejar_clic(event):  
    print("Se hizo clic en el botón")  
    # Desvincular la función después del primer clic  
    boton.unbind("<Button-1>", manejar_clic)  
  
# Creación de la ventana principal  
root = Tk()  
  
# Creación de un botón  
boton = Button(root, text="Haz clic")  
  
# Asociación de la función a un evento de clic  
boton.bind("<Button-1>", manejar_clic)  
  
# Colocación del botón en la ventana  
boton.pack()  
  
# Inicio del bucle principal  
root.mainloop()
```

Eventos Comunes y Manejadores Correspondientes en Tkinter:

Listado de eventos comunes en Tkinter y los manejadores de eventos asociados. Además, te proporcionaré ejemplos de uso para dos eventos específicos: <Button-1> (clic izquierdo) y <Return> (pulsación de la tecla Enter).

Listado de Eventos Comunes y Manejadores Correspondientes:

Eventos de Ratón:

- <Button-1>: Clic izquierdo del ratón.
- <Button-2>: Clic central del ratón.
- <Button-3>: Clic derecho del ratón.
- <B1-Motion>: Movimiento del ratón durante el clic izquierdo.
- <Enter>: Entrada del ratón al área del widget.
- <Leave>: Salida del ratón del área del widget.

Eventos de Teclado:

- <KeyPress>: Pulsación de cualquier tecla.
- <KeyRelease>: Liberación de cualquier tecla.
- <Return>: Pulsación de la tecla Enter.
- <BackSpace>: Pulsación de la tecla de retroceso.



- <Tab>: Pulsación de la tecla de tabulación.
- <Shift-Up>: Pulsación de la tecla de flecha hacia arriba con la tecla Shift presionada.

Otros Eventos:

- <FocusIn>: Obtención del foco por parte del widget.
- <FocusOut>: Pérdida del foco por parte del widget.
- <Configure>: Cambio en la configuración (tamaño, posición) del widget.

Eventos Específicos:

Evento <Button-1> (Clic Izquierdo):

```
from tkinter import Tk, Button

def manejar_clic(event):
    print("Se hizo clic izquierdo en el botón")

# Creación de la ventana principal
root = Tk()

# Creación de un botón
boton = Button(root, text="Haz clic izquierdo")

# Asociación de la función a un evento de clic izquierdo
boton.bind("<Button-1>", manejar_clic)

# Colocación del botón en la ventana
boton.pack()

# Inicio del bucle principal
root.mainloop()
```

Evento <Return> (Pulsación de la Tecla Enter):

```
from tkinter import Tk, Entry

def manejar_enter(event):
    print("Se presionó la tecla Enter en el cuadro de entrada")
    # Realizar acciones adicionales...

# Creación de la ventana principal
root = Tk()

# Creación de un cuadro de entrada
cuadro_entrada = Entry(root)

# Asociación de la función a un evento de tecla Enter
cuadro_entrada.bind("<Return>", manejar_enter)

# Colocación del cuadro de entrada en la ventana
```



```
cuadro_entrada.pack()
```

```
# Inicio del bucle principal  
root.mainloop()
```

Interacción con Entrada del Usuario en Tkinter:

Exploración de Eventos Relacionados con la Entrada del Usuario:

En Tkinter, puedes explorar varios eventos relacionados con la entrada del usuario, especialmente en campos de entrada (Entry) y áreas de texto (Text). Algunos eventos relevantes incluyen:

<FocusIn>: Cuando el widget obtiene el foco.

<FocusOut>: Cuando el widget pierde el foco.

<Key>: Pulsación de cualquier tecla.

Estos eventos son esenciales para realizar acciones específicas en respuesta a la entrada del usuario.

Ejemplos de Validación y Respuesta a la Entrada del Usuario en Tiempo Real:

Validación de números en un Campo de Entrada:

En este ejemplo, validaremos que solo se ingresen números en un campo de entrada (Entry).

Si el usuario intenta ingresar un carácter no numérico, se mostrará un mensaje de error.

```
from tkinter import Tk, Entry, Label, StringVar  
  
def validar_numeros(nuevo_valor):  
    try:  
        # Intenta convertir el nuevo valor a un número  
        float(nuevo_valor)  
        mensaje_error.set("") # No hay error, se borra el mensaje  
    except ValueError:  
        # Si hay un error al convertir a número, muestra un mensaje de error  
        mensaje_error.set("Solo se permiten números")  
  
    # Creación de la ventana principal  
    root = Tk()  
  
    # Variable de control para el mensaje de error  
    mensaje_error = StringVar()  
  
    # Etiqueta de mensaje de error  
    etiqueta_error = Label(root, textvariable=mensaje_error, fg="red")  
    etiqueta_error.pack()  
  
    # Campo de entrada para números  
    entrada_numeros = Entry(root)  
    entrada_numeros.pack()
```



```
# Asociación de la función de validación al evento de pulsación de teclas
entrada_números.bind("<Key>", lambda event:
validar_números(entrada_números.get()))

# Inicio del bucle principal
root.mainloop()
```

Actualización de un Área de Texto en Tiempo Real:

En este ejemplo, actualizaremos un área de texto (Text) en tiempo real cada vez que se escriba en un campo de entrada (Entry).

```
from tkinter import Tk, Entry, Text

def actualizar_texto(event):
    texto = entrada_texto.get()
    area_texto.delete(1.0, "end") # Borra el contenido actual del área de texto
    area_texto.insert("end", texto) # Inserta el nuevo texto

# Creación de la ventana principal
root = Tk()

# Campo de entrada
entrada_texto = Entry(root)
entrada_texto.pack()

# Área de texto
area_texto = Text(root, height=5, width=30)
area_texto.pack()

# Asociación de la función de actualización al evento de pulsación de teclas
entrada_texto.bind("<Key>", actualizar_texto)

# Inicio del bucle principal
root.mainloop()
```

Eventos de Ratón y Teclado en Tkinter (Avanzados):

Detalles sobre Cómo Manejar Eventos de Ratón:

En Tkinter, puedes manejar una variedad de eventos de ratón que te permiten responder a las acciones del usuario. Algunos eventos de ratón comunes incluyen:

- <Button-1>: Clic izquierdo del ratón.
- <Button-2>: Clic central del ratón.
- <Button-3>: Clic derecho del ratón.
- <B1-Motion>: Movimiento del ratón durante el clic izquierdo.
- <Enter>: Entrada del ratón al área del widget.



- <Leave>: Salida del ratón del área del widget.

Ejemplos de Eventos de Teclado:

Eventos de Teclado Básicos:

```
from tkinter import Tk, Label
def manejar_pulsacion_tecla(event):
    tecla_pulsada = event.char # Obtener la tecla pulsada
    etiqueta.config(text=f"Tecla Pulsada: {tecla_pulsada}")
# Creación de la ventana principal
root = Tk()
# Etiqueta para mostrar la tecla pulsada
etiqueta = Label(root, text="Tecla Pulsada: ")
etiqueta.pack()
# Asociación de la función al evento de pulsación de tecla
root.bind("<Key>", manejar_pulsacion_tecla)
# Inicio del bucle principal
root.mainloop()
```

Combinaciones de Teclas:

```
from tkinter import Tk, Label

def manejar_combinacion_teclas(event):
    combinación = event.keysym # Obtener la combinación de teclas
    etiqueta.config(text=f"Combinación de Teclas: {combinación}")

# Creación de la ventana principal
root = Tk()

# Etiqueta para mostrar la combinación de teclas
etiqueta = Label(root, text="Combinación de Teclas: ")
etiqueta.pack()

# Asociación de la función al evento de combinación de teclas
root.bind("<Control-c>", manejar_combinacion_teclas)

# Inicio del bucle principal
root.mainloop()
```

Personalización de Eventos en Tkinter:

Estrategias para Personalizar y Crear Eventos Personalizados:

En Tkinter, puedes personalizar eventos y crear eventos personalizados mediante las siguientes estrategias:

método event_generate():

Utiliza el método event_generate() para generar eventos personalizados. Puedes especificar el tipo de evento, como clic de ratón o pulsación de tecla, y proporcionar detalles



adicionales.

```
widget.event_generate("<NombreEvento>", **detalles_evento)
```

Uso de Clases de Eventos Personalizadas:

Puedes crear clases personalizadas que hereden de la clase tk.Event. Esto te permite definir eventos específicos con atributos personalizados.

```
class MiEvento(tk.Event):
    def __init__(self, atributo_personalizado):
        self.atributo_personalizado = atributo_personalizado
```

Ejemplos de Eventos Personalizados y su Manejo:

Evento Personalizado mediante event_generate():

```
from tkinter import Tk, Button
def manejar_evento_personalizado(event):
    print("Se generó un evento personalizado")

# Creación de la ventana principal
root = Tk()

# Creación de un botón
boton = Button(root, text="Generar Evento")

# Asociación de la función al evento personalizado
boton.bind("<MiEvento>", manejar_evento_personalizado)

# Generación del evento personalizado al hacer clic en el botón
boton.event_generate("<MiEvento>")

# Colocación del botón en la ventana
boton.pack()

# Inicio del bucle principal
root.mainloop()
```

Evento Personalizado mediante Clase de Evento:

```
from tkinter import Tk, Canvas, Event

class MiEvento(Event):
    def __init__(self, atributo_personalizado):
        self.atributo_personalizado = atributo_personalizado

    def manejar_evento_personalizado(event):
        print(f"Se generó un evento personalizado con atributo:
{event.atributo_personalizado}")

# Creación de la ventana principal
```



```
root = Tk()

# Creación de un lienzo
lienzo = Canvas(root, width=200, height=100, bg="white")

# Asociación de la función al evento personalizado
lienzo.bind("<MiEvento>", manejar_evento_personalizado)

# Generación de un evento personalizado mediante una instancia de la clase de
# evento
evento_personalizado = MiEvento(atributo_personalizado="Ejemplo")
lienzo.event_generate("<MiEvento>", when="now", state="normal",
event=evento_personalizado)

# Colocación del lienzo en la ventana
lienzo.pack()

# Inicio del bucle principal
root.mainloop()
```

Gestión de Eventos de Menú en Tkinter:

Cómo Gestionar Eventos Asociados a la Selección de Elementos en Menús:

En Tkinter, puedes gestionar eventos asociados a la selección de elementos en menús mediante el uso de la opción command al crear elementos de menú. La opción command permite asociar una función que se ejecutará cuando se seleccione el elemento de menú.

```
menu.add_command(label="Elemento", command=funcion_asociada)
```

Ejemplos Prácticos de Ejecución de Acciones Específicas al Seleccionar Elementos de Menú: Menú de Opciones con Funciones Asociadas:

```
from tkinter import Tk, Menu, messagebox

def mostrar_mensaje():
    messagebox.showinfo("Información", "Elemento de menú seleccionado")

# Creación de la ventana principal
root = Tk()

# Creación de un menú
menu_principal = Menu(root)

# Creación de un menú desplegable
menu_desplegable = Menu(menu_principal, tearoff=0)
# Añadir elementos al menú desplegable con funciones asociadas
menu_desplegable.add_command(label="Opción 1", command=mostrar_mensaje)
menu_desplegable.add_command(label="Opción 2", command=mostrar_mensaje)
menu_desplegable.add_separator()
menu_desplegable.add_command(label="Salir", command=root.destroy)
# Añadir el menú desplegable al menú principal
menu_principal.add_cascade(label="Menú", menu=menu_desplegable)
```



```
# Configuración del menú principal en la ventana
root.config(menu=menu_principal)
# Inicio del bucle principal
root.mainloop()
```

Menú de Radio con Funciones Asociadas:

```
from tkinter import Tk, Menu, IntVar, messagebox
def seleccionar_opcion():
    selección = opcion_var.get()
    if selección == 1:
        messagebox.showinfo("Información", "Opción A seleccionada")
    elif selección == 2:
        messagebox.showinfo("Información", "Opción B seleccionada")
# Creación de la ventana principal
root = Tk()
# Creación de un menú
menu_principal = Menu(root)
# Variable para rastrear la opción seleccionada en el menú de radio
opcion_var = IntVar()
# Creación de un menú de radio con funciones asociadas
menu_radio = Menu(menu_principal, tearoff=0)
menu_radio.add_radiobutton(label="Opción A", variable=opcion_var, value=1,
                           command=seleccionar_opcion)
menu_radio.add_radiobutton(label="Opción B", variable=opcion_var, value=2,
                           command=seleccionar_opcion)
# Añadir el menú de radio al menú principal
menu_principal.add_cascade(label="Menú Radio", menu=menu_radio)
# Configuración del menú principal en la ventana
root.config(menu=menu_principal)
# Inicio del bucle principal
root.mainloop()
```

Actualización Dinámica y Eventos Continuos en Tkinter:

Estrategias para Realizar Actualizaciones Dinámicas en Respuesta a Eventos Continuos:

En Tkinter, puedes realizar actualizaciones dinámicas en respuesta a eventos continuos mediante el uso de la función `after()` y la actualización de variables de control. La función `after()` permite programar la ejecución de una función después de un cierto tiempo.

```
def actualizar():
    # Actualizar la interfaz o realizar acciones específicas
    root.after(tiempo_milisegundos, actualizar)
    # Llamada recursiva para continuar la actualización
```

además, puedes utilizar variables de control (`StringVar`, `IntVar`, etc.) y asociar funciones a eventos específicos para lograr actualizaciones dinámicas.

Ejemplos de Cómo Implementar Animaciones Simples mediante Eventos Continuos:



Animación de un Label en Movimiento:

Aquí se muestra cómo realizar una animación simple moviendo un Label horizontalmente en la ventana mediante eventos continuos.

```
from tkinter import Tk, Label

def mover_label():
    x_actual = label.winfo_x()
    nuevo_x = (x_actual + 5) % ancho_ventana
    label.place(x=nuevo_x, y=50) # Mover el label a la nueva posición
    root.after(tiempo_milisegundos, mover_label) # Programar la siguiente actualización

# Creación de la ventana principal
root = Tk()
root.geometry("400x100")

# Creación de un label
label = Label(root, text="Animación", font=("Helvetica", 16))

# Obtener el ancho de la ventana
ancho_ventana = root.winfo_width()

# Posicionar el label inicialmente
label.place(x=10, y=50)

# Definir el tiempo de actualización en milisegundos
tiempo_milisegundos = 100

# Iniciar la animación
mover_label()

# Inicio del bucle principal
root.mainloop()
```

Actualización Dinámica de un Contador:

Aquí se muestra cómo realizar una actualización dinámica de un contador mostrado en un Label.

```
from tkinter import Tk, Label

def actualizar_contador():
    contador_var.set(contador_var.get() + 1) # Incrementar el valor del contador
    root.after(tiempo_milisegundos, actualizar_contador) # Programar la siguiente actualización

# Creación de la ventana principal
root = Tk()

# Variable de control para el contador
```



```
contador_var = IntVar()
contador_var.set(0)

# Creación de un label para mostrar el contador
label_contador = Label(root, textvariable=contador_var, font=("Helvetica", 16))

# Definir el tiempo de actualización en milisegundos
tiempo_milisegundos = 1000 # Actualizar cada segundo

# Posicionar el label del contador
label_contador.pack()

# Iniciar la actualización dinámica del contador
actualizar_contador()

# Inicio del bucle principal
root.mainloop()
```

Manejo de Eventos de Ventana en Tkinter:

Manejo de Eventos Relacionados con la Ventana:

En Tkinter, puedes manejar eventos relacionados con la ventana, como redimensionamiento y cierre, mediante la asociación de funciones a eventos específicos.

```
def manejar_redimensionamiento(event):
    # Acciones a realizar cuando la ventana se redimensiona
    pass
def manejar_cierre():
    # Acciones a realizar antes de cerrar la ventana
    root.destroy() # Cerrar la ventana
# Asociación de funciones a eventos de ventana
root.bind("<Configure>", manejar_redimensionamiento) # Redimensionamiento de la ventana
root.protocol("WM_DELETE_WINDOW", manejar_cierre) # Cierre de la ventana.
```

Ejemplos Prácticos de Personalización de Acciones en Respuesta a Eventos de Ventana:

Actualización de Dimensiones en Tiempo Real:

Aquí se muestra cómo actualizar dinámicamente las dimensiones de la ventana en tiempo real.

```
from tkinter import Tk, Label

def actualizar_dimensiones(event):
    dimensiones = f"Ancho: {event.width}, Altura: {event.height}"
    label_dimensiones.config(text=dimensiones)

# Creación de la ventana principal
root = Tk()
```



```
# Creación de un label para mostrar las dimensiones
label_dimensiones = Label(root, text="Dimensiones:")
label_dimensiones.pack()

# Asociación de la función al evento de redimensionamiento
root.bind("<Configure>", actualizar_dimensiones)

# Inicio del bucle principal
root.mainloop()
```

Confirmación de Cierre de Ventana:

Aquí se muestra cómo personalizar el cierre de la ventana con una confirmación.

```
from tkinter import Tk, Button, messagebox

def confirmar_cierre():
    respuesta = messagebox.askokcancel("Confirmar Cierre", "¿Está seguro de que
desea cerrar la ventana?")
    if respuesta:
        root.destroy() # Cerrar la ventana si se confirma
# Creación de la ventana principal
root = Tk()
# Creación de un botón para cerrar la ventana con confirmación
boton_cerrar = Button(root, text="Cerrar Ventana", command=confirmar_cierre)
boton_cerrar.pack()
# Inicio del bucle principal
root.mainloop()
```

Depuración y Manejo de Excepciones en Tkinter:

Cómo Depurar Problemas Relacionados con la Gestión de Eventos:

Imprimir Mensajes de Depuración:

Utiliza declaraciones print para imprimir mensajes de depuración en puntos clave de tu código relacionados con la gestión de eventos. Esto te permitirá identificar dónde se produce un problema. Divide y Vencerás:

Divide el código en secciones más pequeñas y verifica cada parte por separado. Esto ayuda a identificar la sección del código que puede estar causando problemas.

Uso de try-except:

Envuelve bloques de código críticos con bloques try-except para capturar y manejar excepciones. Esto proporciona una manera de anticipar y abordar problemas potenciales.

Manejo de Excepciones en Eventos para Mantener la Estabilidad de la Aplicación:

Manejo General de Excepciones:

Envolverse en un bloque try-except general en eventos críticos. Esto captura cualquier excepción y permite manejarla adecuadamente sin interrumpir el flujo de la aplicación.

```
try:
```



```
# Código crítico del evento
except Exception as e:
    print(f"Error en evento: {e}")
```

Log de Errores:

Utiliza un sistema de registro para registrar errores. Puedes utilizar el módulo logging para redirigir mensajes de error a un archivo de registro.

```
import logging
logging.basicConfig(filename='errores.log', level=logging.ERROR)
try:
    # Código crítico del evento
    except Exception as e:
        logging.error(f"Error en evento: {e}")
```

Manejo Específico de Excepciones:

Identifica excepciones específicas que puedan ocurrir y maneja cada una de manera adecuada. Esto permite una gestión más precisa de los errores.

```
try:
    # Código crítico del evento
    except TkinterError as te:
        print(f"Error de Tkinter: {te}")
    except Exception as e:
        print(f"Error general: {e}")
```

Estas estrategias ayudarán a mejorar la depuración y el manejo de excepciones en eventos en tu aplicación Tkinter, permitiéndote identificar y abordar problemas de manera efectiva.

Extra

Capturador de pantallas

```
import pyautogui
import time
def capturar_pantalla(x, y, ancho, alto, nombre_archivo):
    # Espera unos segundos para que tengas tiempo de enfocar la ventana que deseas capturar
    time.sleep(5)
    # Captura la región especificada de la pantalla
    screenshot = pyautogui.screenshot(region=(x, y, ancho, alto))
    # Guarda la captura de pantalla en un archivo
    screenshot.save(nombre_archivo)
# Especifica las coordenadas (x, y) y las dimensiones (ancho, alto) del área que deseas capturar
x, y, ancho, alto = 100, 100, 300, 200
nombre_archivo = "captura.png"
capturar_pantalla(x, y, ancho, alto, nombre_archivo)
```