«Talento Tech»

Iniciación a la

Programación con Python

Clase 12

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Clase N° 12 | Persistencia de datos

Temario:

- Lectura y escritura de archivos de texto.
- Uso de archivos para almacenar datos de forma persistente.
- Introducción al manejo de excepciones (try-except).
- Conceptos fundamentales de bases de datos y SQL (visión general).

Objetivos de la Clase

En esta clase, aprenderás a gestionar datos de manera **persistente** utilizando archivos de texto. Esto te permitirá guardar información generada por tu programa para que pueda ser recuperada en ejecuciones futuras, una habilidad necesaria para construir sistemas más prácticos y funcionales.

Explorarás cómo leer y escribir en archivos, aplicando estas herramientas para almacenar información de manera eficiente. También introducirás el **manejo de excepciones** mediante **try-except**, una técnica que te permitirá prever errores comunes, como archivos inexistentes o datos corruptos, y asegurarte de que el programa siga funcionando de manera adecuada.

Finalmente, tendrás una primera aproximación al uso de **bases de datos** y **SQL**, entendiendo cómo estas herramientas avanzadas amplían las capacidades de almacenamiento y gestión de datos de tus programas, abriendo nuevas posibilidades para proyectos futuros.

Con estos conocimientos, estarás más cerca de desarrollar aplicaciones completas y listas para el mundo real, cumpliendo con las demandas actuales del manejo de datos.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Bienvenido a tu jornada en TalentoLab 🎉

Talento Lab

El día comienza en TechLab con una reunión de equipo liderada por Diego, el desarrollador senior. Te recibe con entusiasmo y comienza a explicar el nuevo desafío que el cliente ha planteado.

"*Tenemos un pedido muy interesante para este proyecto. El cliente necesita que la información de los usuarios y productos que registramos no se pierda cada vez que cerramos el programa. Quiere que el sistema sea capaz de guardar estos datos en un archivo, para que puedan ser recuperados cuando sea necesario.*"

Diego continúa: "*Hoy vamos a ver cómo leer y escribir en archivos de texto para guardar información como nombres, precios y cualquier dato que consideremos importante. También quiero que aprendan a manejar errores comunes, como intentar abrir un archivo que no existe, usando una herramienta fundamental:* ***try-except****. Además, les voy a dar una introducción al concepto de* ***bases de datos y SQL,*** *que son técnicas más avanzadas para manejar grandes cantidades de datos de manera eficiente.*"

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Persistencia de datos.

En el desarrollo de software, una de las necesidades más comunes es la capacidad de guardar información que pueda ser utilizada nuevamente, incluso después de cerrar el programa. Esto es lo que conocemos como **persistencia de datos**. Sin persistencia, toda la información ingresada o procesada durante la ejecución del programa desaparece al finalizarlo, lo que limita enormemente su utilidad.

Imaginemos un sistema de gestión de clientes o productos: sería poco práctico que los datos se perdieran cada vez que se cierra la aplicación. La persistencia de datos permite que los programas guarden información en **archivos** o **bases de datos**, para luego recuperarla cuando sea necesario. Esto hace que las aplicaciones sean más funcionales en escenarios del mundo real, donde la continuidad de la información es indispensable.

Exploraremos cómo podemos implementar la persistencia de datos en Python. Veremos cómo leer y escribir archivos de texto, lo que representa el primer paso en esta dirección. También aprenderemos a manejar errores que puedan surgir en este proceso y conoceremos los fundamentos de bases de datos y **SQL**, una herramienta poderosa para gestionar grandes volúmenes de datos de forma eficiente.

Archivos de texto.

Un archivo de texto es un tipo de archivo que almacena información en formato de **texto plano.** Esto significa que su contenido está compuesto únicamente por caracteres, como letras, números, símbolos y espacios, organizados en líneas. Estos archivos son legibles tanto por humanos como por programas de computadora. A diferencia de otros tipos de archivos, como imágenes o videos, que contienen datos binarios codificados para representar píxeles, colores, sonido o movimiento, los archivos de texto no requieren ningún software especializado para interpretarlos. Por ejemplo, un archivo de texto se puede abrir con un bloc de notas o cualquier editor de texto básico, mientras que un archivo de imagen necesita un visor gráfico.

En el contexto de la programación, los archivos de texto son muy útiles para guardar datos simples como configuraciones, registros de actividad, listas o resultados procesados por un programa. Al ser más livianos y fáciles de manejar, se utilizan ampliamente en proyectos iniciales o en situaciones donde no se requiere almacenar grandes volúmenes de datos estructurados.

Un archivo de texto suele tener extensiones como **.txt**, **.csv** o **.log**, dependiendo de su contenido y propósito. Por ejemplo, un archivo .csv (*Comma-Separated Values*) es un archivo de texto donde los datos están organizados en líneas y separados por comas, ideal para manejar tablas o listas.

En Python, interactuar con archivos de texto es una habilidad indispensable para agregar persistencia a los programas, permitiendo guardar información que luego se puede recuperar. Esto incluye acciones como abrir un archivo para lectura o escritura, agregar nuevos datos o procesar líneas específicas. A través de esta interacción, podemos extender un montón las capacidades de cualquier programa, haciendo que la información sea accesible más allá de la ejecución en tiempo real.

Archivos de texto en Python.

En Python, los archivos de texto se gestionan mediante funciones integradas que permiten **abrir**, **leer**, **escribir** y **cerrar** archivos con facilidad. Para interactuar con un archivo, primero es necesario abrirlo usando la función **open()**. Esta función requiere al menos dos argumentos: el nombre del archivo y el modo en el que se desea abrir. Los modos más comunes son:

- "r" para leer un archivo existente.
- "w" para escribir en un archivo (crea uno nuevo si no existe, y si existe, lo sobrescribe).
- "r+" para lectura y escritura combinadas
- "a" para agregar contenido al final de un archivo existente.

Por ejemplo, si queremos abrir un archivo llamado "datos.txt" para lectura, escribiríamos:

```
archivo = open("datos.txt", "r")
```

La línea anterior es una instrucción en Python que abre un archivo llamado **datos.txt** para interactuar con él. La función **open()** crea un objeto que nos permite realizar operaciones como leer, escribir o modificar su contenido.

En este caso, "**datos.txt**" es el nombre del archivo que queremos abrir. Python buscará este archivo en la misma carpeta donde se encuentra el script que estamos ejecutando. Si el archivo no existe y estamos intentando abrirlo en modo de lectura, el programa generará un error. El argumento **"r"** indica el modo en el que se abre el archivo, en este caso, solo lectura. Esto significa que podremos leer su contenido, pero no modificarlo.

La variable **archivo** almacena el objeto de archivo que crea la función open(). Este objeto es la necesario para interactuar con el contenido del archivo, ya que nos permite realizar operaciones como leer líneas o recorrer su contenido. Sin embargo, si el archivo no está disponible en la ubicación especificada, Python generará un error, lo que resalta la importancia de manejar excepciones -como veremos en breve- para evitar que el programa falle.

Una vez que terminamos de usar un archivo, es fundamental cerrarlo con el método **close()**. Esto asegura que todos los cambios realizados se guarden correctamente y que el sistema libere los recursos asociados al archivo.

Para leer el contenido de un archivo, Python ofrece varios métodos. Uno de los más comunes es **read()**, que devuelve todo el contenido del archivo como una cadena. Por ejemplo:

```
archivo = open("datos.txt", "r")
contenido = archivo.read()
print(contenido)
archivo.close()
```

Si queremos procesar el archivo línea por línea, podemos usar **readlines()**, que devuelve una lista donde cada elemento es una línea del archivo:

```
archivo = open("datos.txt", "r")
lineas = archivo.readlines()

for linea in lineas:
   # Eliminamos los saltos de línea al final de cada línea
   print(linea.strip())

archivo.close()
```

Este código abre el archivo de texto llamado **datos.txt** en modo lectura. La función **open()** , como antes, crea un objeto de archivo que se asigna a la variable **archivo**, permitiendo interactuar con el contenido del archivo. Luego, con **archivo.readlines(),** se leen todas las líneas del archivo y se almacenan en una lista llamada **lineas**, donde cada elemento de la lista representa una línea del archivo.

A continuación, se utiliza un **bucle for** para recorrer cada línea almacenada en la lista. Dentro del bucle, se usa el método **.strip()** para eliminar los saltos de línea o espacios adicionales al principio y al final de cada línea antes de imprimir su contenido en pantalla. Este método asegura que el texto se muestre limpio, sin caracteres de formato innecesarios.

Finalmente, el archivo se cierra con **archivo.close(),** para liberar los recursos asociados al archivo y garantizar que no queden archivos abiertos en el sistema. Este cierre explícito es una buena práctica de programación cuando se trabaja con archivos.

Para escribir en un archivo, debemos abrirlo en modo escritura ("w") o agregar ("a"). El método más usado para escribir es write(). Por ejemplo:

```
archivo = open("datos.txt", "w")
archivo.write("Hola, este es un archivo de prueba.\n")
archivo.write("Segunda línea del archivo.")
archivo.close()
```

Este código crea un archivo llamado "datos.txt" (o sobrescribe uno existente) y guarda las dos líneas de texto. Si queremos agregar contenido en lugar de sobrescribirlo, abrimos el archivo en modo "a":

```
archivo = open("datos.txt", "a")
archivo.write("Tercera línea del archivo.")
archivo.close()
```

Este último ejemplo agrega una nueva linea con el contenido "*Tercera línea del archivo.*" al archivo creado en el ejemplo anterior. El contenido final del archivo "datos.txt" es el siguiente:

```
Hola, este es un archivo de prueba.
Segunda línea del archivo.
Tercera línea del archivo.
```

Ejemplo práctico:

Ahora que hemos analizado estos conceptos, veamos un ejemplo práctico. Supongamos que queremos registrar una lista de nombres en un archivo y luego leerlos para mostrarlos.

```
# Escribir nombres en un archivo
archivo = open("nombres.txt", "w")
archivo.write("María\n")
archivo.write("Carlos\n")
archivo.write("Lucía\n")
archivo.close()

# Leer y mostrar el contenido del archivo
archivo = open("nombres.txt", "r")
print("Contenido del archivo:")
for linea in archivo:
   print(linea.strip())
archivo.close()
```

Este ejemplo comienza creando un archivo llamado **nombres.txt** en modo escritura utilizando la función **open()**. El parámetro **"w"** indica que el archivo será abierto para escribir, y si ya existe un archivo con el mismo nombre, su contenido será reemplazado. Luego, se utiliza el método **write()** para escribir tres nombres, cada uno seguido de un salto de línea (\n), lo que garantiza que cada nombre quede en una línea separada dentro del archivo. Una vez que se han escrito los datos, el archivo se cierra con **close()**, liberando los recursos asociados.

En la segunda parte, el mismo archivo se abre nuevamente, esta vez en modo lectura utilizando el parámetro **"r"**. Se imprime un encabezado para indicar que se mostrará el contenido del archivo. A continuación, se recorre cada línea del archivo con un bucle **for**. Para evitar mostrar los saltos de línea adicionales, se aplica el método **.strip()** a cada línea . Finalmente, el archivo se cierra otra vez con **close()**.

La apariencia de la terminal luego de ejecutar el programa es la siguiente:

```
Contenido del archivo:
María
Carlos
Lucía
```

Introducción al manejo de excepciones con try-except.

En el desarrollo de software, es común encontrarse con situaciones en las que un programa puede fallar debido a errores inesperados, como intentar leer un archivo que no existe, dividir por cero o trabajar con datos mal formateados. Estos errores, conocidos como excepciones, pueden interrumpir abruptamente la ejecución del programa si no se manejan correctamente.

El **manejo de excepciones** es un mecanismo que permite anticiparse a estos errores y reaccionar de manera controlada, asegurando que el programa no se detenga de forma inesperada y proporcionando mensajes útiles al usuario o desarrollador. En Python, este mecanismo se implementa con la estructura **try-except**.

El **bloque try-except** funciona como un sistema de seguridad: dentro del **bloque try** se coloca el código que puede generar una excepción, y dentro del **bloque except** se define cómo responder a esa excepción si ocurre. Esto permite que el programa continúe ejecutándose o tome acciones específicas frente a un error, como registrar el problema, pedir nuevamente una entrada válida o usar un valor por defecto.

```
try:
    # Código que puede generar una excepción
    # Por ejemplo: abrir un archivo, realizar
    # una operación matemática, convertir un dato, etc.
    operación_riesgosa()

except TipoDeExcepción:
    # Cómo manejar la excepción si ocurre
    # Mostrar un mensaje de error, registrar el problema,
    # usar un valor predeterminado, etc.
    manejar_error()
```

Por ejemplo, al trabajar con archivos, podríamos encontrarnos con situaciones como intentar abrir un archivo que no existe o carecer de permisos para escribir en él. Usar **try-except e**n estos casos no solo hace que el código sea más robusto, sino que también mejora la experiencia del usuario al manejar los errores de forma clara para construir programas confiables, capaces de operar en entornos reales, donde los errores son inevitables. Nos permite anticiparnos a las fallas, mantener el control y mejorar la calidad general del software.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Resumiendo:
Bloque try: Aquí colocás el código que puede generar un error. Python intentará ejecutarlo. Si no hay errores, el bloque except no se ejecutará.
Bloque except: Si ocurre un error en el bloque try, Python "salta" al bloque except correspondiente. Allí definís cómo manejar el error (por ejemplo, mostrando un mensaje o tomando alguna acción correctiva).

Ejemplo práctico:

Acá tenés un ejemplo práctico que utiliza try-except para manejar posibles errores al intentar abrir y leer un archivo en Python. Vamos a suponer que el archivo que queremos abrir tal vez no exista.

```
try:
    # Intentamos abrir un archivo para lectura
    archivo = open("datos.txt", "r")

    # Leemos todo el contenido del archivo
    contenido = archivo.read()
    print("Contenido del archivo:")

    # Mostramos el contenido del archivo
    print(contenido)

    # Cerramos el archivo
    archivo.close()

except FileNotFoundError:
    print("Error: El archivo 'datos.txt' no existe.")
    print("Verificá el nombre o la ubicación del archivo.")
```

¿Cómo funciona esto? Bien, en Python, cuando intentás abrir un archivo que no existe con el modo de lectura ("r"), se genera una excepción llamada **FileNotFoundError**. Este error es una subclase de las excepciones integradas en Python, y su función principal es indicar que el archivo solicitado no se pudo encontrar en la ubicación especificada. Si no manejamos esta excepción, el programa se detendría abruptamente, mostrando un mensaje de error.

El bloque except del ejemplonos permite "atrapar" **este tipo específico de error** y definir cómo debe responder el programa cuando ocurre. Al especificar **except FileNotFoundError:**, estamos indicando que queremos manejar exclusivamente este tipo de error. Esto es útil cuando trabajamos con archivos, ya que el error puede ser común y predecible, pero no queremos que detenga el funcionamiento del resto del programa.

En el ejemplo, al intentar abrir el archivo, si no existe en la carpeta donde se está ejecutando el script, Python lanza un FileNotFoundError, y el bloque try-except captura ese error y muestra un mensaje más claro al usuario.

Validar entrada de datos con try-except.

En muchos programas interactivos, como los que solicitan datos al usuario, es común encontrarse con errores si el usuario ingresa información incorrecta o inesperada. Por ejemplo, si un programa espera un número y el usuario escribe texto, esto puede generar una excepción y hacer que el programa se detenga de manera abrupta. Este tipo de situaciones no solo afectan la experiencia del usuario, sino que también hacen que el software parezca menos profesional.

Para evitar estos problemas, podemos usar el bloque **try-except** de Python. En lugar de detenerse, el programa puede mostrar un mensaje claro al usuario indicando el problema, pedir nuevamente la entrada correcta o incluso tomar una acción predeterminada. Veamos un ejemplo:

Buenos Aires aprende
Agencia de Habilidades para el Futuro

```python
print("=== Calculadora de División ===")
try:
    numerador = float(input("Ingresá el numerador: "))
    denominador = float(input("Ingresá el denominador: "))

    # Intentamos realizar la división
    resultado = numerador / denominador

    print(f"El resultado de la división es: {resultado:.2f}")

except ValueError:
    # Manejo de error si el usuario ingresa algo que no es
    # un número
    print("[ERROR] Debés ingresar valores numéricos válidos.")

except ZeroDivisionError:
    # Manejo de error si el usuario intenta dividir por cero
    print("[ERROR] El denominador no puede ser cero.")
    print(" Intentá nuevamente.")

print("Gracias por usar la calculadora.")
```

El programa del ejemplo es una calculadora sencilla que permite al usuario realizar divisiones. Primero, se imprime un título para informar al usuario qué hace el programa. Luego, se utiliza un **bloque try** para encapsular el código que podría generar errores.

En la primera parte, se solicita al usuario que ingrese el numerador y el denominador utilizando **input()**. Como el valor ingresado debe ser un número, lo convertimos a tipo **float()**.

Dentro del bloque try, se intenta realizar la operación de división. Si todo va bien, el programa muestra el resultado formateado a dos decimales. Sin embargo, si el usuario comete algún error, como ingresar texto en lugar de números, el bloque **except ValueError** captura la excepción y muestra un mensaje de error indicando que debe ingresar valores numéricos válidos.

Además, el bloque **except ZeroDivisionError** se encarga de manejar el caso particular en que el usuario intente dividir por cero, mostrando un mensaje que explica el problema.

Al finalizar, independientemente de si hubo errores o no, se imprime un mensaje de cierre agradeciendo al usuario por usar la calculadora.

Errores comunes manejados con except en Python.

Esta tabla incluye algunos de los errores más frecuentes que podemos manejar con try-except:

Tipo de error	Descripción
ValueError	Ocurre cuando se pasa un valor inválido a una operación o función.
TypeError	Surge cuando se usa un tipo de dato incorrecto en una operación o función.
IndexError	Se genera al intentar acceder a un índice fuera del rango válido en una lista o tupla.
KeyError	Aparece al intentar acceder a una clave que no existe en un diccionario.
FileNotFoundError	Se produce cuando se intenta abrir un archivo que no existe en el sistema.
ZeroDivisionError	Ocurre al intentar dividir un número por cero.
AttributeError	Surge cuando se intenta acceder a un atributo que no existe en un objeto.
ImportError	Aparece cuando no se puede importar un módulo o paquete, ya sea porque no existe o tiene errores.

Bases de datos.

Las **bases de datos** son herramientas que nos permiten almacenar, organizar y acceder a grandes cantidades de información de manera eficiente. Imaginá que estás gestionando una lista de clientes, productos o transacciones. A medida que la información crece, las soluciones como listas o archivos de texto se vuelven menos prácticas. Las bases de datos resuelven este problema al proporcionar una estructura organizada y un sistema para manejar estos datos.

Una base de datos no es más que un conjunto de datos estructurados, organizados en **tablas**. Cada tabla es como una hoja de cálculo donde las filas representan **registros** (datos específicos, como el nombre de un cliente) y las columnas representan **atributos** (como el correo o el número de teléfono de ese cliente).

SQL, que significa **Structured Query Language** (*Lenguaje de Consulta Estructurado*), es el lenguaje utilizado para interactuar con bases de datos. A través de SQL, es posible realizar operaciones esenciales como **agregar información nueva** a una tabla, **recuperar datos almacenados** aplicando filtros o condiciones específicas, **modificar registros existentes** y **eliminar información que ya no se necesita**. Este lenguaje es una herramienta estándar en el manejo de bases de datos y es ampliamente utilizado en proyectos de cualquier escala, desde pequeños sistemas hasta grandes aplicaciones empresariales.

Las bases de datos y SQL son muy importantes en el desarrollo de software porque permiten a los programas gestionar datos de manera eficiente. Estas herramientas ofrecen varias ventajas importantes. Por un lado, los datos se almacenan de manera estructurada, lo que facilita su acceso y manipulación. Además, la información persiste incluso después de cerrar el programa, asegurando su disponibilidad a largo plazo. Tanto SQL como las bases de datos son altamente escalables, lo que significa que siguen siendo prácticas incluso cuando la cantidad de datos crece significativamente.

En Python, es posible trabajar con bases de datos utilizando módulos como sqlite3, que permite conectar el lenguaje con una base de datos y ejecutar operaciones en ella. Aunque abordaremos ese módulo en la clase siguiente, presentaremos una visión general para que comprendas cómo SQL y Python se complementan y cómo esta integración es una base importante para desarrollar tus aplicaciones.

Python ofrece una variedad de herramientas para interactuar con bases de datos, y una de las más accesibles y versátiles es **SQLite**. SQLite es una biblioteca que implementa una base de datos relacional de alto rendimiento, ligera y autónoma, ideal para aplicaciones pequeñas y medianas. Una de sus principales ventajas es que no requiere configuración

previa ni un servidor externo, lo que la hace perfecta para aprender los fundamentos de bases de datos y SQL.

El módulo sqlite3, incluido de forma predeterminada en Python, permite conectar el lenguaje con una base de datos SQLite y ejecutar consultas SQL directamente desde el programa. Esto abre la posibilidad de crear aplicaciones que puedan manejar, organizar y procesar datos de manera eficiente. Por ejemplo, podés utilizar Python y SQLite para registrar datos de clientes o productos, realizar consultas para buscar información específica o actualizar registros existentes.

En la próxima clase, aprenderás a instalar y usar el módulo sqlite3, establecer conexiones con bases de datos SQLite y definir tablas para estructurar la información. También exploraremos las consultas SQL básicas, como SELECT, INSERT, UPDATE y DELETE, que son las herramientas esenciales para interactuar con los datos almacenados en una base de datos.

Ruta de avance.

Ahora que entendiste los conceptos básicos de persistencia de datos y comenzaste a trabajar con archivos en Python, estás un paso más cerca de dominar una habilidad clave para el desarrollo del Trabajo Final Integrador. Incorporar la posibilidad de guardar y recuperar información entre sesiones hará que tu proyecto sea más funcional y relevante.

Pensá en cómo podés aplicar estas herramientas a tu proyecto actual. Por ejemplo, si tu TFI incluye un sistema de gestión de clientes o productos, podrías usar archivos para almacenar los datos que el usuario ingrese y recuperarlos al reiniciar la aplicación. Esto permitirá que el sistema mantenga su utilidad en el tiempo y no pierda información al cerrarse.

Consejos para avanzar:

- Practicá leer y escribir archivos con distintos formatos de datos y estructuras. Por ejemplo, probá registrar listas de productos o nombres de clientes en un archivo y luego recuperarlos para mostrarlos en pantalla.
- Aplicá el manejo de excepciones que aprendiste con try-except para manejar posibles errores en la interacción con archivos, como la falta de permisos o la ausencia de un archivo.
- Reflexioná sobre cómo podés mejorar la experiencia del usuario en tu programa al agregar mensajes claros cuando ocurran errores o confirmaciones al guardar datos correctamente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Esta práctica no sólo te ayudará a consolidar lo aprendido en esta clase, sino que también te preparará para integrar bases de datos en tus proyectos, una habilidad que explorarás en la próxima clase. ¡Seguí adelante, estás haciendo un gran trabajo!

Ejercicio Práctico:

Como parte del proyecto de **TalentoLab**, el cliente necesita un sistema que permita registrar los datos de los usuarios en un archivo de texto, para que puedan ser recuperados y utilizados en futuras sesiones. Diego te asignó la tarea de desarrollar este sistema básico de persistencia de datos, aplicando todo lo que aprendiste. Tu tarea consiste en:

1) Creá un programa que permita ingresar los datos de un cliente: nombre, apellido y correo electrónico. Usá try-except para manejar posibles errores durante la entrada de datos. Por ejemplo:

- Si el archivo no puede abrirse para escritura.
- Si los datos ingresados no son válidos (por ejemplo, si el correo no contiene una "@" o el nombre está vacío).

2) Guardá los datos ingresados en un archivo de texto llamado clientes.txt, de forma que cada cliente quede registrado en una nueva línea.

3)Una vez guardados los datos, mostrá un mensaje de confirmación al usuario y cerrá el archivo correctamente.

La salida por la terminal podría ser algo asi:

```
=== Registro de Clientes ===
Ingresá los datos del cliente:
Nombre: Ana
Apellido: López
Correo: ana.lopez@email.com

Cliente registrado con éxito en clientes.txt
```

Si ocurre un error, como intentar guardar en un archivo protegido, el programa debería mostrar un mensaje claro:

```
[ERROR] No se pudo guardar la información.
```

Materiales y Recursos Adicionales:

Artículos:
Alfredo Sanchez Alberca: Control de errores mediante excepciones
Datacamp: Lectura y escritura de archivos en Python
TutorialesYa: Archivos de texto: creación, escritura y lectura
Oracle: ¿Qué son las bases de datos?

Videos:
Codigofacilito: Excepciones
Dimas: Manejo de archivos en Python

Preguntas para Reflexionar:

1. ¿Por qué es importante la persistencia de datos en las aplicaciones que desarrollamos? ¿Cómo creés que afecta esto a la experiencia del usuario final?
2. ¿Qué ventajas encontrás en el uso de archivos de texto para guardar información? ¿Cuáles serían las limitaciones de esta técnica frente a otras opciones como las bases de datos?
3. ¿Cómo te sentiste utilizando try-except para manejar errores? ¿Podés identificar situaciones reales en las que su uso sería esencial?

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Próximos Pasos:

En la próxima clase, vamos a dar un salto hacia una forma más avanzada y poderosa de manejar datos: las **bases de datos**. Aprenderás a utilizar **SQLite**, un sistema de gestión de bases de datos liviano y eficiente, ideal para proyectos pequeños y medianos. Exploraremos cómo integrar SQLite con Python a través del módulo sqlite3, lo que te permitirá crear, gestionar y consultar bases de datos desde tus programas.

Además, vamos a trabajar con consultas SQL básicas como **SELECT**, **INSERT**, **UPDATE** y **DELETE**, para que puedas manipular los datos de manera eficiente. Este conocimiento te abrirá la puerta a desarrollar aplicaciones que requieran manejar grandes volúmenes de información de forma estructurada y persistente.

Te recomiendo repasar lo aprendido sobre la persistencia de datos en archivos y reflexionar sobre las diferencias con el uso de bases de datos. Así estarás mejor preparado para comprender cómo SQL y Python se combinan para ofrecer una solución robusta a las necesidades de almacenamiento y recuperación de información.¡Nos vemos en la próxima clase!

Buenos Aires Ciudad