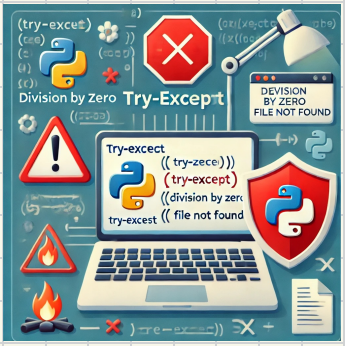
Una Ambiciosa Introducción a Python
Parte II



Una **excepción** es un evento que ocurre durante la ejecución de un programa y que interrumpe el flujo normal del código cuando se encuentra una situación inesperada o incorrecta, como una división por cero o el intento de abrir un archivo inexistente. Python ofrece una forma de **manejar** estas excepciones para evitar que el programa falle.

Existen diferentes tipos de excepciones en Python, y no todas deben manejarse de la misma forma

Errores que deben corregirse por el programador

- SyntaxError: Ocurre cuando hay un error en la estructura del código.
- NameError: Ocurre cuando se intenta usar una variable que no ha sido definida.
- IndentationError: Sucede cuando hay problemas en la indentación del código.

Estos errores reflejan problemas en el código que deben ser corregidos por el programador antes de la ejecución.

Errores que deben manejarse durante la ejecución

- ZeroDivisionError: Ocurre al intentar dividir un número por cero.
- FileNotFoundError: Se produce cuando se intenta acceder a un archivo que no existe.
- ValueError: Ocurre cuando se proporciona un tipo de dato incorrecto a una operación.
- IndexError: Sucede cuando se intenta acceder a un índice fuera del rango de una lista.
- KeyError: Aparece al intentar acceder a una clave inexistente en un diccionario.

Estos errores pueden ser impredecibles (entradas de usuario o condiciones externas) y deben manejarse durante la ejecución del programa usando bloques try-except.

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II

¿Cómo Manejar Excepciones en Python?

```
try:
    # Código que puede fallar, generar una excepción
except TipoDeExcepcion:
    # Bloque de código que nos va a permitir manejar la excepción
```

Ejemplo

```
try:
    resultado = 10 / 0
except ZeroDivisionError:
    print("Error: No se puede dividir por cero.")
```

El manejo de excepciones en Python se realiza con bloques try-except. Sin embargo, Python permite manejar más de una excepción en un mismo bloque y utilizar otros bloques adicionales como else y finally para tener un control más detallado del flujo del programa.

Manejo de Más de Una Excepción

Se pueden manejar varias excepciones utilizando múltiples bloques except. Cada uno se encarga de una excepción específica, y el orden es importante: las excepciones más específicas deben ir primero, seguidas de las más generales. Si existe una excepción general como Exception al principio, bloqueará otras excepciones mas específicas.

```
try:
    resultado = 10 / 0
except ZeroDivisionError:
    print("Error: División por cero.")
except ValueError:
    print("Error: Valor inválido.")
except Exception as e:  # Excepción general
    print(f"Ocurrió un error: {e}")
```

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II

El Bloque else

El bloque else se ejecuta solo si no ocurre ninguna excepción en el bloque try. Es útil para realizar acciones que solo deben ocurrir si el código dentro de try se ejecutó correctamente.

```
try:
    resultado = 10 / 2
except ZeroDivisionError:
    print("Error: División por cero.")
else:
    print(f"El resultado es {resultado}")  # Solo se ejecuta si no hubo excepción
```

El Bloque finally

El bloque finally se ejecuta siempre, independientemente de si ocurrió o no una excepción. Es ideal para liberar recursos, como cerrar archivos o conexiones, sin importar lo que suceda en el bloque try.

```
try:
    archivo = open("archivo.txt")
except FileNotFoundError:
    print("Error: El archivo no existe.")
finally:
    print("Este bloque se ejecuta siempre, cerrando recursos o limpiando.")  # Se ejecuta sin importar si hubo excepción
```

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II

Programación Defensiva

La programación defensiva es una técnica que busca prevenir los errores antes de que ocurran, en lugar de simplemente capturarlos después. Implica escribir código que verifique las condiciones antes de realizar operaciones potencialmente peligrosas.

En lugar de depender de ZeroDivisionError, es mejor verificar si el divisor es cero antes de realizar la operación, actuamos de manera preventiva.

```
def dividir(a, b):
    if b == 0:
        return "Error: No se puede dividir por cero."
    return a / b
```

Ventajas de la programación defensiva:

- Evita errores predecibles.
- Aumenta la robustez del programa.
- Mejora la experiencia del usuario con mensajes claros.
- Facilita el mantenimiento del código.

Aspecto	Programación Defensiva	Manejo Excepciones
Enfoque	Evitar errores antes de que ocurran.	Reaccionar a errores cuando ocurren.
Cuándo se usa	Para errores predecibles o evitables.	Para errores impredecibles o situaciones fuera de control.
Ejemplo	Verificar si el divisor es cero antes de dividir.	Usar un bloque try-except para capturar errores como ZeroDivisionError.
Ventajas	- Mejora la eficiencia. - Evita errores conocidos.	- Proporciona una segunda capa de seguridad. - Útil para manejar entradas inesperadas o situaciones complejas.
Desventajas	Puede requerir mucho código para validar todas las condiciones.	Puede generar un código menos eficiente si se usa para errores evitables.
Caso típico	Evitar una división por cero o la lectura de archivos inexistentes.	Capturar excepciones cuando el programa interactúa con datos externos o situaciones imprevistas.

Una Ambiciosa Introducción a Python
Parte II

En Python, una **excepción** es un evento que interrumpe el flujo normal de ejecución cuando ocurre una situación inesperada, como una división por cero o el intento de abrir un archivo inexistente. En muchos casos, el programa lanza excepciones automáticamente, pero en ciertas situaciones puede ser necesario **generar una excepción de manera explícita**. Para esto, Python proporciona la palabra reservada raise.

raise

Lanzar una excepción con raise nos permite **detectar y manejar errores de manera controlada**, evitando que el programa continúe en un estado incorrecto.

```
def verificar_edad(edad):
    if edad < 18:
        raise ValueError("La edad debe ser mayor o igual a 18.")
    return f"Acceso permitido. Edad: {edad}"

print(verificar_edad(20))  # Funciona correctamente
print(verificar_edad(15))  # Lanza ValueError
```

Cuando lanzamos una excepción con raise, podemos capturarla usando un bloque try-except para evitar que el programa se detenga abruptamente por quien realiza la invocación.

```
try:
    print(verificar_edad(15))
except ValueError as e:
    print(f"Error detectado: {e}")
```

Excepciones Personalizadas

Si bien Python proporciona muchas excepciones predefinidas (ValueError, IndexError, KeyError, etc.), en algunas situaciones necesitamos definir nuestras propias excepciones para representar errores específicos en nuestro programa.
Para crear una excepción personalizada, se debe definir una **clase que herede de Exception**.

```
class SaldoInsuficienteError(Exception):

    def __init__(self, saldo, monto):
        super().__init__(f"Saldo insuficiente: {saldo} disponible, {monto} requerido.")
        self.saldo = saldo
        self.monto = monto
```

Invocación al inicializador de la super clase Exception que espera un mensaje de error

Una Ambiciosa Introducción a Python
Parte II

```
def retirar_dinero(saldo, monto):
        if monto > saldo:
            raise SaldoInsuficienteError(saldo, monto)
        return saldo - monto
try:
    nuevo_saldo = retirar_dinero(100, 200)
except SaldoInsuficienteError as e:
    print(f"Error: {e}")
```

Ventajas de las Excepciones Personalizadas

- Permiten describir errores específicos de nuestro programa.
- Facilitan la depuración al proporcionar mensajes de error más claros.
- Mejoran la organización del código y la programación orientada a objetos.