Excepciones en Python

M2: FUNDAMENTOS DE INGENIERÍA DE DATOS

AE1: CODIFICAR UN ALGORITMO MANEJANDO LAS EXCEPCIONES PARA TOMAR ACCIONES SOBRE LOS ERRORES ACORDE AL LENGUAJE PYTHON Y A LAS BUENAS PRÁCTICAS DE LA DISCIPLINA.

alkemy

Introducción 🤦‍♂️

En el mundo de la programación, es común encontrarse con situaciones inesperadas que pueden interrumpir el flujo normal de un programa. Estas situaciones, conocidas como **excepciones**, son eventos que ocurren durante la ejecución de un programa y que alteran su comportamiento normal. Las excepciones pueden ser causadas por errores en el código, condiciones inesperadas del sistema o datos de entrada incorrectos proporcionados por el usuario.

El manejo adecuado de las excepciones es fundamental para desarrollar aplicaciones robustas y confiables. En Python, las excepciones son una parte integral del lenguaje y proporcionan un mecanismo poderoso para manejar errores y eventos inesperados. Este manual está diseñado para profundizar en el concepto de excepciones en Python, explorando su jerarquía, cómo crear excepciones personalizadas y las mejores prácticas para su manejo efectivo en el código.

Aprendizajes esperados

Cuando finalices la lección serás capaz de:

- Comprender qué es una excepción y su importancia en la programación.
- Conocer la jerarquía de excepciones en Python.
- Construir y utilizar excepciones personalizadas.
- Manejar y capturar excepciones utilizando las sentencias `try`, `except` y `finally`.
- Entender la propagación de excepciones en Python.
- Implementar excepciones en el código de manera efectiva.
- Utilizar correctamente las sentencias `try`, `except` y `finally`.
- Aplicar buenas prácticas para el manejo de excepciones.

Desarrollo

QUÉ ES UNA EXCEPCIÓN Y POR QUÉ SON IMPORTANTES

Una **excepción** es un evento anómalo que ocurre durante la ejecución de un programa y que interrumpe el flujo normal de las instrucciones. En Python, cuando ocurre una situación inesperada, el intérprete lanza una excepción. Si esta no es manejada adecuadamente, el programa se detiene y muestra un mensaje de error detallado, conocido como **traceback**.

Las excepciones son importantes por varias razones:

1. **Robustez del programa**: Permiten manejar errores y situaciones inesperadas sin que el programa termine abruptamente.
2. **Depuración**: Proporcionan información detallada sobre el tipo de error y dónde ocurrió, facilitando la identificación y corrección de problemas.
3. **Experiencia del usuario**: Al manejar excepciones, se pueden proporcionar mensajes de error claros y comprensibles, mejorando la interacción con el usuario.
4. **Mantenimiento del código**: Facilitan la escritura de código modular y más fácil de mantener al separar la lógica principal del manejo de errores.

Ejemplo de excepción no manejada:

```
resultado = 10 / 0
```

Al ejecutar este código, Python lanza una excepción `ZeroDivisionError` porque no es posible dividir un número entre cero.

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

JERARQUÍA DE EXCEPCIONES EN PYTHON

Python organiza las excepciones en una jerarquía de clases que heredan de la clase base `BaseException`. Esta estructura jerárquica permite manejar excepciones de manera más eficiente y específica.

Diagrama simplificado de la jerarquía de excepciones:

```
BaseException
├── SystemExit
```

```
├── KeyboardInterrupt
├── GeneratorExit
└── Exception
    ├── StopIteration
    ├── ArithmeticError
    |   ├── FloatingPointError
    |   ├── OverflowError
    |   └── ZeroDivisionError
    ├── LookupError
    |   ├── IndexError
    |   └── KeyError
    ├── ImportError
    ├── EOFError
    ├── RuntimeError
    |   └── RecursionError
    ├── SyntaxError
    ├── TypeError
    ├── ValueError
    |   └── UnicodeError
    └── ...
```

Descripción de algunas excepciones comunes:

- **`Exception`**: Clase base para todas las excepciones integradas excepto las del sistema.
- **`ArithmeticError`**: Clase base para errores en operaciones aritméticas.
    - **`ZeroDivisionError`**: Error al dividir un número por cero.
    - **`OverflowError`**: Ocurre cuando el resultado de una operación aritmética es demasiado grande para ser representado.
- **`LookupError`**: Clase base para errores de búsqueda.
    - **`IndexError`**: Error al acceder a un índice inválido en una secuencia.
    - **`KeyError`**: Error al acceder a una clave que no existe en un diccionario.
- **`TypeError`**: Error al realizar una operación o función en un tipo inapropiado.
- **`ValueError`**: Error al recibir un argumento con el tipo correcto pero valor inapropiado.
- **`ImportError`**: Error al intentar importar un módulo que no existe.

Tabla comparativa de excepciones comunes:

Excepción	Descripción	Ejemplo

`ZeroDivisionError`	División entre cero	`10 / 0`
`IndexError`	Índice fuera de rango en una secuencia	`lista = [1, 2, 3]; lista[5]`
`KeyError`	Clave inexistente en un diccionario	`dic = {'a': 1}; dic['b']`
`TypeError`	Operación en un tipo inapropiado	`5 + 'cadena'`
`ValueError`	Valor inapropiado para una operación o función	`int('abc')`
`ImportError`	Módulo no encontrado durante la importación	`import modulo_inexistente`

CONSTRUCCIÓN DE EXCEPCIONES PERSONALIZADAS

Python permite crear excepciones personalizadas definiendo nuevas clases que heredan de `Exception` o de alguna de sus subclases. Esto es útil cuando se necesita manejar errores específicos que no están cubiertos por las excepciones integradas.

Pasos para crear una excepción personalizada:

1. Definir una clase que herede de `Exception`.
2. (Opcional) Añadir un método `__init__` para inicializar atributos personalizados.
3. Utilizar la palabra clave `raise` para lanzar la excepción cuando sea necesario.

Ejemplo:

```
class ErrorDeValidacion(Exception):
    """Excepción personalizada para errores de validación."""

    def __init__(self, mensaje, campo):
        self.mensaje = mensaje
        self.campo = campo
        super().__init__(f"Error en {campo}: {mensaje}")

# Uso de la excepción personalizada
def validar_edad(edad):
    if edad < 0:
        raise ErrorDeValidacion("La edad no puede ser negativa",
"edad")
```

```
    elif edad > 120:
        raise ErrorDeValidacion("La edad es irreal", "edad")
    else:
        print(f"Edad válida: {edad}")

try:
    validar_edad(-5)
except ErrorDeValidacion as e:
    print(e)
```

Salida:

```
Error en edad: La edad no puede ser negativa
```

MANEJO Y CAPTURA DE LAS EXCEPCIONES (TRY-EXCEPT-FINALLY)

El manejo de excepciones en Python se realiza mediante las sentencias `try`, `except`, `else` y `finally`. Estas permiten capturar excepciones y definir acciones específicas para cada tipo de error.

Estructura básica:

```
try:
    # Bloque de código que puede generar una excepción
except NombreDeLaExcepcion as e:
    # Manejo de la excepción
else:
    # Código que se ejecuta si no ocurre ninguna excepción
finally:
    # Código que se ejecuta siempre, haya o no excepción
```

Descripción de las sentencias:

- **`try`**: Bloque donde se coloca el código que podría generar una excepción.
- **`except`**: Bloque que se ejecuta si ocurre una excepción en el bloque `try`.
- **`else`**: Opcional, se ejecuta si no ocurre ninguna excepción.
- **`finally`**: Opcional, se ejecuta siempre, independientemente de si ocurrió una excepción o no.

Ejemplo práctico:

```
try:
    archivo = open('datos.txt', 'r')
    contenido = archivo.read()
except FileNotFoundError as e:
    print("Error: El archivo no existe.")
except Exception as e:
    print(f"Error inesperado: {e}")
else:
    print("Archivo leído exitosamente.")
    print(contenido)
finally:
    if 'archivo' in locals():
        archivo.close()
        print("Archivo cerrado.")
```

Explicación:

- Si el archivo no existe, se captura la excepción `FileNotFoundError` y se muestra un mensaje.
- Si ocurre otra excepción, se captura con `Exception` y se muestra el error.
- Si no hay excepciones, se ejecuta el bloque `else`.
- El bloque `finally` se ejecuta siempre, asegurando que el archivo se cierre si fue abierto.

PROPAGACIÓN DE EXCEPCIONES

Cuando una excepción no es manejada en el contexto actual, se propaga hacia los niveles superiores de la pila de llamadas. Esto significa que la excepción sube por las funciones que llamaron a la función actual hasta que es capturada o llega al nivel más alto, causando que el programa termine.

Ejemplo:

```
def funcion1():
    funcion2()

def funcion2():
    funcion3()

def funcion3():
    raise ValueError("Error en funcion3")
```

```
try:
    funcion1()
except ValueError as e:
    print(f"Excepción capturada: {e}")
```

Explicación:

- `funcion3` lanza una excepción `ValueError`.
- La excepción se propaga a través de `funcion2` y `funcion1`.
- Finalmente, es capturada en el bloque `try-except`.

Salida:

```
Excepción capturada: Error en funcion3
```

UTILIZACIÓN DE EXCEPCIONES EN EL CÓDIGO

Las excepciones deben utilizarse para manejar situaciones excepcionales y no para el control de flujo normal del programa. A continuación, se presentan algunas formas comunes de utilizar excepciones en el código:

Validación de entradas:

```
def obtener_numero():
    while True:
        try:
            numero = int(input("Ingrese un número entero: "))
            return numero
        except ValueError:
            print("Entrada inválida. Por favor, intente de nuevo.")
```

Operaciones con archivos:

```
try:
    with open('datos.txt', 'r') as archivo:
        contenido = archivo.read()
except FileNotFoundError:
    print("El archivo no fue encontrado.")
```

Conexiones de red:

```
import requests
```

```
try:
    respuesta = requests.get('https://api.ejemplo.com/datos')
    datos = respuesta.json()
except requests.ConnectionError:
    print("Error de conexión. Verifique su conexión a Internet.")
except requests.JSONDecodeError:
    print("Error al decodificar la respuesta.")
```

Operaciones matemáticas:

```
def dividir(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("No se puede dividir entre cero.")
        return None
```

LAS SENTENCIAS TRY, EXCEPT Y FINALLY

Sentencia `try`

La sentencia `try` delimita un bloque de código en el cual pueden ocurrir excepciones. Es el primer paso para el manejo de excepciones.

Ejemplo:

```
try:
    # Código susceptible a errores
    resultado = 10 / 0
```

Sentencia `except`

La sentencia `except` captura excepciones específicas y permite definir acciones para cada tipo de error.

Ejemplo:

```
except ZeroDivisionError:
    print("Error: División entre cero.")
```

Se pueden especificar múltiples bloques `except` para manejar diferentes excepciones.

Sentencia `finally`

La sentencia `finally` se utiliza para definir acciones que deben ejecutarse siempre, independientemente de si ocurrió una excepción o no. Es útil para liberar recursos o realizar limpiezas.

Ejemplo:

```
finally:
    print("Operación finalizada.")
```

Uso combinado

Ejemplo completo:

```
try:
    archivo = open('datos.txt', 'r')
    contenido = archivo.read()
except FileNotFoundError:
    print("Error: El archivo no existe.")
else:
    print("Contenido del archivo:")
    print(contenido)
finally:
    if 'archivo' in locals() and not archivo.closed:
        archivo.close()
        print("Archivo cerrado.")
```

BUENAS PRÁCTICAS PARA EL MANEJO DE EXCEPCIONES

Capturar excepciones específicas: Evita usar `except Exception` a menos que sea absolutamente necesario. Capturar excepciones específicas ayuda a manejar errores de manera más precisa.

```
# Malo
try:
    # Código
except Exception:
    # Manejo general

# Bueno
try:
    # Código
```

```
except ValueError:
    # Manejo específico
```

No utilizar excepciones para el control de flujo normal: Las excepciones deben reservarse para situaciones excepcionales y no como una alternativa a las estructuras de control habituales.

```
# No recomendado
try:
    valor = diccionario['clave']
except KeyError:
    valor = 'valor por defecto'

# Recomendado
valor = diccionario.get('clave', 'valor por defecto')
```

Limitar el alcance del bloque `try`: Coloca solo el código que puede generar una excepción dentro del bloque `try` para evitar capturar excepciones no deseadas.

```
# Malo
try:
    # Código que puede fallar
    # Código que no falla
except Exception as e:
    # Manejo de la excepción

# Bueno
# Código que no falla
try:
    # Código que puede fallar
except Exception as e:
    # Manejo de la excepción
```

Proporcionar información útil en las excepciones personalizadas: Al crear excepciones personalizadas, incluye mensajes claros y, si es necesario, atributos adicionales para proporcionar contexto.

```
class ErrorDeConexion(Exception):
    def __init__(self, host, puerto):
        super().__init__(f"No se pudo conectar a {host}:{puerto}")
```

Utilizar el bloque `finally` para liberar recursos: Asegura que recursos como archivos, conexiones de bases de datos o sockets se liberen correctamente.

```
try:
    conexion = obtener_conexion()
    # Operaciones con la conexión
except ErrorDeConexion as e:
    print(e)
finally:
    if conexion:
        conexion.cerrar()
```

Registrar las excepciones: En aplicaciones más complejas, es útil registrar las excepciones para análisis posterior.

```
import logging

logging.basicConfig(filename='app.log', level=logging.ERROR)

try:
    # Código
except Exception as e:
    logging.error("Error ocurrido: %s", e)
```

Re-levantar excepciones cuando sea apropiado: Si después de manejar una excepción es necesario que esta siga propagándose, utiliza `raise` sin argumentos.

```
try:
    # Código
except MiExcepcion as e:
    # Manejo
    raise
```

Documentar las excepciones: Indica en la documentación de tus funciones qué excepciones pueden ser lanzadas.

```
def calcular_raiz_cuadrada(valor):
    """
    Calcula la raíz cuadrada de un número.

    :param valor: Número del cual calcular la raíz cuadrada.
    :type valor: float
    :raises ValueError: Si el valor es negativo.
```

```
    :return: Raíz cuadrada del valor.
    :rtype: float
    """
    if valor < 0:
        raise ValueError("El valor no puede ser negativo.")
    return valor ** 0.5
```

Cierre ✍️

El manejo adecuado de excepciones es una habilidad esencial para cualquier desarrollador de Python. Las excepciones permiten crear programas más robustos y resilientes al manejar errores y situaciones inesperadas de manera controlada. Al comprender la jerarquía de excepciones, crear excepciones personalizadas y aplicar buenas prácticas en su manejo, puedes mejorar significativamente la calidad y mantenibilidad de tu código.

Recuerda que las excepciones deben utilizarse para situaciones excepcionales y no como parte del flujo normal del programa. Un manejo de excepciones bien diseñado no solo mejora la experiencia del usuario, sino que también facilita la depuración y el mantenimiento del software.

Referencias 📚

- **Documentación oficial de Python**:
  - Manejo de excepciones: https://docs.python.org/es/3/tutorial/errors.html
  - Excepciones integradas: https://docs.python.org/es/3/library/exceptions.html
- **Libros**:
  - **"Aprendiendo Python"** por Mark Lutz.
  - **"Python para todos"** por Raúl González Duque.
  - **"Python Cookbook"** por David Beazley y Brian K. Jones.
- **Tutoriales en línea**:
  - **Real Python**:
    - Manejo de excepciones en Python
  - **Programiz**:
    - Python Exception Handling
  - **W3Schools**:
    - Python Try Except
- **Cursos en línea**:
  - **Coursera**:
    - "Python para todos" por la Universidad de Michigan.
  - **edX**:
    - "Introducción a la Programación en Python" por la Universidad Carlos III de Madrid.
  - **Udemy**:
    - "Completo Masterclass de Python 3" por Juan Gabriel Gomila.
- **Blogs y artículos**:
  - **GeeksforGeeks**:
    - Manejo de excepciones en Python
  - **Stack Overflow**:
    - Discusiones y soluciones a problemas específicos sobre excepciones.

¡Muchas gracias!

Nos vemos en la próxima lección

alkemy