Arellano Granados Angel Mariano

218123444

Computación Tolerante a Fallas

D06 2023B

Parte 1

Otras herramientas para el manejar errores

Introducción

Al programar es normal cometer errores, que son básicamente de tres tipos:

Sintácticos: el código escrito no conforma una expresión válida en Python y es el propio intérprete el que lanza el aviso con el error detectado. Suelen ser fáciles de corregir.

Errores en tiempo de ejecución (Runtime errors): el código es correcto, pero, en ocasiones, al ejecutarse, el programa falla.

Estos errores son fáciles de corregir si el programador detecta el error en la fase de desarrollo, ya que el intérprete lanza una excepción. Como veremos enseguida, Python tiene herramientas para que podamos manejar adecuadamente estos casos excepcionales.

En caso contrario, si el error surge con el programa ya en producción, entonces los efectos pueden ser catastróficos y reputacional mente graves.

Semánticos: el código es correcto y el programa se ejecuta sin problemas, pero los resultados no son válidos. Son los errores más difíciles de detectar y/o corregir pues muchas veces el programador o cliente ni siquiera es consciente de que algo va mal.

Desarrollo

Errores en tiempo de ejecución

> El enfoque “Piensa antes de actuar”
>
> El enfoque Piensa antes de actuar (LBYL, Look Before You Leap) preconiza la realización anticipada de pruebas explícitas para determinar si se satisfacen las condiciones que evitan la aparición de errores.
>
> Veámoslo con nuestro ejemplo:

```
numerador = 5
denominador = 0

if denominador != 0:
```

```
    cociente = numerador/denominador
    print(f'{numerador}/{denominador} = {cociente}')
else:
    print('Error: el denominador es nulo')
```

El enfoque “Es más sencillo pedir perdón que pedir permiso”

El enfoque Es más sencillo pedir perdón que pedir permiso (EAFP, Easier to Ask Forgiveness than Permission) promueve que, por regla general, es mejor probar (try) directamente la ejecución de las sentencias y, para los casos excepcionales, capturar (catch) el error.

La estructura condicional try ... except

La estructura condicional try ... except es:

```
try:
    # Probamos nuestro código
except TipoDeExcepcion:
    # Tratamos la excepción TipoDeExcepcion si ha sido capturada en el
bloque try
```

El bloque try se ejecuta (no pedimos permiso para ejecutar las sentencias).

Si no se produce ninguna excepción, se salta el bloque o bloques except.

Si al ejecutar alguna de las sentencias del bloque try se produce una excepción, el resto de sentencias del bloque try se ignoran.

Si el bloque try ha lanzado una excepción y su tipo coincide con alguna de las contempladas en un bloque except, tratamos la excepción ejecutando únicamente las sentencias de ese bloque.

try ... except ... else ... finally

De forma adicional a try ...except tenemos dos palabras clave, else y finally que, opcionalmente, pueden facilitarnos aún más el manejo de las excepciones.

La estructura básica es:

```
try:
    print("Probamos nuestro código susceptible de lanzar excepciones.")
except:
    print("Aquí tratamos las excepciones.")
else:
    print("Esta es la parte del programa que creemos que está libre de
excepciones.")
finally:
    print("Ocurra lo que ocurra, esta parte la ejecutamos siempre.")
```

El objetivo del bloque else es separar claramente la zona que creemos susceptible de generar excepciones de la que está libre de ellas. Esto tiene una ventaja adicional: si se produce una excepción no esperada del mismo tipo de las que ya manejamos en el bloque try no quedará enmascarada y podremos rehacer el código dándola el tratamiento adecuado.

El bloque finally suele usarse para tareas de limpieza (cleanup), tales como cerrar recursos que se han abierto, por ejemplo, un fichero, o cualquier otro tipo de sentencias que es necesario ejecutar haya habido o no una excepción.

Ejemplo:

```
try:
    numerador = float(input('Numerador: '))
    denominador = float(input('Denominador: '))
    cociente = numerador/denominador
except ZeroDivisionError:
    print('Error: el denominador es nulo')
except ValueError:
    print('Error: el valor introducido no es un número válido')
else:
    print(f'{numerador}/{denominador} = {cociente}')
finally:
    print('\nFin del programa.')
```

Creación manual de excepciones: raise

Muchos de los ejemplos de temas previos se ejecutan correctamente si utilizamos entradas legítimas.

Analicemos de nuevo el ejemplo de determinar si un número es primo cuando el usuario introduce un entero negativo o nulo.

```python
# Determina si un número entero es primo. (Versión 1)
numero = int(input('Deme un entero positivo mayor que 1: '))
es_primo = True  # Variable centinela o bandera
for div in range(2, numero):
    if numero % div == 0:
        es_primo = False
        break
if es_primo:
    print(f'El número {numero} es primo')
else:
    print(f'El número {numero} no es primo')
```

Usando la sentencia raise podemos forzar a que la entrada sea lo que el programa busca, terminando en un programa como el siguiente:

```python
# Determina si un número entero es primo. Manejo de excepciones con raise.
(Versión 3)
while True:
    try:
        numero = int(input('Deme un entero positivo mayor que 1: '))
        if numero < 2:
            raise ValueError('El entero debe ser igual o mayor que 2.')
        break  # Si ha llegado aquí, no se ha producido ninguna excepción
    except ValueError as error:
        print(f'Se ha producido el error: \'{error}\'.')
        print('Por favor, vuelva a introducir el valor.')
# A partir de aquí sabemos que estamos exentos de las excepciones
contempladas
es_primo = True
for div in range(2, numero):
    if numero % div == 0:
        es_primo = False
        break
if es_primo:
    print(f'El número {numero} es primo')
else:
    print(f'El número {numero} no es primo')
```

La creación de un programa robusto frente a todas las posibles entradas, es una de las facetas de la programación defensiva. Muchas veces, al programar nos centramos en los aspectos algorítmicos de un problema, pero más tarde es necesario ir progresivamente refinando el código. La realización de programas de prueba en paralelo que nos permitan testar nuestro código es otra faceta indispensable en la ingeniería del software.

Un formato habitual para forzar manualmente una excepción es raise TipoDeError(Mensaje). Aunque un programador puede personalizar sus propios tipos de error, lo recomendable es comprobar si alguno de los tipos de excepción nativos de Python se ajusta al tipo de error que queremos lanzar. Así, en nuestro ejemplo, parece lógico catalogar el error que puede cometer el usuario al introducir un valor no contemplado como del tipo ValueError.

El Mensaje que ponemos como argumento es el que se mostrará cuando except trate la excepción. En este caso, con la sintaxis except TipoDeError as error la variable error toma como valor la cadena de caracteres Mensaje que hemos asociado al TipoDeError.

Conclusión

En la gran mayoría de los lenguajes de programación existe una estructura igual o equivalente al try – catch la cual suele sen el método mas popular y efectivo para crear un programa tolerante a fallas, sin embargo, este no es infalible dado a que no puede ser 100% eficaz en ciertos tipos de códigos, por ello existen más métodos de validación que se adaptan a cada necesidad que se nos presente en nuestros algoritmos.

Bibliografía

- UVa. (2023). Fundamentos de Programación en Python. Escuela de Ingenierías Industriales. https://www2.eii.uva.es/fund_inf/python/notebooks/07_Manejo_de_excepciones/Manejo_de_excepciones.html