ESTUDIOS SUPERIORES DE BUENOS AIRES
ESBA

Excepciones

El camino feliz

El **happy path** es el camino ideal que debería seguir el usuario. Es el camino más corto, el que no tiene fallos técnicos ni errores humanos. Es el camino que permite que quien utilice esa aplicación pueda conseguir su objetivo de forma rápida, precisa y efectiva.

Es un tipo de prueba de software que utiliza datos de entrada conocidos y produce una salida esperada.

El camino feliz no duplica las condiciones del mundo real. Solo verifica que la funcionalidad requerida esté en su lugar y funcione correctamente en base a las casuísticas más triviales.

Por ende, se identifica como el escenario predeterminado o la alternativa positiva más probable sin condiciones excepcionales o de error.

Los escenarios del camino feliz excluyen las excepciones y los errores humanos, tales como la posibilidad de cargar valores fuera de rango o de diferente tipo.

Lote de prueba

Se deben realizar las pruebas no sólo para la situación ideal (el camino feliz) sino que nuestro código debe ser lo suficientemente robusto como para comprobar todas aquellas situaciones incómodas en las que una funcionalidad particular se podría ver comprometida.

La intención, es la de detectar errores en etapas tempranas de desarrollo. Si nos quedamos con los casos más sencillos, seguramente estaremos ocultando errores que, tarde o temprano, deberemos corregir.

¿Cuándo es el momento más oportuno para detectar errores y problemas?, ¿cuando estamos desarrollando una nueva funcionalidad o cuando el software está ya en producción con uno o varios clientes usándolo?.

Pensar previamente en los valores tanto válidos como inválidos, valores típicos y valores extremos, deducir los resultados esperados, luego probar el código con esos valores para contrastar con los resultados esperados, nos permitirá evaluar el comportamiento del algoritmo ante casi cualquier situación y obtener un código robusto (menos propenso a fallos).

Depuración de código

La depuración de programas es el proceso de identificar y corregir errores de programación (**debugging**). Si alguien busca la definición de bug en un diccionario de inglés descubrirá que significa insecto o bicho, pero para la informática un bug es cualquier error, mal funcionamiento o falla que produce que el software no funcione como se esperaba.

La mayoría de los bugs provienen de los errores cometidos al programar, aunque algunos otros pueden provenir de fallas en el diseño y, los menos, de la conversión que los compiladores hacen del código fuente a código de máquina o la configuración del equipo que se utiliza el ejecutable.

Cuando se construyeron las primeras computadoras, que ocupaban cuartos enteros, la circuitería consistía en miles de elementos tales como relés, resistencias, capacitores y tubos de vacío. Los relés son llaves que abren y cierran circuitos eléctricos cuya activación se hace por medio de una corriente que activa un electroimán.

Las resistencias y tubos de vacío producían calor, lo cual atrae a los insectos. Si los insectos morían entre los dos contactos de un relé, el circuito podía alterarse lo cual producía resultados inesperados e incorrectos.

Esta polilla es posiblemente el primer bug detectado en una computadora. Encontrada por **Grace Hopper** en la Universidad de Harvard, el 9 de Septiembre de 1947.

Errores frecuentes

DIVISIÓN POR CERO

La división por cero no está definida como operación matemática. Es por esto que si intentamos dividir una variable por otra cuyo valor es cero, el programa falla y su ejecución es finalizada automáticamente o, peor aún, continúa con un resultado incierto.

```
dividendo = int(input("Ingrese el numerador: "))
divisor = int(input("Ingrese el denominador: "))
resultado = dividendo / divisor
print(f"El resultado de la operación {dividendo}/{divisor} es {resultado}")
```

El código anterior funcionará, en tanto y en cuanto, denominador sea un número distinto de 0; sino obtendremos un error:

```
Ingrese el numerador: 33
Ingrese el denominador: 0
Traceback (most recent call last):
  File "errores.py", line 4, in <module>
    resultado = numerador / denominador
ZeroDivisionError: division by zero
```

Errores frecuentes

OPERACIONES CON DIFERENTES TIPOS DE DATOS

Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.

```
dividendo = input("Ingrese el numerador: ")
divisor = input("Ingrese el divisor: ")
resultado = dividendo / divisor
print(f"El resultado de la operación {dividendo}/{divisor} es {resultado}")
```

En este caso, no se contempló la conversión del tipo ingresado por el usuario para realizar una operación matemática:

```
Ingrese el numerador: 11
Ingrese el divisor: 5
Traceback (most recent call last):
  File "errores.py", line 4, in <module>
    resultado = dividendo / divisor
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Errores frecuentes

DESBORDAMIENTO DE VARIABLES NUMÉRICAS

Dependiendo del tipo de variable, por ejemplo las numéricas, se tiene un rango posible de valores. Entonces, si se intenta almacenar un valor mayor, se desborda su capacidad y el valor resultante no es posible conseguirlo.

```
valor = 2.0
for i in range(100):
    print(i, valor)
    valor = valor ** 2
```

En el ejemplo propuesto, luego de la novena iteración, ya no se puede representar el valor con el tipo flotante:

```
0 2.0
...
8 1.157920892373162e+77
9 1.3407807929942597e+154
Traceback (most recent call last):
  File "errores.py", line 5, in <module>
    valor = valor ** 2
OverflowError: (34, 'Result too large')
```

Errores frecuentes

PÉRDIDA DE PRECISIÓN

Este error sucede generalmente cuando se efectúa un redondeo (acercar al entero más próximo) o un truncamiento (uso sólo de la parte entera).

Por ejemplo, matemáticamente la diferencia entre 4,9 y 4,845 es 0,055. Pero en Python, obtenemos otro resultado:

```
print(4.9 - 4.845 == 0.055) #False
print(4.9 - 4.845) #0.05500000000000604
```

Esto se debe a que el punto flotante no se puede representar por el número exacto, es solo una aproximación, y cuando se usa en aritmética, éste causa un pequeño desvío.

ACUMULACIÓN DE ERROR DE REDONDEO

Cuando estamos haciendo una secuencia de cálculos sobre una entrada inicial con error de redondeo, debido a una representación inexacta, los errores pueden magnificarse o acumularse.

```
def sumarYRestar(iteraciones):
    resultado = 1
    for i in range(iteraciones):
        resultado += 1/3
    for i in range(iteraciones):
        resultado -= 1/3
    return resultado
print(1 - 1/3 + 1/3) #1.0
print(sumarYRestar(1000)) #1.0000000000000064
```

Errores frecuentes

ERROR DE INDICE

Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.

```
numeros = [11, 22, 33]
for i in range(0, 7):
    print(numeros[i])
```

En el ejemplo anterior, se recorren una cantidad de elementos más grande (7) que la del total de elementos de la lista (3).

Cuando se supera ese tope, ocurre un error de índice:

```
11
22
33
Traceback (most recent call last):
  File "errores.py", line 4, in <module>
    print(numeros[i])
IndexError: list index out of range
```

Tipos de error

ERRORES DE SINTAXIS

Estos errores son seguramente los más simples de resolver, pues son detectados por el intérprete (o por el compilador, según el tipo de lenguaje que estemos utilizando) al procesar el código fuente y generalmente son consecuencia de equivocaciones al escribir el programa. En el caso de Python estos errores son indicados con un mensaje SyntaxError.

Por ejemplo, si trabajando con Python intentamos definir una función y en lugar de def escribimos dev.

ERRORES DE SEMÁNTICA

Se dan cuando un programa, a pesar de no generar mensajes de error, no produce el resultado esperado. Esto puede deberse, por ejemplo, a un algoritmo incorrecto o a la omisión de una sentencia.

ERRORES DE EJECUCIÓN

Estos errores aparecen durante la ejecución del programa y su origen puede ser diverso. En ocasiones pueden producirse por un uso incorrecto del programa por parte del usuario, por ejemplo si el usuario ingresa una cadena cuando se espera un número o realizar una división por cero. Una causa común de errores de ejecución que generalmente excede al programador y al usuario, son los recursos externos al programa, por ejemplo si el programa intenta leer un archivo y el mismo se encuentra dañado o está en uso.

Mitigación de errores

Veamos dos estrategias para atacar el problema de los errores en tiempo de ejecución:

EL ENFOQUE «MIRA ANTES DE SALTAR»

El enfoque **Piensa antes de actuar** (**LBYL**, Look Before You Leap) propone la realización anticipada de pruebas explícitas para determinar si se satisfacen las condiciones que evitan la aparición de errores. De esta manera, se pueden evitar errores y problemas costosos en el futuro y garantizar que el software cumpla con los requisitos y expectativas de los usuarios finales.

```
dividendo = int(input( "Dividendo: "))
divisor = int( input( "Divisor: "))
if divisor != 0:
    cociente = dividendo/divisor
    print(dividendo, "/", divisor, "=", cociente)
else:
    print('Error: el divisor es 0')
```

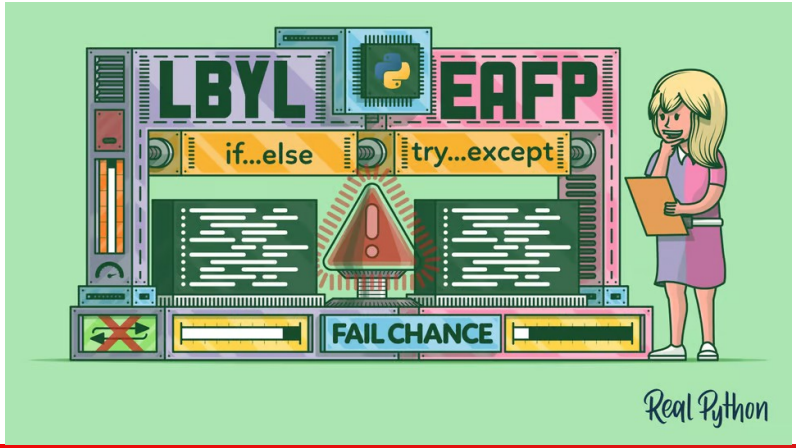
Mitigación de errores

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 directamente la ejecución de las sentencias y, para los casos excepcionales, capturar el error.

El aspecto clave para preferirlo sobre el anterior está en usarlo para tratar los casos excepcionales, que previsiblemente se producirán con baja frecuencia.

En nuestro ejemplo, cualquier usuario sabe que no está definida la división por 0, por lo que no tiene mucho sentido, salvo por despiste, que utilice un denominador 0.



Excepciones

Los errores de ejecución son llamados comúnmente **excepciones**. Durante la ejecución de un programa, si dentro de una función surge una excepción y la función no la maneja, la excepción se propaga hacia la función que la invocó, si esta otra tampoco la maneja, la excepción continua propagándose hasta llegar a la función inicial del programa y si esta tampoco la maneja se interrumpe la ejecución del programa.

Para el manejo de excepciones los lenguajes proveen ciertas palabras reservadas, que nos permiten manejar las excepciones que puedan surgir y tomar acciones de recuperación para evitar la interrupción del programa o, al menos, para realizar algunas acciones adicionales antes de interrumpir el programa.

Python utiliza un objeto especial llamado **excepción** para controlar cualquier error que pueda ocurrir durante la ejecución de un programa. Cuando ocurre un error durante la ejecución de un programa, Python crea una excepción. Si no se controla esta excepción la ejecución del programa se detiene y se muestra el error (traceback):

```
print(1 / 0) # Error al intentar dividir por 0.
Traceback (most recent call last):
ZeroDivisionError: division by zero
```

Tipos de excepciones

Las principales excepciones definidas en Python son:

TypeError

Ocurre cuando se aplica una operación o función a un dato del tipo inapropiado.

ZeroDivisionError

Ocurre cuando se intenta dividir por cero.

OverflowError

Ocurre cuando un cálculo excede el límite para un tipo de dato numérico.

IndexError

Ocurre cuando se intenta acceder a una secuencia con un índice que no existe.

Consultar la documentación de Python para ver la lista completa de excepciones posibles.

Todas la excepciones son hijas de una clase general llamada **Exception**

Sentencia try except

```
try:
    # código bajo prueba
except TipoDeExcepcion:
    # Tratamos la excepción específica si ha sido capturada en el bloque try
except:
    # Tratamos un error genérico si otro tipo de excepción ha sucedido
```

El bloque **try** se ejecuta y, si no se produce ninguna excepción, se ignora 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 tipo de excepción coincide con alguna de las especificadas en un bloque except, tratamos la excepción ejecutando únicamente las sentencias de ese bloque.
- Si el tipo de la excepción no coincide con ninguna de las contempladas se reenvía a otro posible bloque try más externo que contenga a éste o, de no existir, se detiene la ejecución con el mensaje correspondiente.

Ejemplo

Recordemos el error en una división por cero, la excepción es **ZeroDivisionError**

```
Traceback (most recent call last):
total = dividendo / divisor
ZeroDivisionError: division by zero
```

Para evitar que se detenga la ejecución del programa, se captura cualquier error mediante el bloque try-except:

```
dividendo = 5
divisor = 0
try:
    total = dividendo / divisor
except:
    print("Error en la división")
```

Dado que dentro de un mismo bloque try pueden producirse excepciones de distinto tipo, es posible utilizar varios bloques except, cada uno para capturar un tipo distinto de excepción.

Ejemplo

```
try:
    #Aquí ponemos el código que puede lanzar excepciones
except IOError:
    #Entrará aquí en caso que se haya producido una excepción IOError
except ZeroDivisionError:
    #Entrará aquí en caso que se haya producido una excepción ZeroDivisionError
except Exception as e:
    print( "Error: ", e )
    #Entrará aquí en caso que se haya producido una excepción que no corresponda a
    #ninguno de los tipos especificados en los except previos, capturándola en e
```

Es importante destacar que si bien luego de un bloque try puede haber varios bloques except, se ejecutará solamente uno de ellos.

También es posible utilizar una sentencia except capturando las excepciones en forma genérica, mediante la clase **Exception**, en cuyo caso se captura cualquier excepción, sin importar su tipo. La variable e contiene la referencia a la excepción ocurrida.

Cabe destacar, también, que en caso de utilizar una sentencia except general, sin especificar el tipo, la misma debe ser siempre la última de las sentencias except.

Sentencia finally

Opcionalmente, se puede ubicar un bloque **finally** donde se escriben las sentencias de finalización. La particularidad del bloque finally es que se ejecuta siempre, haya surgido una excepción o no. Si hay un bloque except, no es necesario que esté presente el finally, y es posible tener un bloque try sólo con finally, sin except.

```
dividendo = 5
divisor = 0
try:
    total = dividendo / divisor
except ZeroDivisionError:
    total = 0
    print("No se puede dividir por cero")
finally:
    print("Resultado: ", total)
```

Una sentencia 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.

Sentencia else

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 dándole el tratamiento adecuado.

```
try:
    numerador = float(input('Numerador: '))
    denominador = float(input('Denominador: '))
    cociente = numerador/denominador
except ZeroDivisionError:
    print('Error: el denominador es 0')
except ValueError:
    print('Error: el valor introducido no es un número válido')
else:
    print(numerador, "/", denominador, "=", cociente)
finally:
    print('Fin del programa.')
```