\

# Manejo de Errores

Unidad 5

Apunte de cátedra
2do Cuatrimestre de 2023

Pensamiento computacional (90)
Cátedra: Camejo

.UBAXXI

# 1. Manejo de Errores

Cuando trabajamos con datos es una realidad inevitable la cantidad de errores que con seguridad se producirán en su uso y manipulación. La situación es tan común que el proceso de depuración de errores es parte natural de la producción de software.

Para tener una medida de la magnitud de esto, pensemos en el área de Analítica de Datos, un área de aplicación de software de moda.
En esta área normalmente se emplea información con algún nivel de digitalización y seguramente alguna clase -aunque sea básica- de estructuración. Y los recursos que se invierten en mejorar la calidad de la información (parte importante de esto es **la eliminación de errores**) son enormes. Este proceso puede insumir hasta la tercera parte del tiempo en un Proyecto de esta clase.

Se conoce también que **mientras más tarde se identifica algún error, más costos genera el proceso de corrección**: "*Es mejor darse cuenta de que una ventana no cabrá en una pared en los planos de una casa nueva que en la obra, con la pared levantada. Y es preferible esto último a descubrir la realidad ¡cuando la pared ya está pintada!"*

Cuando de Sistemas y Programas hablamos:
- El ingreso de datos es uno de los puntos de mayor frecuencia de errores.
- El ingreso de datos es el lugar más económico donde se pueden detectar y salvar errores.

Para intentar disminuir la tasa de errores en el ingreso podemos:
- Simplificar el formato de los datos o adaptarlos a sistemas de lectura automática (códigos de barra, qr, etc.)
- Validar los ingresos.

Llamamos **Validación de Ingreso o Lectura** al proceso por el que se verifica, y eventualmente rechaza, un ingreso de datos, buscando detectar posibles errores.

¿Hay posibilidades de que nuestro usuario se equivoque al ingresar los datos? **SIEMPRE**. Por múltiples factores. Como regla general, recomendamos que siempre que escribamos un programa asumamos que del otro lado, frente al teclado, habrá alguien que va a cometer algún error.

¿Hay posibilidades de identificar todos los errores que se cometan en el ingreso? Algunos se podrán identificar, otros no. De los que se puedan identificar, no todos tendrán una ecuación costo-beneficio que justifique la validación. Entonces ¿Cuándo Validar?

- ✔ Cuando se pueda
- ✔ Cuando convenga

Si en un ejercicio se pide el ingreso de un número de teléfono, no podremos validar, al menos con las herramientas de las que disponemos en el curso, que el número exista y sea válido; pero si se pide el ingreso

de un simple número negativo, un múltiplo de tres, la edad de una persona, la de un elefante, una fecha, etc., sería interesante pensar en validar que el dato sea correcto. ¿Por qué? Porque son subconjuntos con posibilidades finitas y verificables por medio de expresiones (¿se acuerdan de las expresiones? Las vimos en la guía de estructuras de control).

Si se solicita el ingreso de la edad de una persona viva, seguramente podremos detectar que 300 no es una edad válida, al menos en el siglo XXI, pero no tenemos la posibilidad de saber si la edad (dentro de rango) ingresada de 31 es la edad correcta de esa persona (al menos, no sin cruzar datos). Otro ejemplo es el ingreso de nombres. La verdad es que resulta difícil saber si un nombre está bien escrito o no, y menos si corresponde a la persona en cuestión.

Entonces, se recomienda validar el ingreso siempre que se pueda y cuando la ecuación costo-beneficio sea positiva. Aunque no podamos garantizar la ausencia total de errores, podemos bajar la cantidad que ingresarán y se multiplicarán en nuestro programa.

Para validar un ingreso bastará con escribir una o varias condiciones que nos permitan testear esos datos, o identificar la posible presencia de errores. Y si detectamos un ingreso erróneo, decidir: ¿Qué debemos hacer?

1. ¿Rechazamos el ingreso y paramos la ejecución del programa?
2. ¿Continúa la ejecución como se pueda ignorando el ingreso?
3. ¿Deja de tratarse el caso que contiene un error?
4. ¿Se pide la colaboración del usuario y se solicita un reingreso?

**¡Claro que solicitaremos el reingreso!** ¿Cuántas veces? La idea es ser **tolerantes** con el usuario. Es decir que, cuando solicitemos un ingreso, lo validaremos, y si detectamos un error, **permaneceremos dándole nuevas oportunidades al usuario sin avanzar en el programa.**

Mostraremos un ejemplo de cómo validar números pares:

```
# Valida el ingreso de un número par
par = int(input('Ingresá un número par: '))

while par < 0 or par % 2 != 0: #también te serviría la condición par<0 or par%2
  par = int(input('Ingresá un número par: '))
print('Tu número es',par)
```

Una ejecución ejemplo podría ser:

```
Ingresá un número par: 11
Ingresá un número par: 8
Tu número es 8
>>>
```

⚠️**¡Atención!**

***Retenemos al usuario en el proceso de ingreso cuando se equivoca, por lo tanto la condición para continuar (iterar dentro del bucle de captura) es que se cumpla justo lo que no debe pasar. No debemos escribir la condición de ingreso válido, sino lo contrario. En el ejemplo se testeó que el número ingresado no fuera par.***

***Otra opción es crear la condición que queremos cumplir (que sea par) y agregarle el operador 'not' delante, negando la misma.***

Con esta simple técnica, podemos tener bajo control una gran cantidad de situaciones anómalas y trabajar con datos de buena calidad. Sin embargo hay ingresos de datos que pueden provocar errores fatales y no es nada elegante, y menos profesional, que un programa en producción (en uso real) salte por los aires arrojándonos fuera por un error grave. Muchos de esos errores se pueden evitar con una correcta programación, pero hay otros provocados por el usuario que nos obligan a desarrollar funciones de validación muy complejas o trabajosas, y no es la idea. A estas alturas ya te habrá pasado que cuando solicitas el ingreso de un número entero, por ejemplo, y lo **casteas** en la misma línea. Como en el ejemplo:

```
par=int(input('Ingresá un número par: '))
```

Si el usuario se equivoca y sin querer introduce un dato no compatible con un número entero (ejemplo, una letra), Python enloquece y te acompaña a la salida.

Una forma de manejar elegantemente esta situación sería aceptar el ingreso como string, validar que el valor ingresado sea compatible con el número solicitado y recién castear.

```
# Solicita el ingreso de un número entero
numero = input('Ingresá un número entero: ')
while not numero.isnumeric():
  print('El valor ingresado no es válido.')
  numero = input('Ingresá un número entero: ')
print('Tu número entero es ',int(numero))
```

Miremos una probable ejecución:

```
Ingresá un número entero: l
El valor ingresado no es válido.
Ingresá un número entero: 0.6
El valor ingresado no es válido.
Ingresá un número entero: -7
El valor ingresado no es válido.
Ingresá un número entero: 7
Tu número entero es 7
>>>
```

Note que el programa anterior identifica que los valores ingresados "l" y "0.6" no son números enteros, pero toma como no válido el valor "-7" debido al signo negativo. Este problema de no reconocer números negativos se da con otros métodos de string como **isdecimal**() o **isdigit**().

Por suerte en Python (que siempre cuida la salud de los programadores) tenemos recursos sencillos con los que podemos hacer un manejo elegante de las excepciones o errores.

# 1.1 Manejo de excepciones

Una excepción es un evento que ocurre durante la ejecución de programas que interrumpen el flujo normal de ejecución. En otras palabras, una excepción es un objeto de Python que representa un error y contiene información del mismo.

Seguramente a estas alturas, las excepciones no son nada nuevo, ya que mientras resolvías los ejercicios de la guía te encontraste en muchas oportunidades con un mensaje rojo indicándote que algo no andaba bien. Por ejemplo, si definiste una variable como “names” y luego la llamaste “name”, te apareció lo siguiente:

```
In [1]: name
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In [1], line 1
----> 1 name

NameError: name 'name' is not defined
```

Note que el mensaje contiene:

- la **posición** donde ocurrió el error: en la celda [7], línea 1
- el **tipo de error o nombre de la excepción**: NameError
- una **breve descripción** del error: “el nombre “name” no está definido”, es decir, Python no sabe a qué nos referimos al llamar “name”.

Python viene con varias excepciones incorporadas, así como la posibilidad de crear excepciones autodefinidas. A continuación, se enumeran algunas excepciones más comunes y sus causas:

| **Excepción** | **Causa del error** |
|---|---|
| `AssertionError` | Falla assert. |
| `AttributeError` | Falla la referencia de atributos. |
| `EOFError` | Se genera cuando la función input() alcanza la condición de fin de archivo (EOF). |
| `FloatingPointError` | Falla una operación de punto flotante. |
| `GeneratorExit` | Se genera cuando se llama al método close() de un generador. |
| `ImportError` | Se genera cuando no se encuentra el módulo importado. |

| | |
|---|---|
| `IndexError` | Se genera cuando el índice de una secuencia está fuera de rango. |
| `KeyError` | Se genera cuando una key no se encuentra en un diccionario. |
| `KeyboardInterrupt` | Se genera cuando el usuario presiona la tecla de interrupción (Ctrl+C o Eliminar) |
| `MemoryError` | Se genera cuando una operación se queda sin memoria. |
| `NameError` | Se genera cuando una variable no se encuentra en el ámbito local o global. |
| `OSError` | Se genera cuando la operación del sistema causa un error relacionado con el sistema. |
| `ReferenceError` | Se genera cuando se utiliza un proxy de referencia débil para acceder a un atributo del referente recolectado como basura. |

### 1.1.1. try / except

Para capturar y manejar excepciones, disponemos de la sentencia estructurada **try/except**. Python ejecuta el código e intenta realizar una acción siguiendo la instrucción **try** y si tiene éxito, el programa prosigue con normalidad. Sin embargo, si en el intento emerge un error, la sentencia lo captura y lo trata de una manera conveniente en la cláusula **except**. En otras palabras, el código que sigue a la instrucción **except** es la respuesta del programa frente a cualquier excepción de la cláusula try.

```
try:
  # Corre este código, algo puede fallar.
except:
   # Se ejecuta este código cuando ocurre una excepción en el bloque de try.
```

Mostraremos cómo resolver el problema del ingreso de un entero utilizando esta facilidad:

```
numero = None
while type(numero) is not int:
  try:
    numero = input('Ingresá un número entero: ')
    numero = int(numero)
    print('Tu número entero es ', int(numero))
  except ValueError:
    print('El valor ingresado no es válido.')
```

Una ejecución posible sería:

```
Ingresá un número entero: l
El valor ingresado no es válido.
Ingresá un número entero: 0.6
El valor ingresado no es válido.
Ingresá un número entero: -7
Tu número entero es -7
>>>
```

Note que el programa anterior, no arrojó un mensaje de error **ValueError** al ingresar, por ejemplo, el valor "l", sino que respondió como se indicó en la cláusula **except**.

⚠️**¡Atención!**

***Los errores deben escribirse exactamente como los identifica Python. En el ejemplo, el Código de error que pretendemos capturar es: ValueError. Debe anotarse tal cual.***

Varias cosas para tener en cuenta:

- ✔ En el ejemplo anterior, el manejo de excepciones está dentro de un bucle para darle al usuario tantos intentos como necesite. Por eso solo salimos del bucle cuando se puede ejecutar completamente el bloque vinculado a la cláusula **try**.
- ✔ Existen situaciones en la que múltiples excepciones son posibles. En esos casos, si decidimos actuar sólo con algunos, debemos consignarlos, de lo contrario capturará todos los errores posibles. Veamos los siguientes esquemas, donde existen Excepción_1 y Excepción_2 para notar sus diferencias:

**Caso 1:** No se indica ninguna excepción en particular

```
try:
  # Corre este código, algo puede fallar.
except:
  # Se ejecuta este código cuando ocurre una excepción en el bloque de
  try.
```

**Caso 2:** Se indican distintas excepciones en un mismo bloque except.

```
try:
  # Corre este código, algo puede fallar.
except Excepción_1, Excepción_2:
  # Se ejecuta este código cuando ocurre cualquiera de las dos
excepciones.
```

**Caso 3:** Se indican distintas excepciones en bloques diferentes de except.

```
try:
  # Corre este código, algo puede fallar.
except Excepción_1:
  # Se ejecuta este código cuando ocurre Excepcion_1.
except Excepción_2:
  # Se ejecuta este código cuando ocurre Excepcion_2.
```

⚠️ **¡Atención!**
***También podríamos querer "no hacer nada" frente a una excepción, pero evitando que el programa se bloquee. Para ello, existe pass.***

```
try:
  # Corre este código, algo puede fallar.
except:
  pass # No hagas nada.
```

Quizá pass puede parecer poco útil en un primer momento, pero puede volverse tu mejor amigo. Por ejemplo, mientras estás pensando la estructura del código, y querés ir probando algunas líneas cuando aún no está terminado, y evitar que el programa se interrumpa.

```
if numero % 15 == 0:
  pass # Completar.
elif numero % 3 == 0:
  pass # Completar.
elif numero % 5 == 0:
  pass # Completar.
else:
  pass # Completar.
```

### 1.1.2. Cláusula try con else

Además de las cláusulas try y except, también podemos usar una cláusula **else** cuando deseamos ejecutar un bloque de código específico sólo si no ocurre ninguna excepción en el bloque try.

```
try:
  # Corre este código, algo puede fallar.
except:
  #  Se ejecuta este código cuando ocurre una excepción en el bloque de try.
else:
  # Se ejecuta este código cuando no existe una excepción.
```

Seguramente, te estarás preguntando ¿No podemos simplemente poner más código en el bloque try? Sí, podemos, pero aquí hay un par de razones por las que esto puede ser una muy mala idea:

1. Es posible que nos encontremos con excepciones que no anticipamos anteriormente: cuanto más código se encuentre en la cláusula try, más probable será que esto suceda.
2. Reduce la legibilidad: la cláusula try expresa lo que esperamos que falle, y la cláusula except, las formas en que planeamos manejar las fallas específicas en el código. Cuanto más código se encuentra en la cláusula try, menos claro es lo que realmente estamos intentando, y eso puede hacer que toda la estructura sea más difícil de entender.

   Es por esto que el código en el bloque **try** debe ser lo más pequeño posible, abarcando únicamente a las líneas de código que podrían resultar en un problema (en un error a atajar).

### 1.1.3. Cláusula try con finally

En ocasiones queremos ejecutar alguna acción, incluso si ocurrió un error en un programa o no (como por ejemplo, cerrar un archivo si es que ocurrió un error mientras lo leíamos). En Python, podemos realizar tales acciones usando **finally**:

```
try:
  # Corre este código, algo puede fallar.
except:
  #  Se ejecuta este código cuando ocurre una excepción en el bloque de try.
else:
  # Se ejecuta este código cuando no existe una excepción.
finally:
  # Siempre se ejecuta y bloquea el código.
```

La cláusula finally, es muy especial, ya que:

- Siempre se ejecutará.
- Finaliza el programa.
- Si se produce una excepción no controlada, no importa, ya que finally ejecutará su código antes de que eso ocurra.

Si tenemos un return en el bloque try, finally interrumpe ese retorno para ejecutar su propio código primero, como se puede observar en el siguiente ejemplo:

```
def probar_finally():
    try:
        return 2
    finally:
        print("Código del bloque finally")

probar_finally()
```

Miremos su ejecución:

```
Código del bloque finally

2
>>>
```

Esta propiedad es extremadamente útil para cualquier situación en la que se requiera una **limpieza vital** después de una operación, como por ejemplo, cuando se trabaja con archivos. Si encontramos algún problema mientras procesamos los datos de un archivo y aún así queremos cerrarlo al terminar, finally nos asegura que esto suceda. Si bien, finally es muy utilizado en código más avanzado, sigue siendo una herramienta importante que debemos conocer en esta etapa.

### 1.1.4. Cláusula raise

En Python, raise nos permite lanzar una excepción si se produce una cierta condición. La declaración raise es útil en situaciones en las que necesitamos generar una excepción personalizada como al recibir datos incorrectos o cualquier error de validación.

```
x = 10

if x > 5:
  raise Exception(f'No debe exceder 5. El valor es: {x}').
```

Miremos su ejecución:

```
Exception: No debe exceder 5. El valor es: 10
>>>
```

No es correcto lanzar una excepción para atajarla inmediatamente con un **except**. Lanzar una excepción se suele utilizar para "elevar" el error a una función previa y manejar el error desde ahí. Hay que usar **raise** de forma razonable.

### 1.1.5. Ejemplo

Para entender mejor cómo se utilizan las declaraciones antes vistas, vamos a resolver un ejercicio simple con fines didácticos, cuya consigna es la siguiente:

**Crear una función con parámetros x e que calcule el cociente entre ambos. Deberá usar:**

- **try para probar el cálculo**
- **except en caso de que ocurra una excepción con los números ingresados**
- **else para brindar la respuesta en un formato adecuado**

- **finally para mostrar al usuario que este bloque siempre se ejecuta.**

Para resolver el ejercicio, necesitamos intentar realizar un cálculo (en este caso una división) e identificar cualquier excepción que pueda surgir cuando el usuario proporciona valores no válidos. En este caso, un valor no válido sería un denominador 0, con el que obtendremos un **ZeroDivisionError**, por lo que esta es la excepción que debemos considerar con nuestra cláusula except.

```
def calcular_division(x, y):
    try:
        cociente = x / y
    except ZeroDivisionError:
        pass
    else:
        pass
    finally:
        pass
```

Dentro de la cláusula except vamos a escribir un mensaje para informarle al usuario que los datos ingresados eran inválidos. Además, devolveremos el cociente en la cláusula else y agregaremos el mensaje indicado en la consigna, en finally.

```
def calcular_division(x, y):
    try:
        cociente = x / y
    except ZeroDivisionError:
        print("No se puede dividir por cero.")
    else:
        print("El cociente es: ", cociente)
    finally:
        print("El bloque finally siempre se ejecuta")
```

Observemos qué ocurre si asignamos distintos valores a la función:

**Caso 1: x = 15, y = 5**

```
El cociente es:  5.0
El bloque finally siempre se ejecuta

>>>
```

En este caso, Python intenta hacer la división 15 / 5, y como es posible, ejecuta el bloque else. El bloque finally, se ejecuta siempre.

**Caso 2: x = 15, y = 0**

```
No se puede dividir por cero.
El bloque finally siempre se ejecuta

>>>
```

En este caso, Python intenta hacer la división 15 / 0 pero aparece el mensaje de error ZeroDivisionError, que es capturado por la cláusula except que le indica cómo tratarlo: en este caso, mostrándole al usuario el mensaje “No se puede dividir por cero”. El bloque finally, se ejecuta siempre.

## 1.2. Aclaraciones finales

Es importante notar que, independientemente de cómo se ejecute el bloque try-except (y sus variaciones), al finalizar, el programa sigue con su flujo de control (siempre y cuando el programa no “explote”).
Vemos que este código:

```
def calcular_division(x, y):
  print(f"\nDividiendo {x} sobre {y}")
  try:
    cociente = x / y
  except ZeroDivisionError:
    print("No se puede dividir por cero.")
  else:
    print("El cociente es: ", cociente)
  finally:
    print("El bloque finally siempre se ejecuta")

  print("Este texto siempre se va a imprimir después del bloque try-except")


calcular_division(15, 0)
calcular_division(15, 5)
```

Tiene este comportamiento:

```
Dividiendo 15 sobre 0
No se puede dividir por cero.
El bloque finally siempre se ejecuta
Este texto siempre se va a imprimir después del bloque try-except

Dividiendo 15 sobre 5
El cociente es:  3.0
El bloque finally siempre se ejecuta
Este texto siempre se va a imprimir después del bloque try-except
```

Como vemos, el manejo de errores con try-except es una forma en la cual podemos hacer nuestro programa mucho más robusto y resistente a los errores.

Al mismo tiempo, no todas de nuestras validaciones requieren de un try-except: hay validaciones que se pueden hacer tranquilamente con las herramientas de control que ya conocemos: if, elif y else.