



# Manejo de errores y archivos

Manejo de Errores y Excepciones

***Utilizar sentencias de captura y generación de excepciones para el control del flujo de un programa acorde al lenguaje Python.***

***Codificar un programa que lee y escribe archivos utilizando el lenguaje Python para resolver un problema.***

- Unidad 1:  
Introducción a la programación orientada a objetos con Python
- Unidad 2:  
Tipos de métodos e interacciones entre objetos
- Unidad 3:  
Herencia y polimorfismo
- Unidad 4:  
Manejo de errores y archivos



Te encuentras aquí



## ¿Qué aprenderás en esta sesión?

- *Reconoce los mecanismos y formas del lenguaje Python para generación de errores y su aplicación.*
- *Codifica un programa Python que permite controlar las excepciones para resolver un problema.*
- *Codificar un programa que lanza excepciones personalizadas para resolver un problema.*

¿Qué entendemos por  
errores y excepciones?



# **`/* Tipos de errores */`**

# Errores

1. El primer caso corresponde a un error que se produce **durante la ejecución del código**, llamados también "**excepciones**".
2. El segundo caso corresponde a errores que **impiden que el programa se ejecute**, y corresponde a un "**error de sintaxis**".

En cualquiera de los dos casos, el programa no podrá ejecutarse completamente.

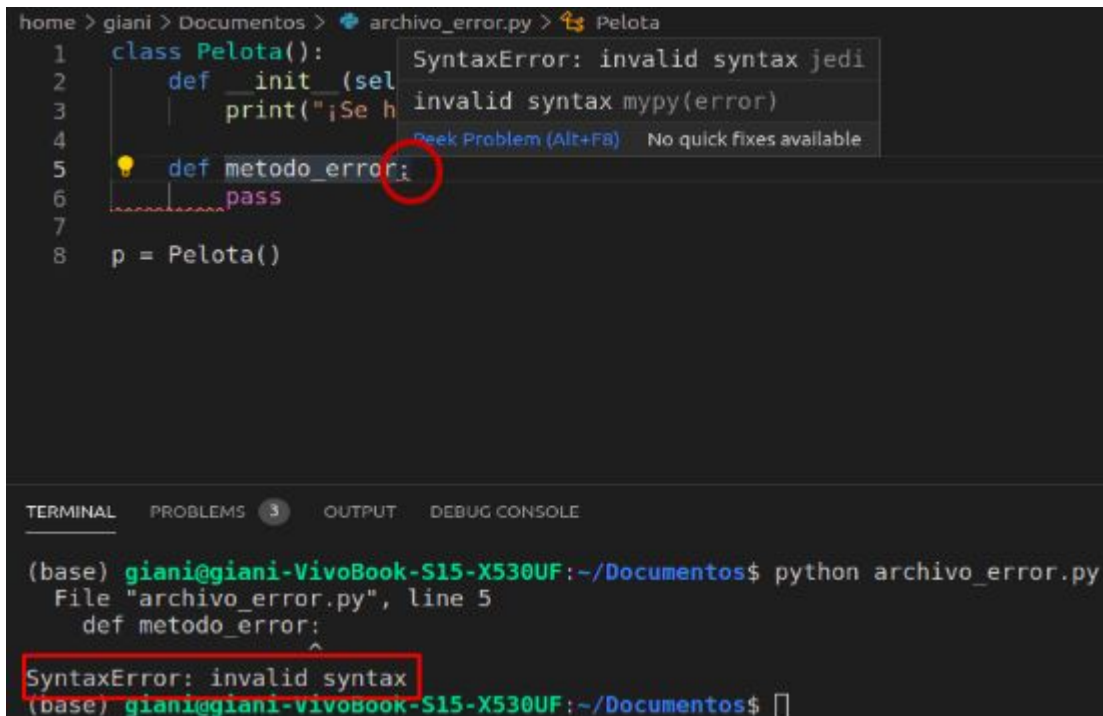


*Puedes revisar la documentación oficial de Python sobre errores y excepciones en [este enlace](#).*

# Errores de sintaxis

(errores de interpretación)

- Consisten en escribir código Python sin seguir la gramática del lenguaje.
- Al tener un error de sintaxis, las líneas anteriores a la ubicación del error no son ejecutadas al momento de ejecutar el código.



```
home > giani > Documentos > archivo_error.py > Pelota
1 class Pelota():
2     def __init__(self):
3         print("¡Se ha creado la pelota!")
4
5     def metodo_error:
6         pass
7
8 p = Pelota()
```

SyntaxError: invalid syntax  
invalid syntax mypy(error)  
Peek Problem (Alt+F8) No quick fixes available

TERMINAL PROBLEMS 3 OUTPUT DEBUG CONSOLE

```
(base) giani@giani-VivoBook-S15-X530UF:~/Documentos$ python archivo_error.py
File "archivo_error.py", line 5
    def metodo_error:
                    ^
SyntaxError: invalid syntax
(base) giani@giani-VivoBook-S15-X530UF:~/Documentos$
```

# Errores de ejecución

(excepciones)

- No impide que el programa comience su ejecución, ya que en este tipo de error la sintaxis del código es correcta.
- Durante la ejecución del programa se produce un error que impide continuarla. Las líneas anteriores a la ubicación de la instrucción que produce la excepción sí son ejecutadas, mientras que la línea donde se ha producido la excepción (y todas las posteriores) no son ejecutadas.
- No se desencadenan en todos los escenarios, sino que se producirán dependiendo de los valores contenidos en variables o argumentos.

```
home > giani > Documentos > archivo_error.py > ...
1  print("Comenzando Programa")
2  edad: int = int(input("Ingrese su edad:\n"))
3  nombre: str = input("Ingrese su nombre:\n")
4  print([edad / nombre])

TERMINAL  PROBLEMS 15  OUTPUT  DEBUG CONSOLE

(base) giani@giani-VivoBook-S15-X530UF:~/Documentos$ python archivo_error.py
Comenzando Programa
Ingrese su edad:
Miau
Traceback (most recent call last):
  File "archivo_error.py", line 2, in <module>
    edad: int = int(input("Ingrese su edad:\n"))
ValueError: invalid literal for int() with base 10: 'Miau'
```



# Errores lógicos

- Son los más difíciles de detectar ya que no impiden la ejecución del programa, ni tampoco producen excepciones.
- A menudo sólo se reconocen porque generan un resultado no esperado o incorrecto, el cual es detectado por el programador (por uso directo del programa, o mediante pruebas automatizadas) o, en el peor de los casos, por el usuario.

```
home > giani > Documentos > archivo_error.py > ...
1  num_1 = input("Ingrese primer número:\n")
2  num_2 = input("Ingrese segundo número:\n")
3  print(num_1 + num_2)
4

TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE

(base) giani@giani-VivoBook-S15-X530UF:~/Documentos$ python archivo_error.py
Ingrese primer número:
15
Ingrese segundo número:
30
1530
(base) giani@giani-VivoBook-S15-X530UF:~/Documentos$
```

**/\* Manejo de excepciones \*/**

# Manejo de excepciones

Para impedir errores de ejecución, o incluso para controlar el flujo de un programa, se debe manejar las excepciones.

En Python, esto es posible mediante el uso de un bloque de código compuesto por tres cláusulas, que son **try/except/finally**.

La importancia de manejar excepciones en Python es que de esta forma:

- Impide la interrupción de un programa frente a situaciones no controladas.
- Permite crear flujos alternativos de ejecución.



# ¿Qué es una excepción?

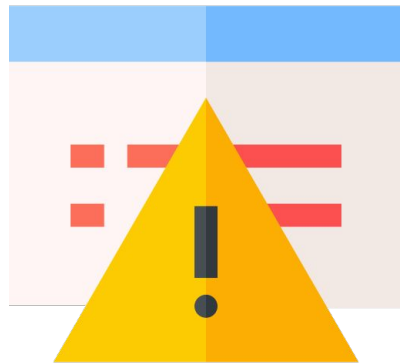
- Corresponde a un error de ejecución, el cual tiene un tipo asociado.
- El tipo de la excepción corresponde al nombre de la clase de la excepción, lo cual es válido para todas las excepciones predefinidas por el intérprete de Python (conocidas como **excepciones built-in**).
- También es posible definir excepciones propias, las cuales no necesariamente deben tener un tipo específico asociado, pero es altamente recomendado hacerlo.
- Todas las excepciones built-in son **clases derivadas de la clase base BaseException**, o de otras clases base derivadas de ella, mientras que todas las excepciones definidas por el usuario son derivadas de la clase base Exception.

# Tipos de excepciones

Tipos de Excepción	Error que la produce
AttributeError	Error en una referencia o asignación a un atributo.
ImportError	Importar un módulo no encontrado.
IndexError	Referenciar un índice fuera del rango posible en una secuencia.
KeyError	Referenciar una clave no existente en un diccionario.
MemoryError	Ejecución que consume toda la memoria disponible.
NameError	Referenciar una variable no encontrada.
TypeError	Aplicar una operación o función a un objeto de tipo incorrecto.
ZeroDivisionError	Dividir por cero.

# Sentencia try/except

- Permite gestionar el manejo de excepciones, de forma que la ejecución del programa no se vea interrumpida aún cuando se produzcan excepciones.
- Se debe incluir tanto 1 declaración try, como al menos 1 declaración except.
- En caso de producirse una excepción en el código definido dentro de try, la ejecución del código continuará en el código contenido en el primer except que capture el tipo de la excepción generada.



# Escribir un bloque `try/except` en Python

1. Declarar la cláusula `try`, mediante la palabra reservada “try” seguida de dos puntos “:”.
2. Las cláusulas `except` se declaran al mismo nivel de la sentencia try asociada a ellos. Adicionalmente, a continuación se puede también declarar una variable local que contiene la instancia de la excepción generada mediante la palabra reservada “as”.
3. Añadir dos puntos “:” al final de la línea, y en la línea siguiente, con una indentación de 4 espacios, se declara el código que se desea ejecutar al producirse el tipo de excepción declarada.

# Escribir un bloque try/except en Python

## *Ejemplo*

```
consultar = True

while consultar:
    try:
        edad = int(input("Ingrese su edad:\n"))
        consultar = False
    except ValueError:
        print("Debe ingresar un número")
```



# Captura de múltiples excepciones

1. Se debe generar una cláusula **except** por cada tipo de excepción para la cual se desea controlar un flujo.
2. Una vez que se genere una excepción dentro de la cláusula **try**, el intérprete de Python revisará, en orden de escritura, los tipos de excepciones dentro de las cláusulas **except** que tiene asociadas.
3. Se ejecutará el código declarado dentro de la primera cláusula **except** que cumple con el tipo de excepción de la excepción generada.
4. Si dentro de las cláusulas **except** declaradas no se ha incluido el tipo de la excepción generada dentro de la cláusula **try**, entonces la excepción no será capturada, provocando la interrupción del programa.

# Captura de múltiples excepciones

## Ejemplo

Si se llegara a generar cualquier otro tipo de excepción que no está manejado por las cláusulas anteriores, entonces se ejecutará la última cláusula `except` declarada, la cual capturará cualquier otro tipo de excepción, ya que no especifica ningún tipo para ingresar en ella.

```
consultar = True

while consultar:
    try:
        edad = int(input("Ingrese su edad:\n"))
        divisor = int(input("Ingrese número para dividir su edad:\n"))
        print(edad / divisor)
        consultar = False
    except ValueError:
        print("Debe ingresar un número")
    except ZeroDivisionError:
        print("El N° por el cual desea dividir no puede ser cero")
    except Exception as e:
        print(f"ERROR: {e}")
    except:
        print("ERROR SIN INFORMACIÓN")
```

# Lanzamiento de excepciones

Dado que el control de excepciones permite controlar el flujo de un programa, puede darse el caso de que el programador desee forzar que se genere una excepción en determinadas circunstancias.

Esto es posible mediante la declaración **raise**, la cual a continuación requiere que se **especifique el tipo de excepción a generar**. Para ello, se debe indicar el tipo de la excepción o generar una instancia de ella.

En caso de que se genere una instancia, se puede ingresar como argumento de su constructor el valor asociado de la excepción.

# Lanzamiento de excepciones

## Ejemplo

```
consultar = True

while consultar:
    try:
        edad = int(input("Ingrese su edad:\n"))
        if edad < 0:
            raise Exception("Edad debe ser un N° positivo.")
        divisor = int(input("Ingrese número para dividir su edad:\n"))
        print(edad / divisor)
        consultar = False
    except ValueError:
        print("Debe ingresar un número")
    except ZeroDivisionError:
        print("El N° por el cual desea dividir no puede ser cero")
    except Exception as e:
        print(f"ERROR: {e}")
```

# Excepciones definidas por el usuario

Para definir una excepción, se debe **crear una clase que derive de la clase base Exception** (o de alguna bajo su jerarquía). Como buena práctica, se sugiere crear una clase base para excepciones definidas, la cual luego será heredada por cada clase de excepciones propias que se desea generar.

Comúnmente, las clases utilizadas para definir excepciones terminan su nombre en “**Error**”.

*Si quieres conocer en mayor detalle cómo definir excepciones, puedes consultar la documentación oficial de Python en [este enlace](#).*

# Excepciones definidas por el usuario

## Ejemplo

El valor asociado de una excepción corresponde en realidad a una lista de argumentos (**\*args**), por lo que se puede especificar cualquier atributo (o atributos) en el constructor.

```
class Error(Exception):  
    pass  
  
class EdadError(Error):  
    def __init__(self, mensaje, edad):  
        self.mensaje = mensaje  
        self.edad = edad
```

# Excepciones definidas por el usuario

## Ejemplo

Se ha reemplazado el código del título anterior, de forma tal de que se lance la excepción propia **EdadError**, en lugar de la excepción base **Exception**.

Al ingresar una edad, por ejemplo, de -1, se mostrará en pantalla el siguiente mensaje: "La edad '-1' no es válida. Debe ser un N° positivo".

```
consultar = True

while consultar:
    try:
        edad = int(input("Ingrese su edad:\n"))
        if edad < 0:
            raise EdadError("Debe ser un N° positivo.", edad)
        divisor = int(input("Ingrese número para dividir su edad:\n"))
        print(edad / divisor)
        consultar = False
    except ValueError:
        print("Debe ingresar un número")
    except ZeroDivisionError:
        print("El N° por el cual desea dividir no puede ser cero")
    except EdadError as e:
        print(f"La edad '{e.edad}' no es válida. {e.mensaje}")
    except Exception as e:
        print(f"ERROR: {e}")
```

**/\* Acciones de limpieza \*/**



# Acciones de limpieza con **finally**

- Además de **except**, la declaración **try** tiene la cláusula opcional **finally**.
- Su propósito es **definir acciones de limpieza que se ejecutarán al final**, al completar todo el bloque **try**.
- Permite ejecutar código asociado a **try**, independiente de si se produce o no una excepción, o en caso que se produzca una nueva excepción dentro de una cláusula **except**, ya que lo declarado en **finally** siempre se ejecutará al final de todo lo incluido y asociado a **try**.

*Si quieres revisar en mayor detalle cómo funciona **finally**, puedes consultar la documentación oficial de Python en [este enlace](#).*

# Acciones de limpieza con `finally`

## Ejemplo

```
intentos = 0

while intentos <= 3:
    try:
        edad = int(input("Ingrese su edad:\n"))
        if edad < 0:
            raise EdadError("Debe ser un N° positivo.", edad)
        divisor = int(input("Ingrese número para dividir su edad:\n"))
        print(edad / divisor)
    except ValueError:
        print("Debe ingresar un número")
    except ZeroDivisionError:
        print("El N° por el cual desea dividir no puede ser cero")
    except EdadError as e:
        print(f"La edad '{e.edad}' no es válida. {e.mensaje}")
    except Exception as e:
        print(f"ERROR: {e}")
    finally:
        intentos += 1
```

Una cosa a tener en cuenta es que también es posible escribir un bloque try/finally sin cláusulas except. Es decir, si se añade la cláusula finally, la cláusula except pasa a ser opcional.



# Acciones de limpieza predefinidas

- Algunos objetos definen acciones de limpieza estándar para llevar a cabo cuando el objeto ya no es necesario, independientemente de que las operaciones sobre el objeto hayan sido exitosas o no.
- Un ejemplo de ellos es la apertura de archivos mediante **open()**, lo cual se encuentra claramente explicado en la documentación oficial de Python en [este enlace](#).
- Alternativamente, también es posible usar la cláusula **else**, la cual se ubica entre **except** y **finally**, y se ejecuta siempre que no ocurra ninguna excepción en el código definido en **try** o en **except**.

# Acciones de limpieza predefinidas

*Resumen de las cláusulas involucradas en el bloque try/except*

Cláusula	Obligatorio	Cuándo se ejecuta
try	□	Siempre, contiene el código del cual se desea controlar las excepciones.
except	□ (Sin finally) ✗ (Con finally)	Solo en caso de que se produzca una excepción del tipo especificado (Si no se especifica tipo de la excepción, se ejecutará frente a cualquier excepción). Puede haber más de 1.
else	✗	En caso de que no se produzca ninguna excepción.
finally	✗	Al final de todo el bloque try/except, haya ocurrido o no una excepción.

Normalmente, todas las excepciones se definen en un mismo módulo, por lo que deben importarse en caso de que no estén definidas en el espacio de trabajo donde se desean utilizar.



# Ejercicio guiado

## "Validación de datos"



# Validación de datos

Usted forma parte de una empresa que se encuentra desarrollando una aplicación de calendario y agenda. Se le ha solicitado, mediante una aplicación de consola Python, crear un pequeño demo que ejecute el algoritmo para validar el ingreso de los datos necesarios para crear una reunión previo a crearla. Por ahora se le pide que considere que una reunión debe tener los siguientes atributos y validaciones:

- Título: Cadena de texto. No puede tener más de 150 caracteres.
- Hora: Cadena de texto. El contenido debe estar en formato "HH:MM:SS".

Desde su empresa le han solicitado que maneje las validaciones de datos ingresados mediante uso de excepciones propias.





# Validación de datos

## Solución

### Paso 1

En un archivo **error.py**, definir clase Error, que deriva de Exception, sin implementación. En el mismo archivo, definir a continuación la clase HoraError, que deriva de Error, sin implementación.

```
class Error(Exception):  
    """Clase Base Excepciones"""  
    pass  
  
class HoraError(Error):  
    pass
```



# Validación de datos

## Solución

### Paso 2

En el mismo archivo, definir la clase `LargoTextoError`; Sobreescrba el constructor, admitiendo los parámetros `mensaje`, `texto` (opcional) y `largo` (opcional), los cuales debe asignar a atributos de instancia. En el caso del `texto`, acortar en caso de que supere los 50 caracteres.

```
class LargoTextoError(Error):
    def __init__(self, mensaje: str, texto: str = None,
                  largo: int = None) -> None:
        self.mensaje = mensaje
        self.texto = (f"{texto[:50]}..." if texto is not
None
                     and len(texto) > 50 else texto)
        self.largo = largo
```



# Validación de datos

## Solución

### Paso 3

En el mismo archivo, sobrecargue el método `__str__`. En caso de que no se haya ingresado texto ni largo, retornar el método de la clase padre. En caso contrario, según los valores ingresados, construir mensaje de retorno.

```
def __str__(self) -> str:
    if self.texto is None and self.largo is None:
        return super().__str__()
    else:
        ret = f"{self.mensaje}."
        if self.texto != None:
            ret += f" Texto ingresado: {self.texto}."
        if self.largo != None:
            ret += f" Máximo {self.largo} caracteres permitidos."
        return ret
```



# Validación de datos

## *Solución*

### Paso 4

En un archivo **reunion.py**, definir clase Reunion, con los atributos solicitados.

```
class Reunion():  
    def __init__(self, titulo: str, hora: str) -> None:  
        self.titulo = titulo  
        self.hora = hora
```



# Validación de datos

## Solución

### Paso 5

En un archivo **demo.py**, importar las clases HoraError, LargoTextoError y Reunion. Importar también el módulo re, y definir las variables titulo y hora asignando None. Definir también una variable time\_re con el valor indicado.

```
from error import HoraError, LargoTextoError
from reunion import Reunion
import re

titulo = None
hora = None
time_re = "^((?:((?:[01]?\\d|2[0-3]))?:)?([0-5]?\\d):)?([0-5]?\\d)$"
```



# Validación de datos

## Solución

### Paso 6

Iniciar un ciclo while infinito y dentro de este crear un bloque try/except. La cláusula except debe manejar todas las excepciones de tipo Exception. En ella, mostrar la instancia de la excepción y agregar la instrucción continue. Agregar también una cláusula else, y en ella agregar la instrucción break.

```
while True:
    try:
        pass
    except Exception as e:
        print(f"\n{e}\n")
        continue
    else:
        break
```



# Validación de datos

## Solución

### Paso 7

Luego, dentro de la cláusula try, solicitar el título, en caso de que este no tenga valor o su largo supere 150. Luego de ser ingresado, en caso de que su largo sea superior a 150, lanzar una excepción de tipo LargoTextoError.

```
if titulo is None or len(titulo) > 150:
    titulo = input("\nIngrese título de la reunión"
                  " (Máximo 150 caracteres):\n")

    if len(titulo) > 150:
        raise LargoTextoError(
            "Título de la reunión excede máximo de caracteres",
            titulo, 150)
```



# Validación de datos

## Solución

### Paso 8

A continuación, también dentro de la cláusula try, solicitar la hora, en caso de que no tenga valor o no tenga el formato permitido. Luego de ser ingresada, en caso de que no tenga el formato solicitado, lanzar una excepción de tipo HoraError.

```
if hora is None or re.search(time_re, hora) is None:
    hora = input("\nIngrese hora de la reunión"
                " (Formato: HH:MM:SS):\n")

    if re.search(time_re, hora) is None:
        raise HoraError("Formato de Hora debe ser HH:MM:SS.")
```





# Validación de datos

## *Solución*

### Paso 9

A continuación del ciclo while, crear una instancia de Reunion con el título y la hora ingresados.

```
r = Reunion(titulo, hora)
print("\nReunión creada correctamente.")
```



¿Qué diferencia un  
error de sintaxis de un  
error de ejecución?



¿Qué error produce una  
excepción de tipo `IndexError`?



¿Qué sentencia en bloque  
permite el manejo de  
excepciones en Python?





## Próxima sesión...

- *Desafío evaluado.*

**{desafío}**  
**latam\_**

*Academia de  
talentos digitales*

