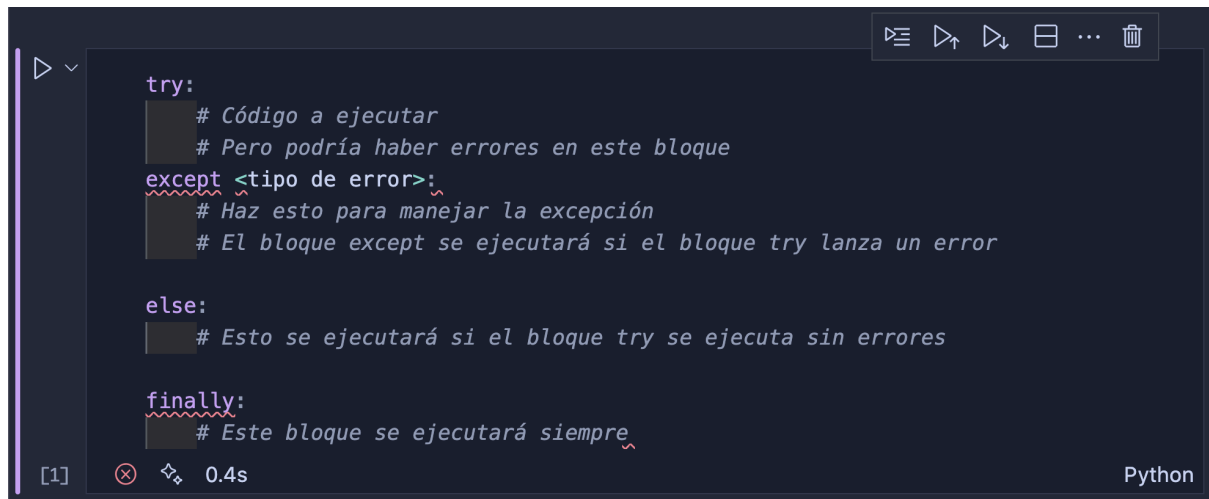




Sentencias Try y Except de Python: Cómo manejar excepciones en Python

Al programar en Python algunas veces podemos anticipar errores de ejecución, incluso en un programa sintáctica y lógicamente correcto, pueden llegar a haber errores causados por entrada de datos inválidos o inconsistencias predecibles. En Python, podemos usar los bloques try y except para manejar estos errores como excepciones.

La sintaxis correcta sería la siguiente:



```
try:
    # Código a ejecutar
    # Pero podría haber errores en este bloque
except <tipo de error>:
    # Haz esto para manejar la excepción
    # El bloque except se ejecutará si el bloque try lanza un error

else:
    # Esto se ejecutará si el bloque try se ejecuta sin errores

finally:
    # Este bloque se ejecutará siempre
```

[1] 0.4s Python

Veamos el uso de cada uno de estos bloques:

- El bloque try es el bloque con las sentencias que quieres ejecutar. Sin embargo, podrían llegar a haber errores de ejecución y el bloque se dejará de ejecutarse.
- El bloque except se ejecutará cuando el bloque try falle debido a un error. Este bloque contiene sentencias que generalmente nos dan un contexto de lo que salió mal en el bloque try.
- Siempre deberías de mencionar el tipo de error que se espera, como una excepción dentro del bloque except dentro de <tipo de error> como lo muestra el ejemplo anterior.



- Podrías usar `except` sin especificar el <tipo de error>. Pero no es una práctica recomendable, ya que no estarás al tanto de los tipos de errores que puedan ocurrir.

Cuando el código dentro del bloque `try` se ejecuta, pueden surgir diversos errores. Es en ese momento donde entra en acción el `except`, permitiendo manejarlos adecuadamente

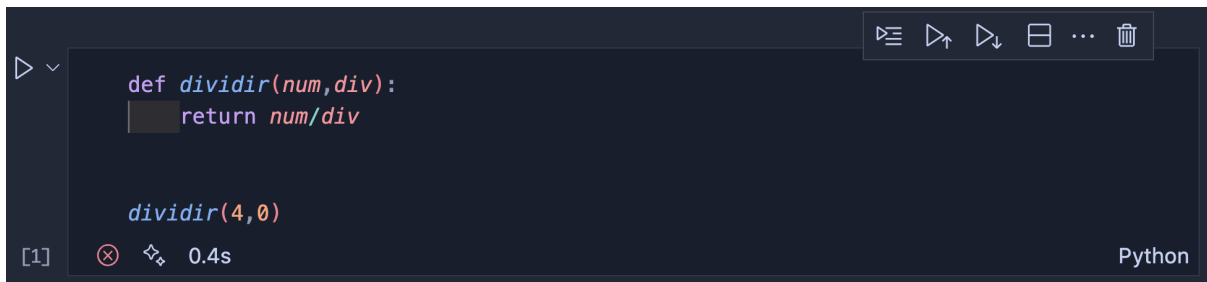
- El bloque `else` se ejecutará, si solo si el bloque `try` se ejecuta sin errores. Esto puede ser útil cuando quieras continuar el código del bloque `try`. Por ejemplo, si abres un archivo en el bloque `try`, podrías leer su contenido dentro del bloque `else`.
- El bloque `finally` siempre es ejecutado sin importar que pase en los otros bloques, esto puede ser útil cuando quieras liberar recursos después de la ejecución de un bloque de código, (`try`, `except` o `else`).

Estos bloques, son totalmente opcionales, podrías trabajar tranquilamente utilizando solo el `try` y `except`.



Manejo de Errores típicos en Python

1- ZeroDivisionError (División por cero)

Ejemplo de función:



```
def dividir(num,div):  
    return num/div  
  
dividir(4,0)
```

[1]   0.4s Python

Al llamar a la función y pasarle como parámetro un 0, tendríamos el siguiente error por consola:



```
... -----
ZeroDivisionError                                Traceback (most recent call last)
Cell In[1], line 5
      1 def dividir(num,div):
      2     return num/div
----> 5 dividir(4,0)

Cell In[1], line 2
      1 def dividir(num,div):
----> 2     return num/div

ZeroDivisionError: division by zero
```

Podemos tratar esta división entre cero como una excepción, haciendo lo siguiente:

1. Desde el bloque **try** llamamos a la función `dividir()`.
2. En el bloque **except** tendremos una excepción en caso de que división sea igual a cero.
3. En este ejemplo se hace una excepción a `ZeroDivisionError` (el tipo de error) el cual se especifica en **except**
4. Cuando se detecte el error **ZeroDivisionError** se ejecutará el bloque **except** donde pondremos un mensaje informando que se trató de dividir entre cero.

Aplicando los ítems, nos queda de la siguiente forma:



```
def dividir(num,div):
    return num/div

try:
    res = dividir(4, 0)
    print(res)
except ZeroDivisionError:
    print("Trataste de dividir entre cero :( ")

[2] ✓ 0.0s Python
```

... Trataste de dividir entre cero :(



2- TypeError (Error de tipo de dato)

En este ejemplo, tenemos una función que recibe por parámetros dos números:

```
def mas(num, num2):  
    return num + num2  
  
mas(10,20)
```

[4] ✓ 0.0s Python

... 30

Sin embargo, ¿qué pasaría si el usuario, por alguna razón, introduce una cadena de texto en lugar de un número? La consola mostraría lo siguiente:

```
def mas(num, num2):  
    return num + num2  
  
mas("10",20)
```

[5] ✗ 0.0s Python

... -----
Traceback (most recent call last)
Cell In[5], line 4
 1 def mas(num, num2):
 2 return num + num2
----> 4 mas("10",20)

Cell In[5], line 2
 1 def mas(num, num2):
----> 2 return num + num2

TypeError: can only concatenate str (not "int") to str

Como podemos ver, ahora tenemos un `TypeError`. Para solucionar esto, podríamos aplicar un except que capture ese error, haciendo lo siguiente:



```
def mas(num, num2):  
    try:  
        return num + num2  
    except TypeError:  
        print("Error: datos invalidos ingresados!")  
mas("10",20)
```

[6] ✓ 0.0s Python

Después de especificar que se debe imprimir "Error: datos inválidos ingresados" en la consola en caso de un `TypeError`, ahora verificaremos que el código siga funcionando correctamente sin excepciones.

```
def mas(num, num2):  
    try:  
        return num + num2  
    except TypeError:  
        print("Error: datos invalidos ingresados!")  
mas("10",20)
```

[6] ✓ 0.0s Python

... Error: datos invalidos ingresados!

Efectivamente, todo está funcionando.

3- IndexError (Error al acceder a un índice inexistente)

Si has trabajado con listas en Python o cualquier tipo de iterable, probablemente ya hayas visto el error IndexError.

Esto suele ocurrir cuando tenemos cambios en un iterable y podrías estar usando un índice no válido para acceder a un elemento del iterable.

Ejemplo de IndexError:

```
mi_lista = ["Python", "C", "C++", "JavaScript"]  
print(mi_lista[4])
```

[7] ✗ 0.0s Python

... -----
IndexError Traceback (most recent call last)
Cell In[7], line 3
 1 mi_lista = ["Python", "C", "C++", "JavaScript"]
----> 3 print(mi_lista[4])

IndexError: list index out of range



Por lo tanto, una posible solución sería implementar un bloque 'except' que capture específicamente la excepción 'IndexError'. Podríamos hacerlo de la siguiente forma:

```
mi_lista = ["Python","C","C++","JavaScript"]

try:
    print(mi_lista[5])
except IndexError:
    print("Error: Indice no existente!")
```

[8] ✓ 0.0s Python

Al aplicar el except de IndexError, tendríamos el siguiente resultado por consola:

```
mi_lista = ["Python","C","C++","JavaScript"]

try:
    print(mi_lista[5])
except IndexError:
    print("Error: Indice no existente!")
```

[8] ✓ 0.0s Python

... Error: Indice no existente!

Efectivamente, controlamos el error de Índice

4- KeyError (Error al acceder a una clave de un diccionario)

Esto sucede cuando intentamos acceder a una clave que no existe en un diccionario, tendríamos el siguiente resultado en consola:



```
mi_dict={"clave1":"valor1", "clave2":"valor2", "clave3":"valor3"}
buscar_clave = "clave no existente"
print(mi_dict[buscar_clave])
```

[9] ✖ 0.0s Python

... **KeyError** Traceback (most recent call last)
Cell In[9], line 3
1 mi_dict={"clave1":"valor1", "clave2":"valor2", "clave3":"valor3"}
2 buscar_clave = "clave no existente"
----> 3 print(mi_dict[buscar_clave])

KeyError: 'clave no existente'

Podemos resolver el error **KeyError** de la misma forma que el error anterior **IndexError**.

```
mi_dict={"clave1":"valor1", "clave2":"valor2", "clave3":"valor3"}
buscar_clave = "clave no existente"

try:
    print(mi_dict[buscar_clave])
except KeyError:
    print("Error: clave invalida!")
```

[11] ✔ 0.0s Python

... Error: clave invalida!

5- FileNotFoundError (archivo no encontrado o inexistente)

Un error común al trabajar con archivos en Python es el error **FileNotFoundError**.

En el siguiente ejemplo, trataremos de abrir el archivo `mi_archivo.txt` especificando su ruta en la función `open()`, después intentaremos leerlo e imprimir su contenido. Sin embargo, aún no hemos creado el archivo en la ruta especificada. Si intentas correr el código siguiente, obtendrás el error **FileNotFoundError**:



```
mi_archivo = open("Contenido/datos_muestra/mi_archivo.txt")
contenido = mi_archivo.read()
print(contenido)

[12] 0.4s Python

-----
FileNotFoundError                                Traceback (most recent call last)
Cell In[12], line 1
----> 1 mi_archivo = open("Contenido/datos_muestra/mi_archivo.txt")
      2 contenido = mi_archivo.read()
      3 print(contenido)

File ~/Library/Python/3.8/lib/python/site-packages/IPython/core/interactiveshell.py:284
    277 if file in {0, 1, 2}:
    278     raise ValueError(
    279         f"IPython won't let you open fd={file} by default "
    280         "as it is likely to crash IPython. If you know what you are doing, "
    281         "you can use builtins' open."
    282     )
--> 284 return io_open(file, *args, **kwargs)

FileNotFoundError: [Errno 2] No such file or directory: 'Contenido/datos_muestra/mi_ar'
```

Usando try y except, puedes hacer lo siguiente:

1. Tratar de **abrir** el archivo en el bloque try.
2. En el bloque **except** tendremos una **excepción** en caso de que el **archivo no exista** y le **notificaremos al usuario**.
3. Si el bloque try **no tiene errores** y el archivo **si existe**, **leeremos e imprimiremos** el contenido del archivo.
4. En el bloque **finally**, **cerramos el archivo para evitar desperdiciar recursos**. Recuerda que el archivo será cerrado independientemente de lo que ocurra en los pasos de apertura y lectura del archivo

```
try:
    mi_archivo = open("Contenido/datos_muestra/mi_archivo.txt")
except FileNotFoundError:
    print(f"Lo siento, el archivo no existe")
else:
    contenido = mi_archivo.read()
    print(contenido)
finally:
    mi_archivo.close()

[13] 0.0s Python
```




Si el bloque Try funcionó sin errores, entonces con el bloque else, mostramos el contenido del archivo TXT

```
try:
    mi_archivo = open("./mi_archivo.txt")
except FileNotFoundError:
    print(f"Lo siento, el archivo no existe")
else:
    contenido = mi_archivo.read()
    print(contenido)
finally:
    mi_archivo.close()
```

[15] ✓ 0.0s Python

... "Hola, Mundo!"

Contenido de “mi_archivo.txt”

exceptions.ipynb • main.py 3 • main.ipynb • mi_archivo.txt × try_exc...

mi_archivo.txt

```
1 "Hola, Mundo!"
```

El bloque finally cerrará el archivo, por lo que si intentamos acceder a su contenido luego de finalizar, nos dará error:

```
try:
    mi_archivo = open("./mi_archivo.txt")
except FileNotFoundError:
    print(f"Lo siento, el archivo no existe")
else:
    contenido = mi_archivo.read()
    print(contenido)
finally:
    mi_archivo.close()
    contenido = mi_archivo.read()
    print(contenido)
```

[16] ✗ 0.0s Python

... "Hola, Mundo!"

... -----

ValueError Traceback (most recent call last)

Cell In[16], line 10

```
8 finally:
9     mi_archivo.close()
--> 10     contenido = mi_archivo.read()
11     print(contenido)
```

ValueError: I/O operation on closed file.



Conclusión:

Los bloques **try**, **except**, **else** y **finally** son herramientas esenciales en Python para manejar errores de manera eficiente. Permiten que el programa continúe ejecutándose incluso cuando ocurren excepciones, mejorando la robustez y fiabilidad del código.

- **try**: permite envolver el código propenso a errores.
- **except**: maneja las excepciones que puedan ocurrir.
- **else**: ejecuta código si no hay errores en el bloque try.
- **finally**: asegura que ciertas acciones se ejecuten siempre, sin importar si hubo errores.

Usar estos bloques adecuadamente te permitirá escribir programas más predecibles, fáciles de mantener y menos propensos a fallos inesperados.