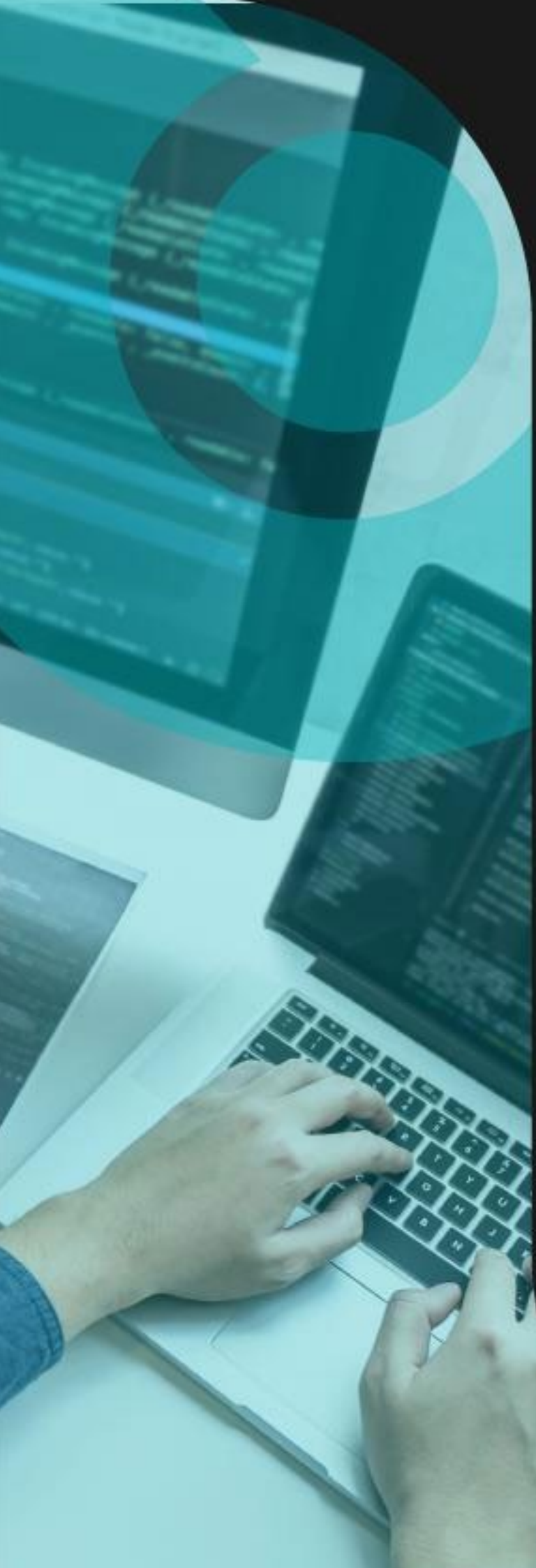

AWAKELAB

BASECAMP

Ciencia de Datos

Módulo: Fundamentos de Programación en Python

Aprendizaje Esperado

- 5. Codificar un algoritmo manejando las excepciones para tomar acciones sobre los errores de acuerdo al lenguaje Python.

Excepciones

Los errores de ejecución son llamados comúnmente excepciones y por eso de ahora en más utilizaremos ese nombre. 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 continúa propagándose hasta llegar a la función inicial del programa y si esta tampoco la maneja se interrumpe la ejecución del programa. Veamos entonces cómo manejar excepciones.

Manejo de excepciones

`try, except y finally`

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.

En el caso de Python, el manejo de excepciones se hace mediante los bloques que utilizan las sentencias `try, except y finally.`

Dentro del bloque `try` se ubica todo el código que pueda llegar a *levantar* una excepción, se utiliza el término levantar para referirse a la acción de generar una excepción.

A continuación se ubica el bloque `except`, que se encarga de capturar la excepción y nos da la oportunidad de procesarla mostrando por ejemplo un mensaje adecuado al usuario. Veamos qué sucede si se quiere realizar una división por cero:

```
print(1/0) # Error al intentar dividir en 0


# ZeroDivisionError: division by zero
```

En este caso, se levantó la excepción `ZeroDivisionError` cuando se quiso hacer la división. Para evitar que se levante la excepción y se detenga la ejecución del programa, se utiliza el bloque `try-except`.

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.

Esto se hace especificando a continuación de la sentencia `except` el nombre de la excepción que se pretende capturar. Un mismo bloque `except` puede atrapar varios tipos de excepciones, lo cual se hace especificando los nombres de las excepciones separados por comas a continuación de la palabra `except`. Es importante destacar que si bien luego de un bloque try puede haber varios bloques `except`, se ejecutará, a lo sumo, uno de ellos.

```
try:
        # aquí colocas el código que puede lanzar excepciones

except IOError:
        # aquí en caso que se haya producido una excepción IOError

except ZeroDivisionError:
        # aquí en caso que se produzca excepción ZeroDivisionError

except:
        # Aquí en caso que se se produzca una excepción que no
corresponda a ninguno de los tipos especificados en los casos
previos
```

Como se muestra en el ejemplo precedente también es posible utilizar una sentencia `except` sin especificar el tipo de excepción a capturar, en cuyo caso se captura cualquier excepción, sin importar su tipo. Cabe destacar, también, que en caso de utilizar una sentencia `except` sin especificar el tipo, la misma debe ser siempre la última de las sentencias `except`, es decir que el siguiente fragmento de código es incorrecto.

```
try:
        # aquí colocas el código que puede lanzar excepciones

except:
        # ERROR de sintaxis, esto no debería ocurrir, pues debe estar
después de except IOError

except IOError:
        # Manejo de la excepción de entrada/salida
```

Finalmente, puede ubicarse un bloque `finally` donde se escriben las sentencias de finalización, que son típicamente acciones de limpieza. 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`.

Veamos ahora cómo es que actúa Python al encontrarse con estos bloques. Python comienza a ejecutar las instrucciones que se encuentran dentro de un bloque `try` normalmente. Si durante la ejecución de esas instrucciones se levanta una excepción, Python interrumpe la ejecución en el punto exacto en que surgió la excepción y pasa a la ejecución del bloque `except` correspondiente.

Para ello, Python verifica uno a uno los bloques `except` y si encuentra alguno cuyo tipo haga referencia al tipo de excepción levantada, comienza a ejecutarlo. Sino encuentra ningún bloque del tipo correspondiente pero hay un bloque `except` sin tipo, lo ejecuta. Al terminar de ejecutar el bloque correspondiente, se pasa a la ejecución del bloque `finally`, si se encuentra definido.

Si, por otra parte, no hay problemas durante la ejecución del bloque `try`, se completa la ejecución del bloque, y luego se pasa directamente a la ejecución del bloque `finally` (si es que está definido).

Bajemos todo esto a un ejemplo concreto, supongamos que nuestro programa tiene que procesar cierta información ingresada por el usuario y guardarla en un archivo. Dado que el acceso a archivos puede levantar excepciones, siempre deberíamos colocar el código de manipulación de archivos dentro de un bloque `try`. Luego deberíamos colocar un bloque `except` que atrape una excepción del tipo `IOError`, que es el tipo de excepciones que lanzan la funciones de manipulación de archivos. Adicionalmente podríamos agregar un bloque `except` sin tipo por si surge alguna otra excepción. Finalmente deberíamos agregar un bloque `finally` para cerrar el archivo, haya surgido o no una excepción.

```
try:
        archivo = open("miarchivo.txt")
        # procesar el archivo

except IOError:
        print("Error de entrada/salida")
        # realizar procesamiento adicional

except:
        # procesar la excepción

finally:
        # si el archivo no está cerrado hay que cerrarlo
        is not (archivo.closed):
                archivo.close()
```

Procesamiento y propagación de excepciones

Hemos visto cómo atrapar excepciones, es necesario ahora que veamos qué se supone que hagamos al atrapar una excepción. En primer lugar podríamos ejecutar alguna lógica particular del caso como: cerrar un archivo, realizar una procesamiento alternativo al del bloque `try`, etc. Pero más allá de esto tenemos algunas opciones genéricas que consisten en: dejar constancia de la ocurrencia de la excepción, propagar la excepción o, incluso, hacer ambas cosas.

Para dejar constancia de la ocurrencia de la excepción, se puede escribir en un archivo de log o simplemente mostrar un mensaje en pantalla. Generalmente cuando se deja constancia de la ocurrencia de una excepción se suele brindar alguna información del contexto en que ocurrió la excepción, por ejemplo: tipo de excepción ocurrida, momento en que ocurrió la excepción y cuáles fueron las llamadas previas a la excepción. El objetivo de esta información es facilitar el diagnóstico en caso de que alguien deba corregir el programa para evitar que la excepción siga apareciendo.

Es posible, por otra parte, que luego de realizar algún procesamiento particular del caso se quiera que la excepción se propague hacia la función

que había invocado a la función actual. Para hacer esto Python nos brinda la instrucción `raise`.

Si se invoca esta instrucción dentro de un bloque `except`, sin pasarle parámetros, Python levantará la excepción atrapada por ese bloque.

También podría ocurrir que en lugar de propagar la excepción tal cual fue atrapada, quisiéramos lanzar una excepción distinta, más significativa para quien invocó a la función actual y que posiblemente contenga cierta información de contexto. Para levantar una excepción de cualquier tipo, utilizamos también la sentencia `raise`, pero indicando el tipo de excepción que deseamos lanzar y pasando a la excepción los parámetros con información adicional que queramos brindar.

El siguiente fragmento de código muestra este uso de `raise`.

```
def dividir(dividendo, divisor):
    try:
        resultado = dividendo / divisor
        return resultado
    except ZeroDivisionError:
        raise ZeroDivisionError("El divisor no puede ser cero")
```

Acceso a información de contexto

Para acceder a la información de contexto estando dentro de un bloque except existen dos alternativas. Se puede utilizar la función `exc_info` del módulo `sys`. Esta función devuelve una tupla con información sobre la última excepción atrapada en un bloque `except`. Dicha tupla contiene tres elementos: el tipo de excepción, el valor de la excepción y las llamadas realizadas.

Otra forma de obtener información sobre la excepción es utilizando la misma sentencia `except`, pasándole un identificador para que almacene una referencia a la excepción atrapada.

```
try:
    # código que puede lanzar una excepción
except Exception, ex:
    # procesamiento de la excepción cuya información
    # es accesible a través del identificador ex
```

Nota

En otros lenguajes, como el lenguaje Java, si una función puede lanzar una excepción en alguna situación, la o las excepciones que lance deben formar parte de la declaración de la función y quien invoque dicha función está obligado a hacerlo dentro de un bloque `try` que la atrape.

En Python, al no tener esta obligación por parte del lenguaje debemos tener cuidado de atrapar las excepciones probables, ya que de no ser así los programas se terminarán inesperadamente.

Referencias

[1] Errores y excepciones
https://docs.python.org/es/3/tutorial/errors.html