Programación Avanzada
IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattan

Anuncios

Jueves 12 de septiembre 2024

1. Mañana se cumple el plazo oficial de entrega de la T2 (20:00 hrs).

2. Domingo se cumple el plazo final de entrega atrasada (20:00 hrs).

3. La próxima semana es **receso** (no hay contenidos para estudiar asociados) .

4. Se encuentra disponible la ETC (Evaluación Temprana de Cursos).

Excepciones

Mensajes de error

Hasta ahora nos hemos encontrado con mensajes de error al realizar ciertas operaciones no permitidas o utilizar métodos de forma incorrecta.

```
while True print("aquí vamos"):
print("no cierro comilla
                         ^
> SyntaxError: invalid syntax
```

→ **Error de sintaxis**

```
10 / 0
> ZeroDivisionError: division by zero.
```

→ **Error durante la ejecución**

Excepciones *Built-in*

- BaseException
  - SyntaxError
  - IndentationError
  - EOFError
  - NameError
  - ZeroDivisionError
  - IndexError
  - KeyError
  - AttributeError
  - TypeError
  - ValueError

Cada una tendrá una forma distinta de **capturar**, tratar y **manejar** la excepción.

... y mas en la documentación

“raise”: levantar una excepción

Dada cierta condición, podríamos diseñar el **levantar** un tipo de excepción particular y añadir un mensaje adicional que informe al usuario sobre el error. Estas excepciones **interrumpen el flujo** del programa.

```
def verificar_largo(lista: list) -> None:
    if len(lista) < 10:
        raise AttributeError("El largo de la lista es menor a 10")
    return None

lista = [1, 2, 3, 4, 5]
verificar_largo(lista)

> AttributeError: El largo de la lista es menor a 10
```

¿Es adecuado?

¿Cómo decido qué tipo de excepción levantar?

“raise”: levantar una excepción

Dada cierta condición, podríamos diseñar el **levantar** un tipo de excepción particular y añadir un mensaje adicional que informe al usuario sobre el error. Estas excepciones **interrumpen el flujo** del programa.

```
def verificar_largo(lista: list) -> None:
    if len(lista) < 10:
        raise AttributeError("El largo de la lista es menor a 10")
    return None

lista = [1, 2, 3, 4, 5]
verificar_largo(lista)

> AttributeError: El largo de la lista es menor a 10
```

¿Qué tipo de excepción es más conveniente a cada caso?

“raise”: levantar una excepción

Dada cierta condición, podríamos diseñar el **levantar** un tipo de excepción particular y añadir un mensaje adicional que informe al usuario sobre el error. Estas excepciones **interrumpen el flujo** del programa.

```
def verificar_largo(lista: list) -> None:
    if len(lista) < 10:
        raise ValueError("El largo de la lista es menor a 10")
    return None

lista = [1, 2, 3, 4, 5]
verificar_largo(lista)
```

```
> ValueError: El largo de la lista es menor a 10
```

Todo dependerá de tu diseño

Veamos una pregunta de Evaluación Escrita

(Midterm 2023-2)

20. Tomando en cuenta el siguiente código, al levantar una excepción con el comando **`raise`**, es correcto afirmar que:

```
print("Comienza")
raise ValueError("Mensaje")
print("Error levantado")
print("Fin del código")
```

A) El error es manejado automáticamente, se imprime el mensaje y el programa continúa su ejecución sin interrupción en la siguiente línea.
B) El programa pausa su ejecución y se reinicia.
C) No se ejecuta la siguiente línea del código automáticamente, sino que se detiene en el punto donde se levantó la excepción y finaliza.
D) El flujo de ejecución del programa se mueve automáticamente a la última línea del programa, ignorando el código restante.
E) El código completo no se ejecuta.

Veamos una pregunta de Evaluación Escrita

(Midterm 2023-2)

20. Tomando en cuenta el siguiente código, al levantar una excepción con el comando **`raise`**, es correcto afirmar que:

```
print("Comienza")
raise ValueError("Mensaje")
print("Error levantado")
print("Fin del código")
```

A) El error es manejado automáticamente, se imprime el mensaje y el programa continúa su ejecución sin interrupción en la siguiente línea.

B) El programa pausa su ejecución y se reinicia.

C) No se ejecuta la siguiente línea del código automáticamente, sino que se detiene en el punto donde se levantó la excepción y finaliza.

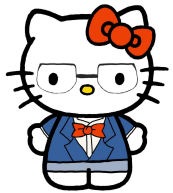
D) El flujo de ejecución del programa se mueve automáticamente a la última línea del programa, ignorando el código restante.

E) El código completo no se ejecuta.

“`try`” y “`except`”: capturar excepciones

Si una excepción fue levantada **durante la ejecución**, podemos **atraparla** y manejarla. Lo que queremos **intentar** se encapsula dentro del bloque **`try`**, mientras que lo que haremos en caso de error va en el bloque **`except`**:

```
try:
    # Intentaremos leer un archivo txt que no existe
    with open("kitty_detective.txt", "r") as txt_file:
        lines = txt_file.readlines()
except FileNotFoundError:
    # Atrapamos FileNotFoundError y levantamos otra excepción
    raise ValueError("Error al leer archivo txt.")
```



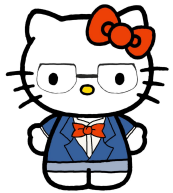
```
> FileNotFoundError: [Errno 2] No such file or directory: 'kitty_detective.txt'
> During handling of the above exception, another exception occurred:
> ValueError: Error al leer archivo txt.
```

“try” y “except”: capturar excepciones

También podemos asignar la instancia del objeto error a una variable, y usar sus atributos o métodos. Dependiendo del manejo del error, **podemos continuar la ejecución del código.**

```
try:
    # Intentaremos leer un archivo txt que no existe
    with open("kitty_detective.txt", "r") as txt_file:
        lines = txt_file.readlines()
except FileNotFoundError as e:
    # Imprimimos un mensaje y el código continúa
    print(f"Error {e.__class__.__name__} al leer archivo txt: {e.filename}")
print("...sigamos")

> Error FileNotFoundError al leer archivo txt: kitty_detective.txt
> ...sigamos
```



“else” y “finally”: bloques adicionales

Junto a los bloques “try”/“except”, se pueden incluir bloques adicionales para complementar el funcionamiento del flujo de captura de errores.

El bloque **“else”** puede añadirse después de **“except”**, y se ejecuta cuando no se cae en la excepción

```
try:
    division = 10/5
except ZeroDivisionError:
    # Imprimimos un mensaje si hay error
    print("Hey, la matemática no permite eso.")
else:
    # Si el error no ocurrió, ejecuto esto
    print(f"Tu resultado es: {division}")

> Tu resultado es: 0.5
```

“else” y “finally”: bloques adicionales

El bloque “**finally**” puede añadirse después de “**except**”, y se ejecuta siempre, sin importar si la excepción ocurrió o no:

```
try:
    division = 10/5
except ZeroDivisionError:
    # Imprimimos un mensaje si hay error
    print("Hey, la matemática no permite eso.")
else:
    # Si el error no ocurrió, ejecuto esto
    print(f"Tu resultado es:", division)
finally:
    # Esto siempre se ejecutará
    print("Gracias por usar mi bello programa")

> Tu resultado es: 0.5
> Gracias por usar mi bello programa
```

Veamos una pregunta de Evaluación Escrita

(Examen 2024-1)

15. Considere el siguiente código en python. Sin considerar saltos de línea, ¿qué imprime este código?

```python
try:
    resultado = 10 / 0
    print("TRY")
except ZeroDivisionError:
    print("EXCEPT")
else:
    print("ELSE")
finally:
    print("FINALLY")
```

A) EXCEPT FINALLY
B) ELSE FINALLY
C) EXCEPT FINALLY TRY
D) TRY EXCEPT FINALLY
E) TRY ELSE FINALLY

Veamos una pregunta de Evaluación Escrita

(Examen 2024-1)

15. Considere el siguiente código en python. Sin considerar saltos de línea, ¿qué imprime este código?

```python
try:
    resultado = 10 / 0
    print("TRY")
except ZeroDivisionError:
    print("EXCEPT")
else:
    print("ELSE")
finally:
    print("FINALLY")
```

A) **EXCEPT FINALLY**
B) ELSE FINALLY
C) EXCEPT FINALLY TRY
D) TRY EXCEPT FINALLY
E) TRY ELSE FINALLY

“`try/except`” vs “`if/else`”

¿Cuándo debo manejar posibles casos bordes de mi programa con “**`try/except`**”?
¿Cuándo debo manejarlos mediante bloques “**`if/else`**”?

En la realidad depende de varios factores, como lo son:

- **Estilo de programación.**
- Si sabemos de antemano **los posibles casos bordes que podrían ocurrir**.
- Si **tenemos acceso de edición de las secciones de código vulnerables.**
- Entre otros.

Estilos de código

`if/else`

LBYL

Look Before You Leap

Mira antes de saltar

`try/except`

EAFP

Easier to Ask Forgiveness than Permission

Mejor pedir perdón que permiso

Estilos de código

`if/else`

LBYL

Look Before You Leap

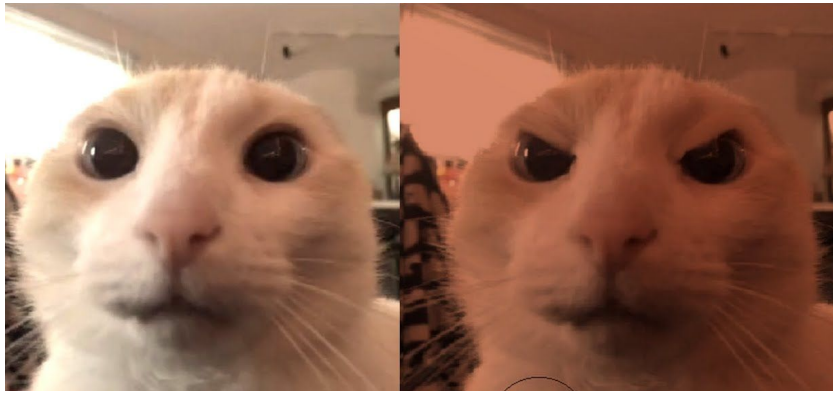
Mira antes de saltar

`try/except`

EAFP

Easier to Ask Forgiveness than Permission

Mejor pedir perdón que permiso



Experiencia 2

Programación Avanzada
IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattan