

Módulo 10 - Manejo de errores.md

Uso de tracebacks para buscar errores

Las excepciones en Python son una característica principal del lenguaje. Es posible que te sorprenda leer que algo que genera errores se resalta como una característica. Esta sorpresa puede deberse a que las herramientas de software sólidas no parecen bloquearse con un traceback seguimiento (varias líneas de texto que indican cómo se inició y finalizó el error).

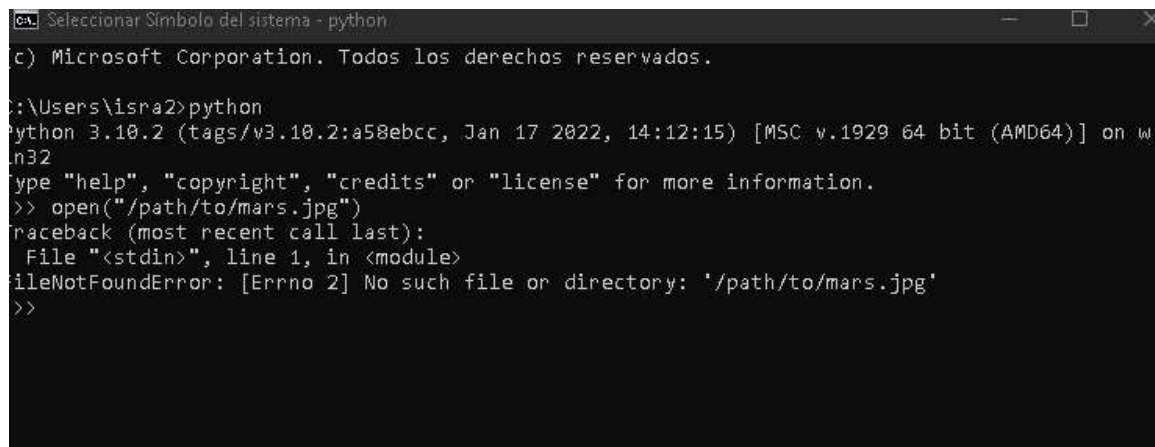
Sin embargo, las excepciones son útiles porque ayudan en la toma de decisiones generando mensajes de error descriptivos. Pueden ayudarte a controlar los problemas esperados e inesperados.

Tracebacks

Un traceback es el cuerpo del texto que puede apuntar al origen (y al final) de un error no controlado. Comprender los componentes de un traceback hará que seas más eficaz al corregir errores o depurar un programa que no funciona bien.

La primera vez que encuentres excepciones en Python podrías verte tentado/a a evitar el error suprimiéndolo. Cuando un programa sufre un error no controlado, aparece un traceback como salida. Como verás en este módulo, los tracebacks son útiles. Hay maneras de controlar correctamente los errores para que no aparezcan o muestren información útil.

Si intentamos en un notebook, abrir un archivo inexistente sucede lo siguiente:



```
Selecciónar Símbolo del sistema - python
(c) Microsoft Corporation. Todos los derechos reservados.

C:\Users\isra2>python
Python 3.10.2 (tags/v3.10.2:a58ebcc, Jan 17 2022, 14:12:15) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>> open("/path/to/mars.jpg")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: '/path/to/mars.jpg'
>>
```

Intenta crear un archivo de Python y asígnale el nombre open.py, con el contenido siguiente:

```
def main():
```

```

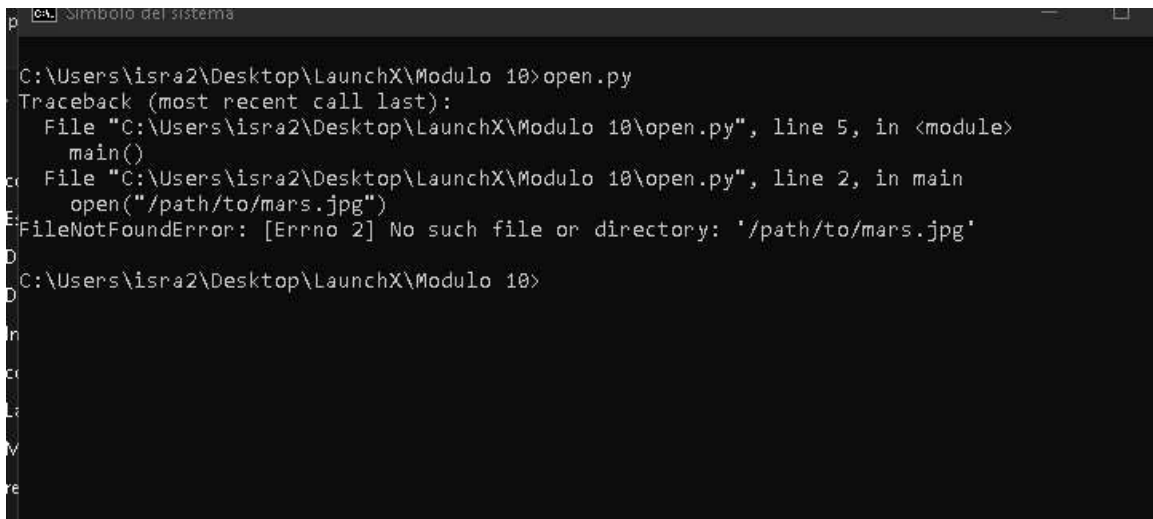
    open("/path/to/mars.jpg")

if __name__ == '__main__':

    main()

```

Se trata de una sola función main() que abre el archivo inexistente, como antes. Al final, esta función usa un asistente de Python que indica al intérprete que ejecute la función main() cuando se le llama en el terminal. Ejecútala con Python y podrás comprobar el siguiente mensaje de error:



```

C:\Users\isra2\Desktop\LaunchX\Modulo 10>open.py
Traceback (most recent call last):
  File "C:\Users\isra2\Desktop\LaunchX\Modulo 10\open.py", line 5, in <module>
    main()
  File "C:\Users\isra2\Desktop\LaunchX\Modulo 10\open.py", line 2, in main
    open("/path/to/mars.jpg")
FileNotFoundError: [Errno 2] No such file or directory: '/path/to/mars.jpg'

C:\Users\isra2\Desktop\LaunchX\Modulo 10>

```

La salida de error tiene más sentido ahora. Las rutas de acceso apuntan a un único archivo denominado open.py. La salida menciona que el error se inicia en la línea 5, que incluye la llamada a main(). A continuación, la salida sigue el error a la línea 2 en la llamada de función open(). Y, por último, FileNotFoundError notifica de nuevo que el archivo o el directorio no existen.

Los tracebacks casi siempre incluyen la información siguiente:

Todas las rutas de acceso de archivo implicadas, para cada llamada a cada función.

Los números de línea asociados a cada ruta de acceso de archivo.

Los nombres de las funciones, métodos o clases implicados en la generación de una excepción.

El nombre de la excepción que se ha producido.

Controlando las excepciones

Cuando encuentres por primera vez excepciones que muestren tracebacks grandes como salida, es posible que te veas tentado/a a detectar todos los errores para evitar que esto suceda.

Si estás en una misión a Marte, ¿Qué podrías hacer si un texto del sistema de navegación dice 'Se ha producido un error'? Imagina que no hay ninguna más información ni contexto, simplemente una luz roja parpadeante con el texto del error. Como desarrollador, resulta útil ponerse al otro lado del programa: ¿Qué puede hacer un usuario cuando se produce un error?

Aunque en este módulo se explica cómo controlar las excepciones detectándolas, no es necesario detectar las excepciones todo el tiempo. A veces resulta útil permitir que se puedan generar excepciones para que otros autores de llamadas puedan tratar los errores.

Try y Except de los bloques

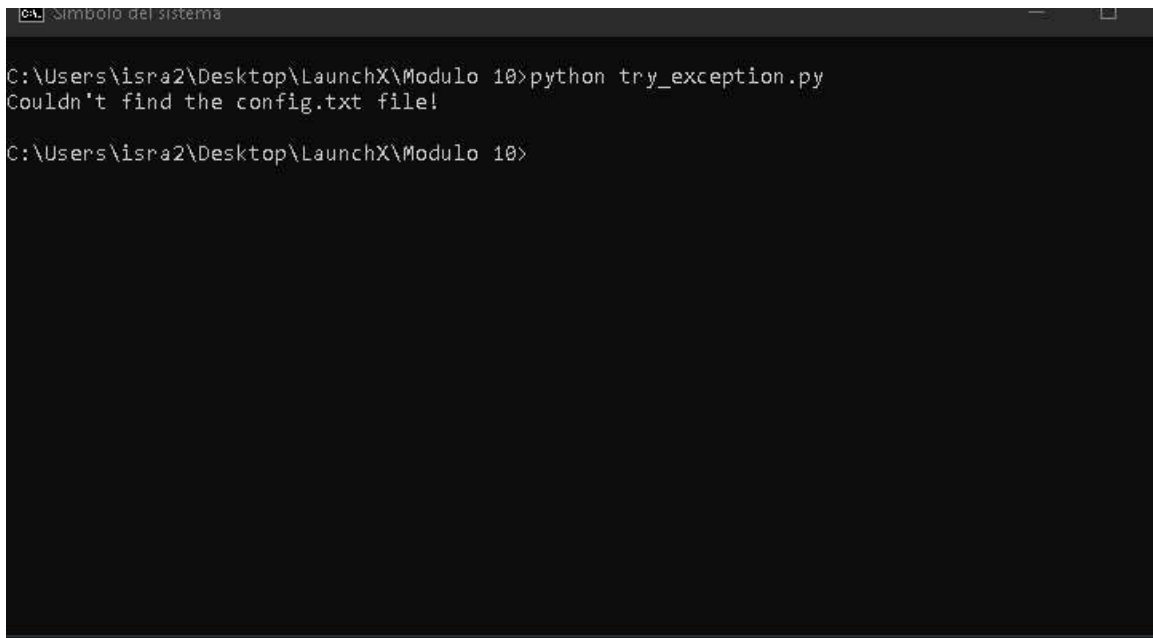
Vamos a usar el ejemplo de navegador a fin de crear código que abra archivos de configuración para la misión de Marte. Los archivos de configuración pueden tener todo tipo de problemas, por lo que es fundamental notificarlos con precisión cuando se presenten. Sabemos que, si no existe un archivo o directorio, se genera `FileNotFoundError`. Si queremos controlar esa excepción, podemos hacerlo con un bloque try y except:

try:

```
... open('config.txt')
```

... except `FileNotFoundError`:

```
... print("Couldn't find the config.txt file!")
```



```
ca Símbolo del sistema
C:\Users\isra2\Desktop\LaunchX\Modulo 10>python try_exception.py
Couldn't find the config.txt file!
C:\Users\isra2\Desktop\LaunchX\Modulo 10>
```

Después de la palabra clave `try`, agregamos código que tenga la posibilidad de producir una excepción. A continuación, agregamos la palabra clave `except` junto con la posible excepción, seguida de cualquier código que deba ejecutarse cuando se produce esa condición. Puesto que `config.txt` no existe en el sistema, Python imprime que el archivo de configuración no está ahí. El bloque `try` y `except`, junto con un mensaje útil, evita un seguimiento y sigue informando al usuario sobre el problema.

Aunque es común un archivo que no existe, no es el único error que podemos encontrar. Los permisos de archivo no válidos pueden impedir la lectura de un archivo, incluso si este existe. Vamos a crear un archivo de Python denominado `config.py`. El archivo tiene código que busca y lee el archivo de configuración del sistema de navegación:



```
Simbolo del sistema
C:\Users\isra2\Desktop\LaunchX\Modulo 10>python config.py
Couldn't find the config.txt file!
C:\Users\isra2\Desktop\LaunchX\Modulo 10>

config: Bloc de notas
Archivo Edición Formato Ver Ayuda
def main():
    try:
        configuration = open('config.txt')
    except FileNotFoundError:
        print("Couldn't find the config.txt file!")

if __name__ == '__main__':
    main()
```

A continuación, quita,ps el archivo config.txt y creamos un directorio denominado config.txt. Intentaremos llamar al archivo config.py para ver un error nuevo que debería ser similar al siguiente:

```
$ python config.py
Traceback (most recent call last):
  File "/tmp/config.py", line 9, in <module>
    main()
  File "/tmp/config.py", line 3, in main
    configuration = open('config.txt')
IsADirectoryError: [Errno 21] Is a directory: 'config.txt'
```

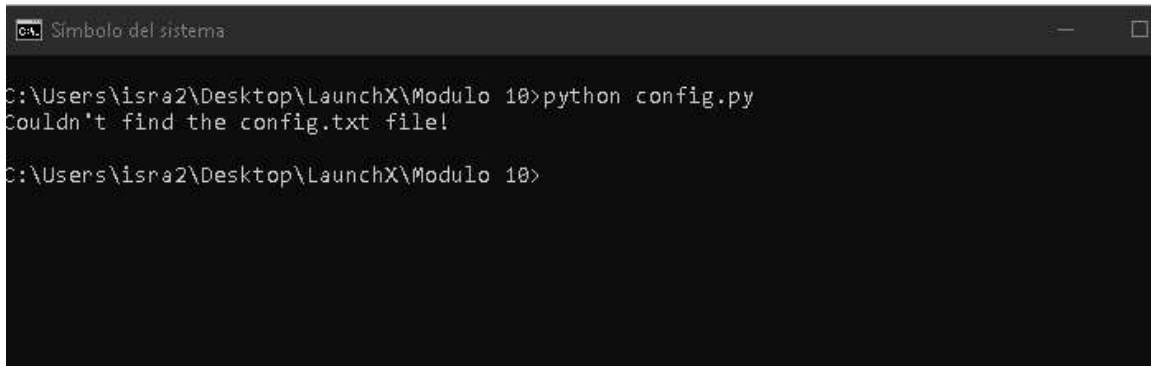
El problema ahora es que el mensaje de error es incorrecto. El archivo existe, pero tiene permisos diferentes y Python no puede leerlo. Cuando se trata con errores de software, puede resultar frustrante tener errores que hagan lo siguiente:

No indiquen cuál es el problema real.
Proporcionen una salida que no coincida con el problema real.
No sugieran lo que se puede hacer para corregir el problema.
Vamos a corregir este fragmento de código para abordar todas estas frustraciones.
Revertiremos la detección de FileNotFoundError y luego agregamos otro bloque except para detectar PermissionError:

Archivo Edición Formato Ver Ayuda

```
def main():
    try:
        configuration = open('config.txt')
    except FileNotFoundError:
        print("Couldn't find the config.txt file!")
    except IsADirectoryError:
        print("Found config.txt but it is a directory, couldn't read it")

if __name__ == '__main__':
    main()
```

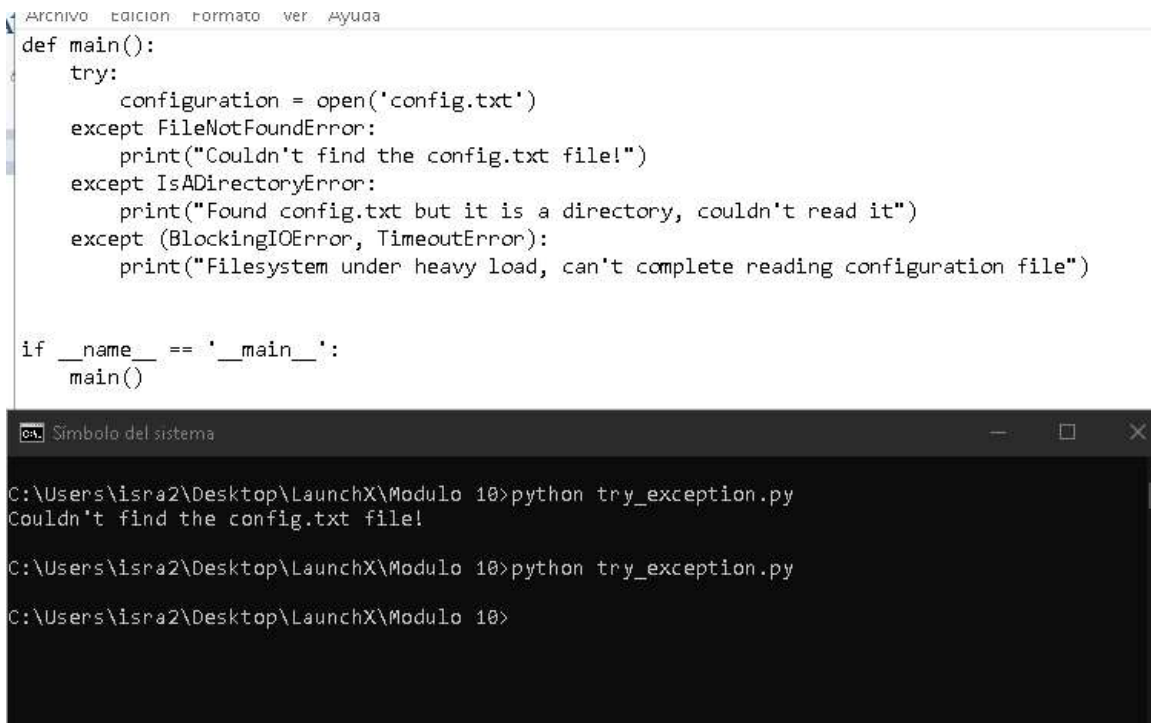


```
C:\Users\isra2\Desktop\LaunchX\Modulo 10>python config.py
Couldn't find the config.txt file!

C:\Users\isra2\Desktop\LaunchX\Modulo 10>
```

Cuando los errores son de una naturaleza similar y no es necesario controlarlos individualmente, puedes agrupar las excepciones como una usando paréntesis en la línea `except`. Por ejemplo, si el sistema de navegación está bajo cargas pesadas y el sistema de archivos está demasiado ocupado, tiene sentido detectar `BlockingIOError` y `TimeoutError` juntos:

```
def main():
    try:
        configuration = open('config.txt')
    except FileNotFoundError:
        print("Couldn't find the config.txt file!")
    except IsADirectoryError:
        print("Found config.txt but it is a directory, couldn't read it")
    except (BlockingIOError, TimeoutError):
        print("Filesystem under heavy load, can't complete reading configuration file")
```



The image shows a Python script in a text editor and its execution in a command prompt. The script, named `try_exception.py`, defines a `main()` function that attempts to open a file named `config.txt`. It uses a `try` block to handle potential exceptions: `FileNotFoundError` (prints "Couldn't find the config.txt file!"), `IsADirectoryError` (prints "Found config.txt but it is a directory, couldn't read it"), and a general exception for `(BlockingIOError, TimeoutError)` (prints "Filesystem under heavy load, can't complete reading configuration file"). The script is executed in a Windows command prompt, showing the first two error messages.

```
Archivo Edición Formato Ver Ayuda
def main():
    try:
        configuration = open('config.txt')
    except FileNotFoundError:
        print("Couldn't find the config.txt file!")
    except IsADirectoryError:
        print("Found config.txt but it is a directory, couldn't read it")
    except (BlockingIOError, TimeoutError):
        print("Filesystem under heavy load, can't complete reading configuration file")

if __name__ == '__main__':
    main()
```

```
Simbolo del sistema
C:\Users\isra2\Desktop\LaunchX\Modulo 10>python try_exception.py
Couldn't find the config.txt file!

C:\Users\isra2\Desktop\LaunchX\Modulo 10>python try_exception.py
C:\Users\isra2\Desktop\LaunchX\Modulo 10>
```

Generación de excepciones

Ahora que tienes una buena comprensión de los `tracebacks` y el control de excepciones, vamos a revisar la generación de excepciones.

Es posible que ya conozcas una situación que podría provocar una condición de error al escribir código. En estas situaciones, resulta útil generar excepciones que permitan que otro código comprenda cuál es el problema.

La generación de excepciones también puede ayudar en la toma de decisiones para otro código. Como hemos visto antes, en función del error, el código puede tomar decisiones inteligentes para resolver, solucionar o ignorar un problema.

Los astronautas limitan su uso de agua a unos 11 litros al día. Vamos a crear una función que, con base al número de astronautas, pueda calcular la cantidad de agua quedará después de un día o más:

```
def water_left(astronauts, water_left, days_left):
    daily_usage = astronauts * 11
    total_usage = daily_usage * days_left
    total_water_left = water_left - total_usage
    return f"Total water left after {days_left} days is: {total_water_left} liters"
```

```
C:\Users\isra2\Desktop\LaunchX\Modulo 10>python agua.py
Total water left after 2 days is: -10 liters
```

```
agua: Bloc de notas
Archivo  Edición  Formato  Ver  Ayuda
def water_left(astronauts, water_left, days_left):
    daily_usage = astronauts * 11
    total_usage = daily_usage * days_left
    total_water_left = water_left - total_usage
    return f"Total water left after {days_left} days is: {total_water_left} liters"

print(water_left(5, 100, 2))
```

Esto no es muy útil, ya que una carencia en los litros sería un error. Después, el sistema de navegación podría alertar a los astronautas que no habrá suficiente agua para todos en dos días. Si eres un ingeniero(a) que programa el sistema de navegación, podrías generar una excepción en la función `water_left()` para alertar de la condición de error:

```
def water_left(astronauts, water_left, days_left):
    daily_usage = astronauts * 11
    total_usage = daily_usage * days_left
    total_water_left = water_left - total_usage
    if total_water_left < 0:
        raise RuntimeError(f"There is not enough water for {astronauts} astronauts after
{days_left} days!")
    return f"Total water left after {days_left} days is: {total_water_left} liters"
```

```
C:\Users\isra2\Desktop\LaunchX\Modulo 10>python agua.py
Traceback (most recent call last):
  File "C:\Users\isra2\Desktop\LaunchX\Modulo 10\agua.py", line 9, in <module>
    print(water_left(5, 100, 2))
  File "C:\Users\isra2\Desktop\LaunchX\Modulo 10\agua.py", line 6, in water_left
    raise RuntimeError(f"There is not enough water for {astronauts} astronauts after {days_l
eft} days!")
RuntimeError: There is not enough water for 5 astronauts after 2 days!

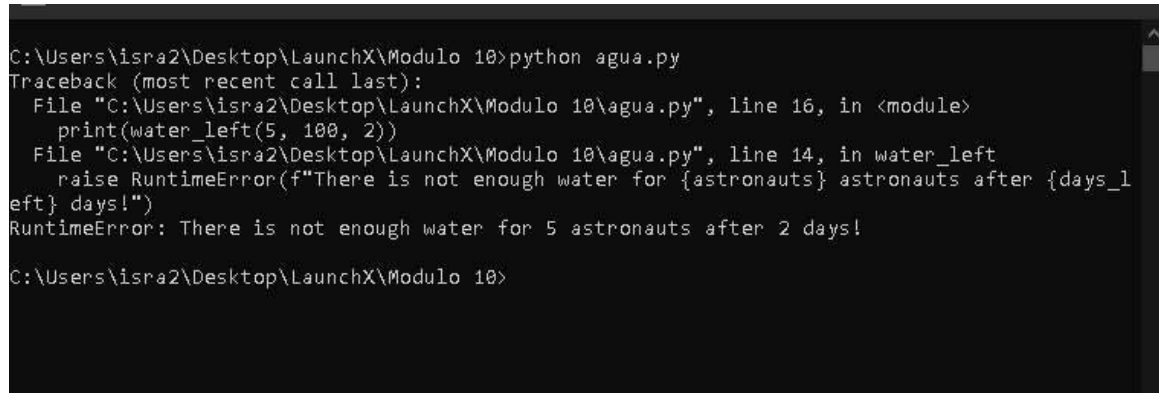
C:\Users\isra2\Desktop\LaunchX\Modulo 10>
```

El error de `TypeError` no es muy descriptivo en el contexto de lo que espera la función. Actualizaremos la función para que use `TypeError`, pero con un mensaje mejor:


```

def water_left(astronauts, water_left, days_left):
    for argument in [astronauts, water_left, days_left]:
        try:
            # If argument is an int, the following operation will work
            argument / 10
        except TypeError:
            # TypeError will be raised only if it isn't the right type
            # Raise the same exception but with a better error message
            raise TypeError(f"All arguments must be of type int, but received: '{argument}'")
    daily_usage = astronauts * 11
    total_usage = daily_usage * days_left
    total_water_left = water_left - total_usage
    if total_water_left < 0:
        raise RuntimeError(f"There is not enough water for {astronauts} astronauts after
{days_left} days!")
    return f"Total water left after {days_left} days is: {total_water_left} liters"

```



```

C:\Users\isra2\Desktop\LaunchX\Modulo 10>python agua.py
Traceback (most recent call last):
  File "C:\Users\isra2\Desktop\LaunchX\Modulo 10\agua.py", line 16, in <module>
    print(water_left(5, 100, 2))
  File "C:\Users\isra2\Desktop\LaunchX\Modulo 10\agua.py", line 14, in water_left
    raise RuntimeError(f"There is not enough water for {astronauts} astronauts after {days_
left} days!")
RuntimeError: There is not enough water for 5 astronauts after 2 days!

C:\Users\isra2\Desktop\LaunchX\Modulo 10>

```

Resumen

Para ser un desarrollador eficaz, debes saber cómo funcionan las excepciones y cómo controlarlas. En este módulo, has descubierto cómo usar la salida de excepción para la depuración, cómo detectar y generar excepciones y, por último, cómo afecta a la lógica de un programa cuando se producen excepciones.

Con los conocimientos del uso y el control de excepciones, puedes evitar problemas comunes, como detectarlos todos para evitar tracebacks.

En este módulo, has obtenido información sobre estas aptitudes:

- Leer y usar la salida de error de las excepciones
- Controlar correctamente las excepciones
- Generar excepciones con mensajes de error útiles
- Usar excepciones para controlar el flujo de un programa
- Kata del módulo 10
- Intenta realizar los ejercicios descritos en este documento.

Curso Propedúctico de Python para Launch X - Innovación Virtual.

Material desarrollado con base en los contenidos de MSLearn y la metáfora de LaunchX, traducción e implementación por: Fernanda Ochoa - Learning Producer de LaunchX.

Redes:

GitHub: FernandaOchoa

Twitter: @imonsh

Instagram: fherz8a