Modulo 4: Excepciones

Excepciones

El lidiar con errores de programación tiene (al menos) dos partes. La primera es cuando te metes en problemas porque tu código, aparentemente correcto, se alimenta con datos incorrectos. Por ejemplo, esperas que se ingrese al código un valor entero, pero tu usuario descuidado ingresa algunas letras al azar.

Puede suceder que tu código termine en ese momento y el usuario se quede solo con un mensaje de error conciso y a la vez ambiguo en la pantalla. El usuario estará insatisfecho y tu también deberías estarlo. Te mostraremos cómo proteger tu código de este tipo de fallas y cómo no provocar la ira del usuario.

La segunda parte de lidiar con errores de programación se revela cuando ocurre un comportamiento no deseado del programa debido a errores que se cometieron cuando se estaba escribiendo el código. Este tipo de error se denomina comúnmente «bug» (bicho en inglés), que es una manifestación de una creencia bien establecida de que, si un programa funciona mal, esto debe ser causado por bichos maliciosos que viven dentro del hardware de la computadora y causan cortocircuitos u otras interferencias.

Esta idea no es tan descabellada como puede parecer: incidentes de este tipo eran comunes en tiempos en que las computadoras ocupaban grandes pasillos, consumían kilovatios de electricidad y producían enormes cantidades de calor. Afortunadamente, o no, estos tiempos se han ido para siempre y los únicos errores que pueden estropear tu código son los que tú mismo sembraste en el código. Por lo tanto, intentaremos mostrarte cómo encontrar y eliminar tus errores, en otras palabras, cómo depurar tu código.

Cuando los datos no son lo que deberían ser

Escribamos un fragmento de código extremadamente trivial: leerá un número natural (un entero no negativo) e imprimirá su recíproco. De esta forma, 2 se convertirá en 0.5 (1/2) y 4 en 0.25 (1/4).

```
value = int(input('Ingresa un número natural: '))
print('El recíproco de', value, 'es', 1/value)
```

¿Hay algo que pueda salir mal? El código es tan breve y compacto que no parece que vayamos a encontrar ningún problema allí.

Parece que ya sabes hacia dónde vamos. Sí, tienes razón: ingresar datos que no sean un número entero (que también incluye ingresar nada) arruinará completamente la ejecución del programa. Esto es lo que verá el usuario del código:

```
Traceback (most recent call last):
  File "code.py", line 1, in
    value = int(input('Ingresa un número natural: '))
ValueError: invalid literal for int() with base 10: ''
```

Todas las líneas que muestra Python son significativas e importantes, pero la última línea parece ser la más valiosa. La primera palabra de la línea es el nombre de la excepción la cual provoca que tu código se detenga. Su nombre aquí es ValueError. El resto de la línea es solo una breve explicación que especifica con mayor precisión la causa de la excepción ocurrida.

¿Cómo lo afrontas? ¿Cómo proteges tu código de la terminación abrupta, al usuario de la decepción y a ti mismo de la insatisfacción del usuario?

La primera idea que se te puede ocurrir es verificar si los datos proporcionados por el usuario son válidos y negarte a cooperar si los datos son incorrectos. En este caso, la verificación puede basarse en el hecho de que esperamos que la cadena de entrada contenga solo dígitos.

Ya deberías poder implementar esta verificación y escribirla tu mismo, ¿no es así? También es posible comprobar si la variable value es de tipo int (Python tiene un medio especial para este tipo de comprobaciones: es un operador llamado is. La revisión en sí puede verse de la siguiente manera:

```
type(value) is int
```

Su resultado es verdadero si el valor actual de la variable value es del tipo int.

Perdónanos si no dedicamos más tiempo a esto ahora; encontrarás explicaciones más detalladas sobre el operador `is` en un módulo del curso dedicado a la programación orientada a objetos.

Es posible que te sorprendas al saber que no queremos que realices ninguna validación preliminar de datos. ¿Por qué? Porque esta no es la forma que Python recomienda.

El Código Python

En el mundo de Python, hay una regla que dice: «Es mejor pedir perdón que pedir permiso».

Detengámonos aquí por un momento. No nos malinterpretes, no queremos que apliques la regla en tu vida diaria. No tomes el automóvil de nadie sin permiso, con la esperanza de que puedas ser tan convincente que evites la condena por lo ocurrido. La regla se trata de otra cosa.

En realidad, la regla dice: «es mejor manejar un error cuando ocurre que tratar de evitarlo».

«De acuerdo», puedes decir, «pero ¿cómo debo pedir perdón cuando el programa finaliza y no queda nada que más por hacer?». Aquí es donde algo llamado excepción entra en escena.

```
try:
    # Es un lugar donde
    # tu puedes hacer algo
    # sin pedir permiso.
except:
    # Es un espacio dedicado
    # exclusivamente para pedir perdón.
```

Puedes ver dos bloques aquí:

- El primero, comienza con la palabra clave reservada `try:` este es el lugar donde se coloca el código quesospecha que es riesgoso y puede terminar en caso de un error; nota: este tipo de error lleva por nombre excepción, mientras que la ocurrencia de la excepción se le denomina generar; podemos decir que se genera (o se generó) una excepción.

69

- El segundo, la parte del código que comienza con la palabra clave reservada `except:` esta parte fue diseñada para manejar la excepción; depende de ti lo que quieras hacer aquí: puedes limpiar el desorden o simplemente puede barrer el problema debajo de la alfombra (aunque se prefiere la primera solución).

70 71 Entonces, podríamos decir que estos dos bloques funcionan así: 72 73

- La palabra clave reservada `try` marca el lugar donde intentas hacer algo sin permiso.

74

- La palabra clave reservada `except` comienza un lugar donde puedes mostrar tu talento para disculparte o se sospecha que es riesgoso y puede terminar en caso de un error; nota: este tipo de error lleva por nombre excepción, mientras que la ocurrencia de la excepción se le denomina generar; podemos decir que se genera (o se generó) una excepción.
- El segundo, la parte del código que comienza con la palabra clave reservada `except:` esta parte fue diseñada para manejar la excepción; depende de ti lo que quieras hacer aquí: puedes limpiar el desorden o simplemente puede barrer el problema debajo de la alfombra (aunque se prefiere la primera solución).

Entonces, podríamos decir que estos dos bloques funcionan así:

- La palabra clave reservada `try` marca el lugar donde intentas hacer algo sin permiso.
- La palabra clave reservada `except` comienza un lugar donde puedes mostrar tu talento para disculparte o pedir perdón.

Como puedes ver, este enfoque acepta errores (los trata como una parte normal de la vida del programa) en lugar de intensificar los esfuerzos para evitarlos por completo.

La excepción confirma la regla

Reescribamos el código para adoptar el enfoque de Python para la vida.

```
try:
    value = input('Ingresa un número natural: ')
    print('El recíproco de', value, 'es', 1/int(value))
except:
    print('No se que hacer con', value)
```

Resumamos lo que hemos hablado:

- Cualquier fragmento de código colocado entre `try` y `except` se ejecuta de una manera muy especial: cualquier error que ocurra aquí dentro no terminará la ejecución del programa. En cambio, el control saltará inmediatamente a la primera línea situada después de la palabra clave reservada `except`, y no se ejecutará ninguna otra línea del bloque `try`.
- El código en el bloque `except` se activa solo cuando se ha encontrado una excepción dentro del bloque `try`. No hay forma de llegar por ningún otro medio.
- Cuando el bloque `try` o `except` se ejecutan con éxito, el control vuelve al proceso normal de ejecución y cualquier código ubicado más allá en el archivo fuente se ejecuta como si no hubiera pasado nada.

Ahora queremos hacerte una pregunta: ¿Es `ValueError` la única forma en que el control podría caer dentro del bloque `except`?

Cómo lidiar con más de una excepción

La respuesta obvia es «no»: hay más de una forma posible de plantear una excepción. Por ejemplo, un usuario puede ingresar cero como entrada, ¿puedes predecir lo que sucederá a continuación?

Sí, tienes razón: la división colocada dentro de la invocación de la función **print()** generará la excepción **ZeroDivisionError**. Como es de esperarse, el comportamiento del código será el mismo que en el caso anterior: el usuario verá el mensaje «No se que hacer con...», lo que parece bastante razonable en este contexto, pero también es posible que desees manejar este tipo de problema de una manera un poco diferente.

¿Es posible? Por supuesto que lo es. Hay al menos dos enfoques que puedes implementar aquí.

El primero de ellos es simple y complicado al mismo tiempo: puedes agregar dos bloques `try` por separado, uno que incluya la invocación de la función **input()** donde se puede generar la excepción **ValueError**, y el segundo dedicado a manejar posibles problemas inducidos por la división. Ambos bloques `try` tendrían su propio `except`, y de esa manera, tendrías un control total sobre dos errores diferentes.

Esta solución es buena, pero es un poco larga: el código se hincha innecesariamente. Además, no es el único peligro que te espera. Toma en cuenta que dejar el primer bloque `try-except` deja mucha incertidumbre; tendrás que agregar código adicional para asegurarte de que el valor que ingresó el usuario sea seguro para usar en la división. Así es como una solución aparentemente simple se vuelve demasiado complicada.

Dos excepciones después de un try.

```
try:
    value = input('Ingresa un número natural: ')
    print('El recíproco de', value, 'es', 1/int(value))
except ValueError:
    print('No se que hacer con', value)
except ZeroDivisionError:
    print('La división entre cero no está permitida en nuestro Universo.')
```

Como puedes ver, acabamos de agregar un segundo `except`. Esta no es la única diferencia; toma en cuenta que ambos `except` tienen **nombres** de excepción específicos. En esta variante, cada una de las excepciones esperadas tiene su propia forma de manejar el error, pero se debe enfatizarse en que **solo una** de todas puede interceptar el control; **si se ejecuta una, todas las demás permanecen inactivas**. Además, la cantidad de excepciones no está limitado: puedes especificar tantas o tan pocas como necesites, pero no se te olvide que **ninguna** de las excepciones se puede especificar más de una vez.

Pero esta todavía no es la última palabra de Python sobre excepciones.

La excepción por defecto y cómo usarla

El código ha cambiado de nuevo, ¿puedes ver la diferencia?

```
try:
    value = input('Ingresa un número natural: ')
    print('El recíproco de', value, 'es', 1/int(value))
except ValueError:
    print('No se que hacer con', value)
except ZeroDivisionError:
    print('La división entre cero no está permitida en nuestro Universo.')
except:
    print('Ha sucedido algo extraño, ¡lo siento!')
```

Hemos agregado un tercer `except`, pero esta vez **no tiene un nombre de excepción** específico; podemos decir que es **anónimo** o (lo que está más cerca de su función real) es el **por defecto**. Puedes esperar que cuando se genere una excepción y no haya un except dedicado a esa excepción, esta será manejada por la excepción por defecto.

Nota: ¡el `except` por defecto debe ser el último `except`! ¡Siempre!

Algunas excepciones útiles

Analicemos con más detalle algunas excepciones útiles (o más bien, las más comunes) que puedes llegar a experimentar.

ZeroDivisionError

Esta aparece cuando intentas forzar a Python a realizar cualquier operación que provoque una división en la que el divisor es cero o no se puede distinguir de cero. Toma en cuenta que hay más de un operador de Python que puede hacer que se genere esta excepción. ¿Puedes adivinarlos todos?

Si, estos son: /, //, y %.

ValueError

Espera esta excepción cuando estás manejando valores que pueden usarse de manera inapropiada en algún contexto. En general, esta excepción se genera cuando una función (como **int()** o **float()**) recibe un argumento de un tipo adecuado, pero su valor es inaceptable.

TypeError

Esta excepción aparece cuando intentas aplicar un dato cuyo tipo no se puede aceptar en el contexto actual. Mira el ejemplo:

```
short_list = [1]
one_value = short_list[0.5]
```

No está permitido usar un valor flotante como índice de una lista (la misma regla también se aplica a las tuplas). TypeError es un nombre adecuado para describir el problema y una excepción adecuada a generar.

AttributeError

Esta excepción llega, entre otras ocasiones, cuando intentas activar un método que no existe en un elemento con el que se está tratando. Por ejemplo:

```
short_list = [1]
short_list.append(2)
short_list.depend(3)
```

La tercera línea de nuestro ejemplo intenta hacer uso de un método que no está incluido en las listas. Este es el lugar donde se genera la excepción AttributeError.

SyntaxError

Esta excepción se genera cuando el control llega a una línea de código que viola la gramática de Python. Puede sonar extraño, pero algunos errores de este tipo no se pueden identificar sin ejecutar primero el código. Este tipo de comportamiento es típico de los lenguajes interpretados: el intérprete siempre trabaja con prisa y no tiene tiempo para escanear todo el código fuente. Se conforma con comprobar el código que se está ejecutando

en el momento. Muy pronto se te presentará un ejemplo de esta categoría.

Es una mala idea manejar este tipo de excepciones en tus programas. Deberías producir código sin errores de sintaxis, en lugar de enmascarar las fallas que has causado.

Por qué no se puede evitar el probar tu código

Aunque vamos a resumir nuestras consideraciones excepcionales aquí, no creas que es todo lo que Python puede ofrecer para ayudarte a suplicar perdón. La maquinaria de excepciones de Python es mucho más compleja y sus capacidades te permiten desarrollar estrategias de manejo de errores expandidas. Volveremos a estos temas, lo prometemos. No dudes en realizar tus experimentos y sumergirte en las excepciones por ti mismo.

Ahora queremos contarte sobre el segundo lado de la lucha interminable contra los errores: el destino inevitable de la vida de un desarrollador. Como no puedes evitar la creación de errores en tu código, siempre debes estar listo para buscarlos y destruirlos. No entierres la cabeza en la arena: ignorar los errores no los hará desaparecer.

Un deber importante para los desarrolladores es probar el código recién creado, pero no debes olvidar que las pruebas no son una forma de demostrar que el código está libre de errores. Paradójicamente, lo único que las pruebas determinan, es que tu código contiene errores. No creas que puedes relajarte después de una prueba exitosa.

El segundo aspecto importante de las pruebas de software es estrictamente psicológico. Es una verdad conocida desde hace años que los autores, incluso aquellos que son confiables y conscientes de sí mismos, **no pueden evaluar y verificar objetivamente sus trabajos.**

Es por eso por lo que cada novelista necesita un editor y cada programador necesita un «tester». Algunos dicen, con un poco de rencor, pero con sinceridad, que los desarrolladores prueban su código para mostrar su perfección, no para encontrar problemas que puedan frustrarlos. Los «testers» o probadores están libres de tales dilemas, y es por eso por lo que su trabajo es más efectivo y rentable.

Por supuesto, esto no te exime de estar atento y cauteloso. Prueba tu código lo mejor que puedas. No facilites demasiado el trabajo a los probadores.

Su deber principal es asegurarse de haber verificado todas las rutas o caminos de ejecución por las que puede pasar tu código. ¿Suena misterioso? ¡Por supuesto que no!

Rastreando las rutas de ejecución

Supón que acabas de terminar de escribir este fragmento de código.

```
temperature = float(input('Ingresa la temperatura actual:'))

if temperature > 0:
    print("Por encima de cero")
elif temperature < 0:
    print("Por debajo de cero")
else:
    print("Cero")
```

Existen tres rutas o caminos de ejecución independientes en el código, ¿puedes verlas? Están determinadas por

las sentencias **if-elif-else**. Por supuesto, las rutas de ejecución pueden construirse mediante muchas otras sentencias como bucles, o incluso bloques **try-except**.

Si vas a probar tu código de manera justa y quieres dormir profundamente y soñar sin pesadillas (las pesadillas sobre errores pueden ser devastadoras para el rendimiento de un desarrollador), estás obligado a preparar un conjunto de datos de prueba que obligará a tu código a negociar todos los posibles caminos.

En nuestro ejemplo, el conjunto debe contener al menos tres valores flotantes: uno positivo, uno negativo y cero.

Cuando Python cierra sus ojos

Tal prueba es crucial. Queremos mostrarte por qué no debes omitirlo. Observa el siguiente código.

```
temperature = float(input('Ingresa la temperatura actual:'))

if temperature > 0:
    print("Por encima de cero")
elif temperature < 0:
    prin("Por debajo de cero")
else:
    print("Cero")
```

Introdujimos intencionalmente un error en el código; esperamos que tus ojos atentos lo noten de inmediato. Sí, eliminamos solo una letra y, en efecto, la invocación válida de la función **print()** se convierte en la obviamente inválida invocación **«prin()«**. No existe tal función como **«prin()«** en el alcance de nuestro programa, pero ¿es realmente obvio para Python?

Ejecuta el código e ingresa un **0**.

Como puedes ver, el código finaliza su ejecución sin ningún obstáculo.

¿Cómo es eso posible? ¿Por qué Python **pasa por alto** un error de desarrollador tan evidente?

¿Puedes encontrar las respuestas a estas preguntas fundamentales?

Pruebas y probadores

La respuesta es más simple de lo esperado y también un poco decepcionante. Python, como seguramente sabes, es un lenguaje **interpretado**. Esto significa que el código fuente se analiza y ejecuta al mismo tiempo. En consecuencia, es posible que Python no tenga tiempo para analizar las líneas de código que no están sujetas a ejecución. Como dice un antiguo dicho de los desarrolladores: «es una característica, no un error» (no utilices esta frase para justificar el comportamiento extraño de tu código).

¿Entiendes ahora por qué el pasar por todos los caminos de ejecución es tan vital e inevitable?

Supongamos que terminas tu código y que las pruebas que has realizado son exitosas. Entregas tu código a los probadores y, ¡afortunadamente! - encontraron algunos errores en él. Estamos usando la palabra «afortunadamente» de manera completamente consciente. Debes aceptar que, en primer lugar, los probadores son los mejores amigos del desarrollador; no debes tratar a los errores que ellos encuentran como una ofensa o una malignidad; y, en segundo lugar, cada error que encuentran los probadores es un error que no afectará a los usuarios. Ambos factores son valiosos y merecen tu atención.

Ya sabes que tu código contiene un error o errores (lo segundo es más probable). Ahora, ¿cómo los localizas y

cómo arreglas tu código?

Error frente a depuración (Bug vs. debug)

La medida básica que un desarrollador puede utilizar contra los errores es, como era de esperarse, un **depurador**, mientras que el proceso durante el cual se eliminan los errores del código se llama **depuración**. Según un viejo chiste, la depuración es un complicado juego de misterio en el que eres simultáneamente el asesino, el detective y, la parte más dolorosa de la intriga, la víctima. ¿Estás listo para interpretar todos estos roles? Entonces debes armarte con un depurador.

Un depurador es un software especializado que puede controlar cómo se ejecuta tu programa. Con el depurador, puedes ejecutar tu código línea por línea, inspeccionar todos los estados de las variables y cambiar sus valores en cualquier momento sin modificar el código fuente, detener la ejecución del programa cuando se cumplen o no ciertas condiciones, y hacer muchas otras tareas útiles.

Podemos decir que todo IDE está equipado con un depurador más o menos avanzado. Incluso IDLE tiene uno, aunque puedes encontrar su manejo un poco complicado y problemático. Si deseas utilizar el depurador integrado de IDLE, debes activarlo mediante la opción «Debug» en la barra de menú de la ventana principal de IDLE. Es el punto de partida para la depuración.

Las capturas de pantalla que ves al lado muestran el depurador IDLE durante una simple sesión de depuración.

Puedes ver cómo el depurador visualiza las variables y los valores de los parámetros. Observa la pila de llamadas que muestra la cadena de invocaciones que van desde la función actualmente ejecutada hacia el intérprete.

Si deseas saber más sobre el depurador IDLE, consulta la documentación IDLE.

Depuración por impresión

Esta forma de depuración, que se puede aplicar a tu código mediante cualquier tipo de depurador, a veces se denomina **depuración interactiva**. El significado del término se explica por sí mismo: el proceso necesita su interacción (la del desarrollador) para que se lleve a cabo.

Algunas otras técnicas de depuración se pueden utilizar para cazar errores. Es posible que no puedas o no quieras usar un depurador (las razones pueden variar). ¿Estás entonces indefenso? ¡Absolutamente no!

Puedes utilizar una de las tácticas de depuración más simples y antiguas (pero aún útil) conocida como la **depuración por impresión**. El nombre habla por sí mismo: simplemente insertas varias invocaciones **print()** adicionales dentro de tu código para generar datos que ilustran la ruta que tu código está negociando actualmente. Puedes imprimir los valores de las variables que pueden afectar la ejecución.

Estas impresiones pueden generar texto significativo como «Estoy aquí», «Ingresé a la función foo()», «El resultado es 0», o pueden contener secuencias de caracteres que solo tu puedes leer. Por favor, no uses palabras obscenas o indecentes para ese propósito, aunque puedas sentir una fuerte tentación; tu reputación puede arruinarse en un momento si estas payasadas se filtran al público.

Como puedes ver, este tipo de depuración no es realmente interactiva en lo absoluto, o es interactiva solo en pequeña medida, cuando decides aplicar la función input() para detener o retrasar la ejecución del código.

Una vez que se encuentran y eliminan los errores, las impresiones adicionales pueden comentarse o eliminarse; tu decides. No permitas que se ejecuten en el código final; pueden confundir tanto a los probadores como a los usuarios, y traer mal karma sobre ti.

Algunos consejos útiles

Aquí hay algunos consejos que pueden ayudarte a encontrar y eliminar errores. Ninguno de ellos es definitivo. Úsalos de manera flexible y confía en tu intuición. No te creas a ti mismo, comprueba todo dos veces.

1. **Intenta decirle a alguien** (por ejemplo, a tu amigo o compañero de trabajo) qué es lo que se espera que haga tu código y cómo se espera que se comporte. Se concreto y no omitas detalles. Responde todas las preguntas que te hagan. Es probable que te des cuenta de la causa del problema mientras cuentas tu historia, ya que el hablar activa esas partes de tu cerebro que permanecen inactivas mientras codificas. Si ningún humano puede ayudarte con el problema, usa un patito amarillo de goma en su lugar. No estamos bromeando, consulta el artículo de Wikipedia para obtener más información sobre esta técnica de uso común: Método de depuración del patito de goma.
2. **Intenta aislar el problema**. Puedes extraer la parte de tu código que se sospecha que es responsable de tus problemas y ejecutarla por separado. Puedes comentar partes del código para ocultar el problema. Asigna valores concretos a las variables en lugar de leerlos desde la consola. Prueba tus funciones aplicando valores de argumentos predecibles. Analiza el código cuidadosamente. Léelo en voz alta.
3. Si el error apareció recientemente y no había aparecido antes, **analiza todos los cambios que has introducido en tu código**; uno de ellos puede ser la razón.
4. **Tómate un descanso**, bebe una taza de café, toma a tu perro y sal a caminar, lee un buen libro, incluso dos, haz una llamada telefónica a tu mejor amigo; te sorprenderás de la frecuencia con la que esto ayuda.
5. **Se optimista**: eventualmente encontrarás el error; te lo prometemos.

Prueba unitaria: un nivel más alto de codificación

También existe una técnica de programación importante y ampliamente utilizada que tendrás que adoptar tarde o temprano durante tu carrera de desarrollador: se llama **prueba unitaria**. El nombre puede ser un poco confuso, ya que no se trata solo de probar el software, sino también (y, sobre todo) **de cómo se escribe el código**.

Para resumir la historia, las pruebas unitarias asumen que las pruebas son partes inseparables del código y la preparación de los datos de prueba es una parte inseparable de la codificación. Esto significa que cuando escribes una función o un conjunto de funciones cooperativas, también estás obligado a crear un conjunto de datos para los cuales el comportamiento de tu código es predecible y conocido.

Además, debes equipar a tu código con una interfaz que pueda ser utilizada por un entorno de pruebas automatizado. En este enfoque, cualquier enmienda realizada al código (incluso la menos significativa) debe ir seguida de la ejecución de todas las pruebas unitarias que acompañan al código fuente.

Para estandarizar este enfoque y facilitar su aplicación, Python proporciona un módulo dedicado llamado **unittest**. No vamos a discutirlo aquí, es un tema amplio y complejo. Por lo tanto, hemos preparado un curso y una ruta de certificación independiente para este tema. Se llama «Python para pruebas», y ahora te invitamos a participar en él.

Puntos Clave: Excepciones

1. En Python, existe una distinción entre dos tipos de errores:

- **Errores de sintaxis** (errores de análisis), que ocurren cuando el analizador encuentra una sentencia de código que no es correcta. Por ejemplo:

El intentar ejecutar la siguiente línea:

```
print("Hola, ¡Mundo!)
```

Provocará un error del tipo *SyntaxError*, y da como resultado el siguiente (o similar) mensaje que se muestra en la consola:

```
  File "main.py", line 1

    print("Hola, ¡Mundo!)
                         ^
SyntaxError: EOL while scanning string literal
```

Presta atención a la flecha: indica el lugar donde el analizador de Python ha tenido problemas. En este caso, la doble comilla es la que falta. ¿Lo notaste?

Excepciones, ocurren incluso cuando una sentencia o expresión es sintácticamente correcta. Estos son los errores que se detectan durante la ejecución, cuando tu código da como resultado un error que no es incondicionalmente fatal. Por ejemplo: El intentar ejecutar la siguiente línea:

```
print(1/0)
```

Provocará una excepción *ZeroDivisionError*, y da como resultado el siguiente (o similar) mensaje que se muestra en la consola:

```
Traceback (most recent call last):
  File "main.py", line 1, in
    print(1/0)
ZeroDivisionError: division by zero
```

Presta atención a la última línea del mensaje de error; en realidad, te dice lo que sucedió. Existen muchos diferentes tipos de excepciones, como *ZeroDivisionError*, *NameError*, *TypeError*, y muchas mas; y esta parte del mensaje te informa qué tipo de excepción se ha generado. Las líneas anteriores muestran el contexto en el que ha ocurrido la excepción.

2. Puedes «capturar» y manejar excepciones en Python usando el bloque try-except. Por lo tanto, si tienes la sospecha de que cualquier fragmento de código en particular puede generar una excepción, puedes escribir el código que la manejará con elegancia y no interrumpirá el programa. Observa el ejemplo:

```
while True:
    try:
        number = int(input("Ingresa un número entero: "))
        print(number/2)
        break
    except:
        print("Advertencia: el valor ingresado no es un número válido. Intenta de
nuevo...")
```

El código anterior le pide al usuario que ingrese un valor hasta que el valor ingresado sea un número entero válido. Si el usuario ingresa un valor que no se puede convertir a un int, el programa imprimirá en la consola Advertencia: el valor ingresado no es un número válido. Intenta de nuevo..., y pide al usuario que ingrese un número nuevamente. Veamos que sucede en dicho caso.

1. El programa entra en el bucle while.
2. El bloque try se ejecuta y el usuario ingresa un valor incorrecto, por ejemplo: ¡hola!.
3. Se genera una excepción y el resto del código del bloque try se omite. El programa salta al bloque

except, lo ejecuta, y luego se sigue al código que se encuentra después del bloque try-except. Si el usuario ingresa un valor correcto y no se genera ninguna excepción, las instrucciones subsiguientes al bloque try, son ejecutadas. En este caso, los excepts no se ejecutan.

3. Puedes manejar múltiples excepciones en tu bloque de código. Analiza los siguientes ejemplos:

```
while True:
    try:
        number = int(input("Ingresa un número entero: "))
        print(5/number)
        break
    except ValueError:
        print("Valor incorrecto.")
    except ZeroDivisionError:
        print("Lo siento. No puedo dividir entre cero.")
    except:
        print("No se que hacer...")
```

Puedes utilizar varios bloques except dentro de una sentencia try, y especificar nombres de excepciones. Si se ejecuta alguno de los except, los otros se omitirán. Recuerda: puedes especificar una excepción integrada solo una vez. Además, no olvides que la excepción por **defecto** (o genérica), es decir, a la que no se le especifica nombre, debe ser colocada **al final** (utiliza las excepciones más específicas primero, y las más generales al último).

También puedes especificar y manejar múltiples excepciones integradas dentro de un solo bloque except:

```
while True:
    try:
        number = int(input("Ingresa un número entero: "))
        print(5/number)
        break
    except (ValueError, ZeroDivisionError):
        print("Valor incorrecto o se ha roto la regla de división entre cero.")
    except:
        print("Lo siento, algo salió mal...")
```

4. Algunas de las excepciones integradas más útiles de Python son: *ZeroDivisionError*, *ValueError*, *TypeError*, *AttributeError*, y *SyntaxError*. Una excepción más que, en nuestra opinión, merece tu atención es la excepción *KeyboardInterrupt*, que se genera cuando el usuario presiona la tecla de interrupción (CTRL-C o Eliminar). Ejecuta el código anterior y presiona la combinación de teclas para ver qué sucede.

Para obtener más información sobre las excepciones integradas de Python, consulta la documentación oficial de Python aquí.

5. Por último, pero no menos importante, debes recordar cómo probar y depurar tu código. Utiliza técnicas de depuración como depuración de impresión; si es posible, pide a alguien que lea tu código y te ayude a encontrar errores o mejorarlo; intenta aislar el fragmento de código que es problemático y susceptible a errores, **prueba tus funciones** aplicando valores de argumento predecibles, y trata de **manejar** las situaciones en las que alguien ingresa valores incorrectos; **comenta** las partes del código que ocultan el problema. Finalmente, toma descansos y vuelve a tu código después de un tiempo con un par de ojos nuevos.

ejercicio

Tu tarea es escribir **un simple programa que simule jugar a tic-tac-toe (nombre en inglés) con el usuario**. Para hacerlo más fácil, hemos decidido simplificar el juego. Aquí están nuestras reglas:

- La maquina (por ejemplo, el programa) jugará utilizando las 'X's.
- El usuario (por ejemplo, tu) jugarás utilizando las 'O's.
- El primer movimiento es de la maquina: siempre coloca una 'X' en el centro del tablero.
- Todos los cuadros están numerados comenzando con el 1 (observa el ejemplo para que tengas una referencia).
- El usuario ingresa su movimiento introduciendo el número de cuadro elegido. El número debe de ser valido, por ejemplo un valor entero mayor que 0 y menor que 10, y no puede ser un cuadro que ya esté ocupado.
- El programa verifica si el juego ha terminado. Existen cuatro posibles veredictos: el juego continua, el juego termina en empate, tu ganas, o la maquina gana.
- La maquina responde con su movimiento y se verifica el estado del juego.
- No se debe implementar algún tipo de inteligencia artificial, la maquina elegirá un cuadro de manera aleatoria, eso es suficiente para este juego.

El ejemplo del programa es el siguiente:

```
+-------+-------+-------+
|       |       |       |
|   1   |   2   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------+-------+-------+
Ingresa tu movimiento: 1
+-------+-------+-------+
|       |       |       |
|   0   |   2   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
|       |       |       |
+-------+-------+-------+
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   8   |   9   |
```

```
|       |       |       |
+-------+-------+-------+
Ingresa tu movimiento: 8
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   6   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   0   |   9   |
|       |       |       |
+-------+-------+-------+
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   4   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   0   |   9   |
|       |       |       |
+-------+-------+-------+
Ingresa tu movimiento: 4
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   3   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   0   |   9   |
|       |       |       |
+-------+-------+-------+
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   0   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   7   |   0   |   9   |
|       |       |       |
+-------+-------+-------+
```

```
Ingresa tu movimiento: 7
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   X   |   X   |
|       |       |       |
+-------+-------+-------+
|       |       |       |
|   O   |   O   |   9   |
|       |       |       |
+-------+-------+-------+
¡Has Ganado!
```

Requerimientos

Implementa las siguientes características:

- El tablero debe ser almacenado como una lista de tres elementos, mientras que cada elemento es otra lista de tres elementos (la lista interna representa las filas) de manera que todos los cuadros puedas ser accedidos empleado la siguiente sintaxis:

```
board[row][column]
```

- Cada uno de los elementos internos de la lista puede contener 'O', 'X', o un digito representando el número del cuadro (dicho cuadro se considera como libre).
- La apariencia de tablero debe de ser igual a la presentada en el ejemplo.
- Implementa las funciones definidas para ti en el editor.

Para obtener un valor numérico aleatorio se puede emplear una función integrada de Python denominada **randrange()**. El siguiente ejemplo muestra como utilizarla (El programa imprime 10 números aleatorios del 1 al 8).

Nota: La instrucción **from-import** provee acceso a la función randrange definida en un módulo externo de Python denominado `random`.

```
from random import randrange

for i in range(10):
    print(randrange(8))
```

```
def DisplayBoard(board):
    # La función acepta un parámetro el cual contiene el estado actual del tablero
    # y lo muestra en la consola.

def EnterMove(board):
    # La función acepta el estado actual del tablero y pregunta al usuario acerca
de su movimiento,
    # verifica la entrada y actualiza el tablero acorde a la decisión del usuario.

def MakeListOfFreeFields(board):
    # La función examina el tablero y construye una lista de todos los cuadros
```

```
vacíos.
    # La lista esta compuesta por tuplas, cada tupla es un par de números que
indican la fila y columna.

def VictoryFor(board, sign):
    # La función analiza el estatus del tablero para verificar si
    # el jugador que utiliza las 'O's o las 'X's ha ganado el juego.

def DrawMove(board):
    # La función dibuja el movimiento de la máquina y actualiza el tablero.
```