Talento Digital para Chile:

Módulo 3
Programación Avanzada en Python

nodovirtual.awakelab.cl

JELOU futurejob by Adalid Chile

3.5.- Contenido 5: Codificar un programa Python utilizando generación y captura de errores y excepciones para el control del flujo durante excepciones

Objetivo de la jornada

1. Reconoce los mecanismos y formas del lenguaje Python para generación de errores y su aplicación
2. Codifica un programa Python que permite controlar las excepciones para resolver un problema
3. Codificar un programa que lanza excepciones personalizadas para resolver un problema

3.5.1.- Manejo de Errores y Excepciones

En esta clase aprenderá a manejar los errores para que sus programas no se bloqueen cuando se encuentren con situaciones inesperadas. Aprenderá sobre las excepciones, que son objetos especiales que Python crea para administrar los errores que surgen mientras se ejecuta un programa.

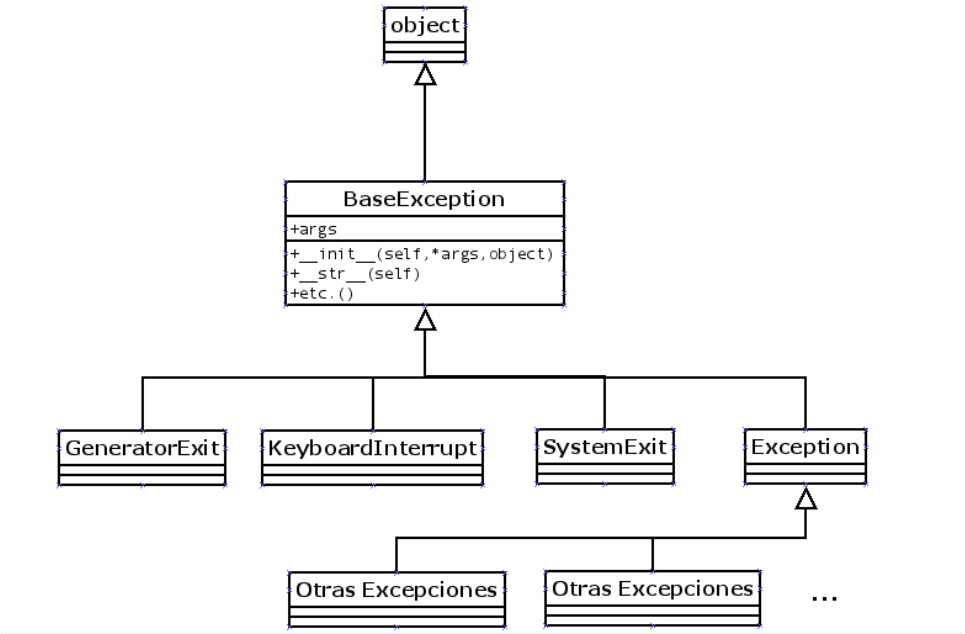
3.5.1.1. Qué es una excepción

Python usa objetos especiales llamados excepciones para manejar errores que surgen durante la ejecución de un programa. Siempre que ocurre un error que hace que Python no esté seguro de qué hacer a continuación, crea un objeto de excepción. Si escribe código que maneja la excepción, el programa continuará ejecutándose. Si no maneja la excepción, el programa se detendrá y mostrará un rastreo, que incluye un informe de la excepción que se generó. Las excepciones se manejan con bloques "try-except". Un bloque "try-except" le pide a Python que haga algo, pero también le dice a Python qué hacer si se genera una excepción. Cuando usa bloques "try-except", sus programas continuarán ejecutándose incluso si las

cosas comienzan a salir mal. En lugar de rastreos, que pueden resultar confusos para los usuarios de leer, los usuarios verán mensajes de error amigables que usted escriba.
Los programas son muy frágiles. Sería bueno si el código siempre devolviera un resultado válido, pero a veces no se puede calcular un resultado válido. No es posible dividir por cero o acceder al octavo elemento en una lista de cinco elementos, por ejemplo.
En los viejos tiempos, la única forma de evitar esto era verificar rigurosamente las entradas de cada función para asegurarse de que tuvieran sentido. Normalmente, las funciones tenían valores de retorno especiales para indicar una condición de error; por ejemplo, podrían devolver un número negativo para indicar que no se pudo calcular un valor positivo. Diferentes números pueden significar que ocurrieron diferentes errores. Cualquier código que llamara a esta función tendría que verificar explícitamente una condición de error y actuar en consecuencia. Una gran cantidad de código no se molestó en hacer esto y los programas simplemente fallaron.
¡No es así en el mundo orientado a objetos! En este apartado estudiaremos las excepciones, objetos de error especiales que solo necesitan ser manejados cuando tiene sentido manejarlos.

3.5.1.2. Tipos de excepciones

Python nos entrega una amplia gama de excepciones provenientes de la clase **BaseException** Si examinamos el archivo donde se define la clase **BaseException (__builtin__.pyi)** podemos ver algunas dependencias



Naturalmente no queremos examinar todo el codigo de Python en sí mismo para descubrir a que tenemos acceso, una lista de las excepciones disponibles se puede encontrar en https://docs.python.org/3/library/exceptions.html

Entonces, ¿qué es una excepción, realmente? Técnicamente, una excepción es solo un objeto. Hay muchas clases de excepción diferentes disponibles y podemos definir fácilmente más de las nuestras. Lo único que todos tienen en común es que derivan de una clase incorporada llamada BaseException.

Estos objetos de excepción se vuelven especiales cuando se manejan dentro del flujo de control del programa. Cuando ocurre una excepción, todo lo que se suponía que debía suceder no sucede, a menos que se suponga que suceda cuando ocurrió una excepción. ¿Tiene sentido? ¡No se preocupe, lo hará!

3.5.1.3. Errores de sintaxis

Entonces, ¿cómo hacemos que ocurra una excepción? ¡La forma más fácil es hacer algo sin sentido! Es probable que ya haya hecho esto y haya visto el resultado de la excepción. Por ejemplo, cada vez que Python encuentra una línea en su programa que no puede entender, se rescata con **SyntaxError**, que es un tipo de excepción.

Aquí hay uno común:

```
print "hola amigos"
```

Ejecutar eso resulta en:

```
File "<ipython-input-91-8c507a3b6601>", line 1
  print "hola amigos"                    ^

SyntaxError: Missing parentheses in call to 'print'. Did you mean
print("hola amigos")?
```

Si se fija con detención la parte clave es **SyntaxError: Missing parentheses....**
Esa declaración **print** era un comando válido en Python 2 y versiones anteriores, pero en Python 3, debido a que **print** es una función, tenemos que encerrar los argumentos entre paréntesis. Entonces, si escribimos lo anterior en un intérprete de Python 3, obtenemos la excepción **SyntaxError**.

Un **SyntaxError**, aunque es común, es en realidad una excepción especial, porque no podemos manejarlo. Nos dice que escribimos algo mal y es mejor que averigüemos qué es. Algunas otras excepciones comunes, que podemos manejar, se muestran en los siguientes ejemplos:

```
x = 5 / 0
```

Resulta en:

```
ZeroDivisionError Traceback (most recent call last)
<ipython-input-92-663c7a933a87> in <module>
----> 1 x = 5 / 0

ZeroDivisionError: division by zero

lst = [1,2,3]
print(lst[3])
```

Resulta en:

```
IndexError Traceback (most recent call last)
<ipython-input-93-9bad12320794> in <module>
      1 lst = [1,2,3]
----> 2 print(lst[3])

IndexError: list index out of range

lst + 2
```

Resulta en:

```
TypeError Traceback (most recent call last)
<ipython-input-94-ffe4e6e220bf> in <module>
----> 1 lst + 2

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

Lst.add
```

Resulta en:

```
AttributeError Traceback (most recent call last)
<ipython-input-95-50e4efd52cad> in <module>
----> 1 lst.add

AttributeError: 'list' object has no attribute 'add'

d = {'a': 'hola'}
d['b']
```

Resulta en:

```
KeyError Traceback (most recent call last)
<ipython-input-98-78b0e3f0353a> in <module>      1 d = {'a': 'hola'}
----> 2 d['b']

KeyError: 'b'
```

```
print(esta_no_es_una_variable)
```

Resulta en:

```
NameError  Traceback (most recent call last)
<ipython-input-99-0e53a6a47162> in <module>
----> 1 print(esta_no_es_una_variable)

NameError: name 'esta_no_es_una_variable' is not defined
```

A veces, estas excepciones son indicadores de algo mal en nuestro programa (en cuyo caso iríamos al número de línea indicado y lo arreglaríamos), pero también ocurren en situaciones legítimas. Un **ZeroDivisionError** no siempre significa que recibimos una entrada no válida, solo una entrada diferente. El usuario puede haber ingresado un cero por error, o a propósito, o puede representar un valor legítimo, como una cuenta bancaria vacía o la edad de un niño recién nacido.

Es posible que haya notado que todas las excepciones integradas anteriores terminan en el nombre **Error**. En Python, las palabras "error" y "exception" se usan casi indistintamente. Los errores a veces se consideran más graves que las excepciones, pero se tratan exactamente de la misma manera. De hecho, todas las clases de error anteriores tienen **Exception** (que extiende **BaseException**) como su superclase (ver diagrama UML arriba).

3.5.1.4. Manejo de excepciones

Ahora veamos el otro lado de la moneda llamada excepción. Es decir, si nos encontramos con una situación de excepción, ¿cómo debería reaccionar nuestro código o recuperarse de ella?

La sentencia try/cach

Manejamos las excepciones envolviendo cualquier código que pueda arrojar una (ya sea un código de excepción en sí mismo o una llamada a cualquier función o método que pueda tener una excepción en su interior) dentro de una cláusula **try ... except**. La sintaxis más básica se ve así:

```
def no_return():
    print("Voy a levantar un excepcion")
    raise Exception("Esta siempre se levanta")
    print("Esta linea nunca se ejecuta")
    return "Yo no voy a retornar"

try:
```

```
    no_return()
except:
    print("encontre una exception")
print("ejecutado despues de la exception")
```

Resulta en:

```
Voy a levantar un excepcion
encontre una exception
ejecutado despues de la exception
```

Por el momento no se preocupe de la función **no_return()** es solo para demostrar la generación de un error o exception, ya veremos que hace más adelante.
La función **no_return** nos informa felizmente que está a punto de generar una excepción. Pero la engañamos y capturamos la excepción. Una vez atrapada, podemos limpiar mensajes nosotros mismos (en este caso, dando a conocer que estábamos manejando la situación), y continuar en nuestro camino, sin interferencia de la función ofensiva. El resto del código en la función **no_return** aún no se ejecutó, pero el código que llamó a la función pudo recuperarse y continuar.

Tenga en cuenta los espacios alrededor de **try** y **except**. La cláusula **try** envuelve cualquier código que pueda generar una excepción. La cláusula **except** vuelve a estar en el mismo nivel de espacios que la línea **try**. Cualquier código para manejar la excepción se pone con espacios después de la cláusula **except**. Luego, el código normal se reanuda en el nivel de espacios original.

El problema con el código anterior es que detectará cualquier tipo de excepción. ¿Qué pasaría si estuviéramos escribiendo un código que pudiera generar tanto **TypeError** como **ZeroDivisionError**? Es posible que queramos capturar **ZeroDivisionError**, pero dejemos que **TypeError** se propague a la consola. ¿Puede adivinar la sintaxis? Aquí hay una función bastante simple que hace precisamente eso:

```
def division(numero):
    try:
        return 100 / numero
    except ZeroDivisionError:
        return "No se puede dividir por cero"

print(division(0))
print(division(50))
print(division("ola de mar"))
```

Resulta en:

```
No se puede dividir por cero2.0
---------------------------------------------------------------------
------
TypeError  Traceback (most recent call last)
<ipython-input-109-4d974d3be925> in <module>
```

```
    7 print(division(0))
    8 print(division(50.0))
----> 9 print(division("ola de mar"))

<ipython-input-109-4d974d3be925> in division(numero)
      1 def division(numero):
      2     try:
----> 3         return 100 / numero
      4     except ZeroDivisionError:
      5         return "No se puede dividir por cero"

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

La primera línea de salida muestra que, si ingresamos 0, nos protegemos adecuadamente. Si nosotros llamamos con un número válido (tenga en cuenta que no es un número entero, pero sigue siendo un divisor válido), funciona correctamente. Sin embargo, si ingresamos una cadena (se preguntaba cómo obtener un **TypeError**, ¿no es así?), Falla con una excepción. Si hubiéramos usado una cláusula **except** vacía, que no especificara un **ZeroDivisionError**, nos habría acusado de dividir por cero cuando le enviamos una cadena, lo que no es un comportamiento adecuado en absoluto.

Captura de múltiples excepciones

Incluso podemos detectar dos o más excepciones diferentes y manejarlas con el mismo código. A continuación, se muestra un ejemplo que plantea tres tipos diferentes de excepciones. Maneja **TypeError** y **ZeroDivisionError** con el mismo controlador de excepciones, pero también puede generar un **ValueError** si proporciona el número 13:

```
def division2(numero):
    try:
        if numero == 13:
        raise ValueError("13 es un numero de mala suerte")
        return 100 / numero
   except (ZeroDivisionError, TypeError):
        return "Ingrese otro numero que no sea cero"

for val in (0, "hola", 50.0, 13):

    print("Probando {}:".format(val), end=" ")
    print(division2(val))
```

El loop for en la parte inferior simplemente recorre varias entradas de prueba e imprime los resultados. Si se está preguntando sobre ese argumento "end=" ") en la declaración de impresión, simplemente convierte la nueva línea final predeterminada en un espacio para que se una con la salida de la siguiente línea. Aquí hay una ejecución del script:

```
Probando 0: Ingrese otro numero que no sea cero
Probando hola: Ingrese otro numero que no sea cero
Probando 50.0: 2.0
Probando 13:
---------------------------------------------------------------------
------
ValueError Traceback (most recent call last)
<ipython-input-112-28d687c0461a> in <module>
     11
     12     print("Probando {}:".format(val), end=" ")
---> 13     print(division2(val))

<ipython-input-112-28d687c0461a> in division2(numero)
      2     try:
      3     if numero == 13:
----> 4        raise ValueError("13 es un numero de mala suerte")
      5     return 100 / numero
      6     except (ZeroDivisionError, TypeError):

ValueError: 13 es un numero de mala suerte
```

El número 0 y la cadena son capturados por la cláusula **except** y se imprime un mensaje de error adecuado. La excepción del número 13 no se captura, porque es un **ValueError**, que no se incluyó en los tipos de excepciones que se manejan.

Todo esto está muy bien, pero ¿qué pasa si queremos detectar diferentes excepciones y hacer cosas diferentes con ellas? ¿O tal vez queremos hacer algo con una excepción y luego permitir que continúe burbujeando hasta la función principal, como si nunca se hubiera detectado? No necesitamos ninguna sintaxis nueva para tratar estos casos. Es posible apilar cláusulas **except**, y solo se ejecutará una. Para la segunda pregunta, la palabra clave **raise**, sin argumentos, volverá a generar la última excepción si ya estamos dentro de un controlador de excepciones, observe:

```
def division3(numero):
   try:
      if numero == 13:
         raise ValueError("13 es un numero de mala suerte")
      return 100 / numero
   except ZeroDivisionError:
      return "Ingrese otro numero que no sea cero"
   except TypeError:
      return "Ingrese un valor numerico"
   except ValueError:
       print("No, No, no 13!")
       raise
```

La última línea vuelve a generar **ValueError**, por lo que después de generar **No, No, no 13!**, se generará la excepción nuevamente; obtendremos el error informado en la pila de la consola.

Si apilamos cláusulas de excepción como hicimos anteriormente, solo la primera cláusula coincidente se va a ejecutar, incluso si más de una de ellas encaja. ¿Cómo puede coincidir más de una cláusula? Recuerde que las excepciones son objetos y la herencia puede tener lugar. La mayoría de las excepciones amplían la clase **Exception** (que es, en sí misma, derivada de **BaseException**). Si detectamos **Exception** antes de detectar **TypeError**, solo se ejecutará el controlador de **Exception**, porque **TypeError** es una **Exception** por herencia.

Esto puede resultar útil en los casos en los que queremos manejar algunas excepciones específicamente y luego manejar todas las excepciones restantes en un caso más general. Simplemente podemos detectar **Exception** después de detectar cualquier excepción específica y manejar el caso general allí.

A veces, cuando detectamos una excepción, necesitamos una referencia al objeto **Exception** en sí. Esto sucede con mayor frecuencia si definimos nuestras propias excepciones con argumentos personalizados, pero también puede ser relevante con excepciones estándar. La mayoría de las clases de excepción aceptan un conjunto de argumentos en su constructor y es posible que deseemos acceder a esos atributos en el controlador de excepciones. Si definimos nuestra propia clase de excepción, incluso podemos llamar a métodos personalizados cuando la detectemos. La sintaxis para capturar una excepción como variable utiliza la palabra clave **as**:

```
try:
    raise ValueError("Este es un argumento")
except ValueError as e:
    print("Los argumentos de la excepción fueron", e.args)
```

Si ejecutamos este simple fragmento, imprime el argumento de cadena que pasamos a **ValueError** al inicializar.

3.5.1.5. Throw de excepciones

Ahora, entonces, ¿qué hacemos si estamos escribiendo un programa que necesita informar al usuario o una función que llama que las entradas son de alguna manera inválidas? Sería bueno si pudiéramos usar el mismo mecanismo que usa Python... ¡y podemos! ¿Quieres ver cómo? Aquí hay una clase simple que agrega elementos a una lista solo si son números enteros pares:

```
class SoloPares(list):
    def append(self, entero):
        if not isinstance(entero, int):
            raise TypeError("Solo enteros pueden ser agregados")
        if entero % 2:
            raise ValueError("Solo numeros pares pueden ser \
                              agregados")
        super().append(entero)
```

Esta clase extiende **list**, como discutimos con anterioridad, y sobreescribe el método **append** para comprobar dos condiciones que garantizan que el elemento sea un número entero par. Primero verificamos si la entrada es una instancia del tipo **int**, y luego usamos el operador de módulo para asegurarnos de que sea divisible por dos. Si no se cumple alguna de las dos condiciones, la palabra clave **raise** se utiliza para provocar una excepción. La palabra clave **raise** es seguida simplemente por el objeto que se genera como excepción. En el ejemplo anterior, se han construido dos objetos a partir de las clases integradas **TypeError** y **ValueError**. El objeto que emitió **raise** podría ser fácilmente una instancia de una nueva clase de excepción que creamos nosotros mismos (veremos cómo en breve), una excepción que se definió en otro lugar, o incluso un objeto de excepción que se haya generado y manejado previamente. Si probamos esta clase en el intérprete de Python, podemos ver que está generando información de error útil cuando ocurren excepciones, como antes:

```
s = SoloPares()
s.append("una cadena")
```

Resulta en:

```
---------------------------------------------------------------------------
------
TypeError Traceback (most recent call last)<ipython-input-133-
21a7630ea1e7> in <module>      1 s = SoloPares()----> 2 s.append("una
cadena")<ipython-input-130-b40edd70d5e3> in append(self, entero)
2        def append(self, entero):      3          if not
isinstance(entero, int):----> 4            raise
TypeError("Soloenteros pueden ser agregados")            5
if entero % 2:      6                       raise ValueError("Solo
numeros pares pueden ser agregados")TypeError: Soloenteros pueden ser
agregados
```

```
s.append(3)
```

Resulta en:

```
---------------------------------------------------------------------------
------
ValueError Traceback (most recent call last)
<ipython-input-134-abd9a893f30f> in <module>
----> 1 s.append(3)

<ipython-input-130-b40edd70d5e3> in append(self, entero)
         4              raise TypeError("Soloenteros pueden ser
agregados")
      5          if entero % 2:
----> 6              raise ValueError("Solo numeros pares pueden ser
agregados")
      7          super().append(entero)

ValueError: Solo numeros pares pueden ser agregados
```

¿Qué sucede cuando ocurre una excepción?

Cuando se genera una excepción, parece que detiene la ejecución del programa inmediatamente. Las líneas que se suponía que iban a ejecutar después de la excepción no se ejecutan y, a menos que se resuelva la excepción, el programa se cerrará con un mensaje de error. Eche un vistazo a esta sencilla función que ya usamos anteriormente:

```
def no_return():
    print("Voy a levantar un excepcion")
    raise Exception("Esta siempre se levanta")
    print("Esta linea nunca se ejecuta")
    return "Yo no voy a retornar"
```

Si ejecutamos esta función, vemos que se ejecuta la primera llamada a **print** y luego se genera la excepción. La segunda llamada a **print** nunca se ejecuta y la declaración de retorno tampoco se ejecuta nunca:

```
no_return()
```

Resulta en:

```
Voy a levantar un excepcion
---------------------------------------------------------------------------
------
Exception  Traceback (most recent call last)
<ipython-input-136-7cb40636301c> in <module>
----> 1 no_return()

<ipython-input-135-9f23ed92097c> in no_return()
      1 def no_return():
      2     print("Voy a levantar un excepcion")
----> 3     raise Exception("Esta siempre se levanta")
      4     print("Esta linea nunca se ejecuta")
      5     return "Yo no voy a retornar"

Exception: Esta siempre se levanta
```

Además, si tenemos una función que llama a una segunda función que genera una excepción, no se ejecutará nada en la primera función después del punto donde se llamó a la segunda función. La generación de una excepción detiene toda la ejecución en la pila de llamadas de función hasta que se maneja o obliga al intérprete a salir. Para demostrarlo, agreguemos una segunda función que llame a la primera:

```
def llamar_al_exceptor():
```

```
    print("llamar_al_exceptor comienza aca")
    no_return()
    print("una excepcion fue lanzada...")
    print("...asi que estas lineas no corren")
```

Cuando llamamos a esta función, vemos que la primera declaración **print** se ejecuta al igual que la primera línea en la función **no_return**. Pero una vez que se genera la excepción, nada más se ejecuta:

```
llamar_al_exceptor()
```

Resulta en:

```
llamar_al_exceptor comienza aca
Voy a levantar un excepcion----------------------------------------
--------------------------------
Exception Traceback (most recent call last)
<ipython-input-138-3ac227b2fc1d> in <module>
----> 1 llamar_al_exceptor()

<ipython-input-137-6999b2658827> in llamar_al_exceptor()
      1 def llamar_al_exceptor():
      2     print("llamar_al_exceptor comienza aca")
----> 3     no_return()
      4     print("una excepcion fue lanzada...")
      5     print("...asi que estas lineas no corren")

<ipython-input-135-9f23ed92097c> in no_return()
      1 def no_return():
          2     print("Voy a levantar un excepcion")
----> 3     raise Exception("Esta siempre se levanta")
      4     print("Esta linea nunca se ejecuta")
          5     return "Yo no voy a retornar"

Exception: Esta siempre se levanta
```

El intérprete no está tomando un atajo y saliendo inmediatamente; la excepción se puede manejar dentro de cualquier método. De hecho, las excepciones pueden manejarse en cualquier nivel después de que se lanzan inicialmente. Si miramos la salida de la excepción (llamada traceback) de abajo hacia arriba, vemos ambos métodos listados. Dentro de **no_return**, la excepción se genera inicialmente. Luego, justo encima de eso, vemos que dentro de **llamar_al_exceptor,** esa molesta función **no_return** fue llamada y la excepción "burbujeó" en el método de llamada. A partir de ahí subió un nivel más para el intérprete principal, que finalmente imprimió el rastreo (traceback).

3.5.1.6. Excepciones definidas por el usuario

A menudo, cuando queremos generar una excepción, encontramos que ninguna de las excepciones integradas es lo que necesitamos. Afortunadamente, es muy fácil definir nuestras propias excepciones. El nombre de la clase generalmente está diseñado para comunicar lo que salió mal y podemos proporcionar argumentos arbitrarios en el inicializador para agregar información adicional.

Todo lo que tenemos que hacer es heredar de la clase **Exception**. ¡Ni siquiera tenemos que agregar ningún contenido a la clase! Por supuesto, podemos extender **BaseException** directamente, pero luego no será detectado por cláusulas genéricas **except Exception**.

Sin más preámbulos, aquí hay una simple excepción que podríamos usar en una aplicación bancaria:

```
class RetiroInvalido(Exception):
    pass

raise RetiroInvalido("Usted no tiene 100 pesos en su cuenta")
```

La última línea ilustra cómo generar la excepción recién definida. Podemos pasar un número arbitrario de argumentos (a menudo un mensaje de cadena, pero se puede almacenar cualquier objeto útil) a la excepción. El método **Exception.__init__** está diseñado para aceptar cualquier argumento y almacenarlo como una tupla en un atributo llamado **args**. Esto hace que las excepciones sean más fáciles de definir sin necesidad de sobreescribir **__init__**.

Por supuesto, si queremos personalizar el inicializador, podemos hacerlo. Aquí hay una excepción cuyo inicializador exige el saldo actual y la cantidad que el usuario quería retirar. Además, agrega un método para calcular cuánto más sobre el retiro fue:

```
class RetiroInvalido(Exception):
    def __init__(self, balance, monto):
        super().__init__("la cuenta no tiene ${}"
            .format(monto))
        self.monto = monto
        self.balance = balance

    def exceso(self):
        return self.monto - self.balance

raise RetiroInvalido(25, 50)
```

La declaración **raise** al final ilustra cómo construir la excepción. Como puede ver, podemos hacer cualquier cosa con la excepción de lo que haríamos con otros objetos. Podríamos capturar una excepción y pasarla como un objeto de trabajo,

aunque es más común incluir una referencia al objeto de trabajo como un atributo en una excepción y pasarla en su lugar.

Así es como manejaríamos un **RetiroInvalido** si se generó uno:

```
try:
    raise RetiroInvalido(25, 50)
except RetiroInvalido as e:
    print("Lo sentimos, pero su retiro es mas que su balance por ${}"
            .format(e.exceso()))
```

Aquí vemos un uso válido de la palabra clave **as**. Por convención, la mayoría de los codificadores de Python nombran la variable de excepción e, aunque, como de costumbre, puede llamarla **ex**, **exception** o **michael_jackson** si lo prefiere.

Ahora que tenemos control total sobre las definiciones de excepción, incluido el inicializador, los atributos y los métodos, y podemos acceder a una instancia de excepción cuando se está manejando, tenemos poder absoluto sobre la información que se pasa con una excepción.

Hay muchas razones para definir nuestras propias excepciones. A menudo es útil agregar información a la excepción o registrarla de alguna manera. Pero la utilidad de las excepciones personalizadas realmente sale a la luz cuando se crea un framework, biblioteca o API que está destinada al acceso de otros usuarios. En ese caso, tenga cuidado de asegurarse de que su código genere excepciones que tengan sentido para el programador cliente, que sean fáciles de manejar y describan claramente lo que salió mal para que puedan solucionarlo (si es un error en su código) o manejarlo (si es una situación de la que deben ser informados).

3.5.1.7. Acciones de limpieza con finally

Hemos visto varias variaciones en la sintaxis para manejar excepciones, pero aún no sabemos cómo ejecutar el código independientemente de si se ha producido o no una excepción. Tampoco podemos especificar código que deba ejecutarse **solo** si **no** ocurre una excepción. Dos palabras clave más, **finalmente** y **else**, pueden proporcionar las piezas que faltan. Ninguna acepta argumentos adicionales. El siguiente ejemplo elige aleatoriamente una excepción para lanzarla y la genera. Luego, se ejecuta un código de manejo de excepciones no tan complicado que ilustra la sintaxis introducida con anterioridad:

```
import random
algunas_exceptions = [ValueError, TypeError, IndexError, None]
try:
    alternativa = random.choice(algunas_exceptions)
    print("lanzando {}".format(alternativa))
    if alternativa:
        raise alternativa("An error")
except ValueError:
```

```
    print("Atrapando un ValueError")
except TypeError:
    print("Atrapando un TypeError")
except Exception as e:
    print("Atrapando algun otro error: %s" % (e.__class__.__name__))
else:
    print("Este codigo es llamado si no hay excepcion")
finally:
    print("Este codigo de limpieza siempre es llamado")
```

Si ejecutamos este ejemplo, que ilustra casi todos los escenarios de manejo de excepciones imaginables, unas cuantas veces, obtendremos resultados diferentes cada vez, según la excepción que elija el azar. Aquí hay algunos ejemplos de ejecuciones:

```
lanzando <class 'ValueError'>
Atrapando un ValueError
Este codigo de limpieza siempre es llamado

lanzando None
Este codigo es llamado si no hay exception
Este codigo de limpieza siempre es llamado

lanzando <class 'TypeError'>
Atrapando un TypeError
Este codigo de limpieza siempre es llamado
```

Tenga en cuenta cómo se ejecuta la declaración **print** en la cláusula **finally** sin importar qué suceda. Esto es extremadamente útil, por ejemplo, cuando necesitamos limpiar una conexión de base de datos abierta, cerrar un archivo abierto o enviar un saludo de cierre a través de la red cuando nuestro código ha terminado de ejecutarse, incluso si se ha producido una excepción. Esto también se puede aplicar de formas interesantes si estamos dentro de una cláusula **try** cuando regresamos **(return)** de una función; **finally** aún se ejecutará al regresar.

También preste atención a la salida cuando no se genere ninguna excepción; se ejecutan tanto las cláusulas **else** como las **finally**. La cláusula **else** puede parecer redundante, ya que parece código que debe ejecutarse solo cuando no se genera ninguna excepción y se podría colocar después de todo el bloque **try...except**. Sin embargo, este código aún se ejecutaría en el caso en el que se detecta y gestiona una excepción.

Cualquiera de las cláusulas **except**, **else**, y **finally** se pueden omitir después de un bloque **try** (aunque, **else**, por sí mismo no es válido). Si incluye más de una, las cláusulas **except** deben ir primero, luego la cláusula **else**, con la cláusula **finally** al final. El orden de las cláusulas **except** normalmente va del más específico al más genérico.

3.5.1.7. Acciones de limpieza predefinidas

Algunos objetos definen acciones de limpieza estándar que se realizarán cuando el objeto ya no sea necesario, independientemente de si la operación que utiliza el objeto tuvo éxito o no. Mire el siguiente ejemplo, que intenta abrir un archivo e imprimir su contenido en la pantalla. (Veremos archivos en la siguiente clase, pero este ejemplo es una buena aproximación al tema)

```
for linea in open("miarchivo.txt"):
    print(linea, end="")
```

El problema con este código es que deja el archivo abierto durante un período de tiempo indeterminado después de que esta parte del código ha terminado de ejecutarse. Esto no es un problema en scripts simples, pero puede ser un problema para aplicaciones más grandes. A propósito, la instrucción **with** permite que objetos como archivos se utilicen de una manera que garantice que siempre se limpien de manera rápida y correcta.

Una solución para garantizar que el archivo fue cerrado una vez ocupado podría ser la siguiente:

```
try:
    archivo = open("miarchivo.txt", "wb")
    for linea in archivo:
            print(linea, end="")
except Exception as e:
    print("problemas con el archivo")
finally:
    archivo.close()
```

3.5.2. Referencias.

[1] Mark Lutz, Python Pocket Reference, Fifth Edition 2014.

[2] Matt Harrison, Ilustrated Guide to Python3, 2017.

[4] Magnus Lie Hetland, Beginning Python: From Novice to Professional, 2017.

[5] Python Tutorial.
https://www.w3schools.com/python/