Clase Nº 7

Manejo de Excepciones

Concepto

- Una **excepción** es la indicación de un problema ocurrido durante la ejecución de un programa.
- Dado que la "regla" es que los programas se ejecuten en forma normal, la "excepción" es que ocurra algún problema.

Concepto

- Históricamente parte del código del programa se dedicaba a contemplar posibles situaciones de error.

```
if cantidad != 0:
    promedio = suma / cantidad
else:
    print("No se puede dividir por cero")
```


Concepto

- Con el manejo de excepciones es posible independizar la lógica del programa del código de control de errores, es decir que evita tener que mezclarlos.
- De este modo las aplicaciones son más robustas y veloces.

Concepto

- El control de errores tradicional es **preventivo**: Evita que el error ocurra.
- El control de errores mediante excepciones es **correctivo**: Intenta solucionar el problema después de haber ocurrido.

Tipos de excepciones

- Cada error genera un tipo de excepción distinto, que puede ser *capturado* o *atrapado* por el programador:
  - División por cero
  - Uso de variables no inicializadas
  - Subíndices fuera de rango
  - Tratar de convertir a entero o a float un valor no numérico

Tipos de excepción

```
>>> print(1/0)
Traceback (most recent call last):
   File "<pyshell>", line 1, in <module>
ZeroDivisionError: division by zero
```

- En este caso la excepción es ***ZeroDivisionError***.

Tipos de excepción

- Para capturar excepciones se utilizan los bloques ***try/except***. El código ubicado entre try y except se denomina ***bloque protegido***.

```
try:
   <código que puede provocar errores>
except <excepción>:
   <código de tratamiento del error>
```

Ejemplo 1

Impedir que un programa falle debido a una división por cero.


```
try:
    a = int(input("Dividendo ? "))
    b = int(input("Divisor ? "))
    cociente = a // b
    resto = a % b
    print( )
    print("Cociente:", cociente)
    print("Resto: ", resto)
except ZeroDivisionError:
    print("No se puede dividir por cero.")
```

Funcionamiento

- **Primero se ejecuta el bloque protegido, es decir el código ubicado entre *try* y *except* que delimita la zona en la que pueden llegar a ocurrir los errores.**
- **Si no ocurren errores el bloque except se saltea y la ejecución continúa luego de este bloque except.**

Funcionamiento

- **Si ocurre un error durante la ejecución del bloque protegido, el resto de este bloque se omite.**
- **Si el tipo de error coincide con la excepción detallada en el except se ejecuta este bloque, que es donde se remediará el problema. Luego el programa continúa normalmente.**

Funcionamiento

- Si ocurre un error que no coincide con la excepción detallada en el except, ésta se traslada a otros bloques *try/except* exteriores.
- Si no se encuentra nada que la maneje, será considerada como una ***excepción no manejada*** y el programa se detendrá con el mensaje de error correspondiente.

Funcionamiento

- Pueden escribirse varios bloques *except* para manejar distintos tipos de excepciones, los que serán analizados en el orden en que se encuentran.
- Cada tipo de excepción debe ser manejado por un **único bloque *except***. No puede haber más de uno para la misma excepción.

Funcionamiento

- Un *except* sin tipo de excepción capturará cualquier error que pudiera producirse.
- No se recomienda hacerlo porque no permite diferenciar los errores. ▼
- Se admite escribirlo debajo de otros except, como medida de último recurso.

Ejemplo 2

Impedir que un programa falle
debido a una división por cero
o por el ingreso de datos inválidos.

El programa finaliza
con un mensaje de error.

```
try:
    a = int(input("Dividendo? "))
    b = int(input("Divisor? "))
    cociente = a // b
    resto = a % b
    print( )
    print("Cociente:", cociente)
    print("Resto: ", resto)
except ZeroDivisionError:
    print("No se puede dividir por cero")
except ValueError:
    print("Sólo se permiten números enteros")
except:
    print("Error desconocido. Intente más tarde.")
```


Observaciones

- Los errores de sintaxis no pueden ser atrapados mediante excepciones, porque la sintaxis es verificada durante el análisis sintáctico del programa y no durante la ejecución.

Observación

- Si se le va a dar el mismo tratamiento a más de un tipo de error, puede usarse el mismo bloque *except*:

```
except (ValueError, ZeroDivisionError):
    print("Datos inválidos")
```

- En este caso los nombres de las excepciones deben escribirse entre paréntesis.

Ejemplo 3

Utilizar manejo de excepciones para continuar normalmente la ejecución de una función luego de producido un error.

```
def leerentero(msj="Ingrese un entero: "):
    ''' Función para ingresar un número entero '''
    while True:
        try:
            n = int(input(msj))
            break
        except ValueError:
            print("Dato inválido.")
            print("Intente nuevamente.")
    return n
```



```
# Programa principal
while True:
    try:
        a = leerentero("Dividendo? ")
        b = leerentero("Divisor? ")
        cociente = a // b
        resto = a % b
        break
    except ZeroDivisionError:
        print("No se puede dividir por cero.")
        print("Intente nuevamente.")
print( )
print("Cociente:", cociente)
print("Resto: ", resto)
```

Ejemplo 4

Escribir un programa para imprimir números enteros a partir del 1, que no pueda ser interrumpido con Ctrl-C.


```
contador = 1
while True:
    try:
        print(contador, end=" ")
        contador = contador + 1
    except KeyboardInterrupt:
        pass
```

Nota: Cerrar la consola de Python para detenerlo o presionar "Stop" en Thonny.

Instrucción raise

- Se utiliza cuando se desea provocar una excepción.
- Puede ir seguida del nombre de la excepción a producir.
- Si no se detalla el tipo de excepción se relanza la última excepción producida.

Ejemplo 5

Darle al usuario
la posibilidad de abortar
la ejecución del programa
en caso de producirse un error.

```
def leerentero(msj="Ingrese un entero: "):
    while True:
        try:
            n = int(input(msj))
            break
        except ValueError:
            print("Dato inválido.")
            a = input("Desea ingresarlo otra vez? (S/N): ")
            if a.upper( ) == "N":
                raise
    return n
```


Cláusula else:

- Esta cláusula es opcional en un bloque try/except.
- Debe escribirse después de todos los except.
- Sólo será ejecutada cuando el bloque try precedente haya finalizado en forma normal, es decir sin haberse producido excepciones.

Ejemplo 6

ALTERNATIVA AL EJEMPLO Nº 3

Utilizar manejo de excepciones para continuar normalmente la ejecución de una función luego de producido un error.


```
def leerentero(msj="Ingrese un entero: "):
    while True:
        try:
            n = int(input(msj))
        except ValueError:
            print("Dato inválido.")
            print("Intente nuevamente.")
        else:
            break
    return n
```

Cláusula finally

- La cláusula finally es opcional, y se escribe luego de un bloque try o de un bloque except.
- Su propósito es garantizar que una porción de código se ejecute siempre, sin importar si hubo errores o no.

Cláusula finally

- Se utiliza en tareas de limpieza, para liberar recursos previamente asignados y así evitar que los mismos se agoten, o para continuar en forma prolija.

Cláusula finally

Recursos que se protegen con finally:

- **Memoria**
- **Conexiones de red**
- **Sesiones con bases de datos**
- **Archivos temporales**
- **Opciones de menú**
- **Cursores del mouse**
- **etc.**

Cláusula finally

```
try:
    <bloque protegido>
except <tipo de excepción>:
    <código de control de errores>
finally:
    <código de saneamiento>
```

Cláusula finally

- finally puede utilizarse solo con el bloque try, sin necesidad del except.
- Si se usó *else:* en el bloque try, finally va después de éste.
- El uso de la cláusula finally se verá en detalle en la próxima clase, vinculado al tema Archivos.

Ejemplo 7

Imprimir una matriz por pantalla, utilizando manejo de excepciones para darle el formato apropiado.

```
def imprimirmatriz(mat):
    filas = len(mat)
    columnas = len(mat[0])
    elementos = filas * columnas
    f = 0
    c = 0
    for i in range(elementos):
        try:
            print("%3d" %mat[f][c], end="")
        except IndexError:
            f = f + 1
            c = 0
            print( )
            print("%3d" %mat[f][c], end="")
        finally:
            c = c + 1
```


Instrucción assert

- La instrucción ***assert*** incorpora al programa un control expresado como una condición.
- Este control es una afirmación que realiza el programador, y que el programa debe superar para poder continuar.

assert <condición>

Instrucción assert

- Si la afirmación es cierta (condición verdadera), *no ocurre nada*.
- Si la condición es falsa, se produce una excepción **AssertionError**.

Ejemplo 8

Verificar el rango de un número de mes a través de la instrucción assert.

```
while True:
    try:
        mes = int (input("Ingrese el mes: "))
        assert 1 <= mes <= 12
        break
    except ValueError:
        print("Sólo se permiten números.")
    except AssertionError:
        print("El mes debe estar entre 1 y 12.")
    print("Intente nuevamente.")
print("El mes ingresado es", mes)
```


Instrucción assert

- assert puede ser reemplazado fácilmente por *if* y *raise*:

```
assert 1 <= mes <= 12
```

equivale a

```
if not (1 <= mes <=12):
    raise AssertionError
```

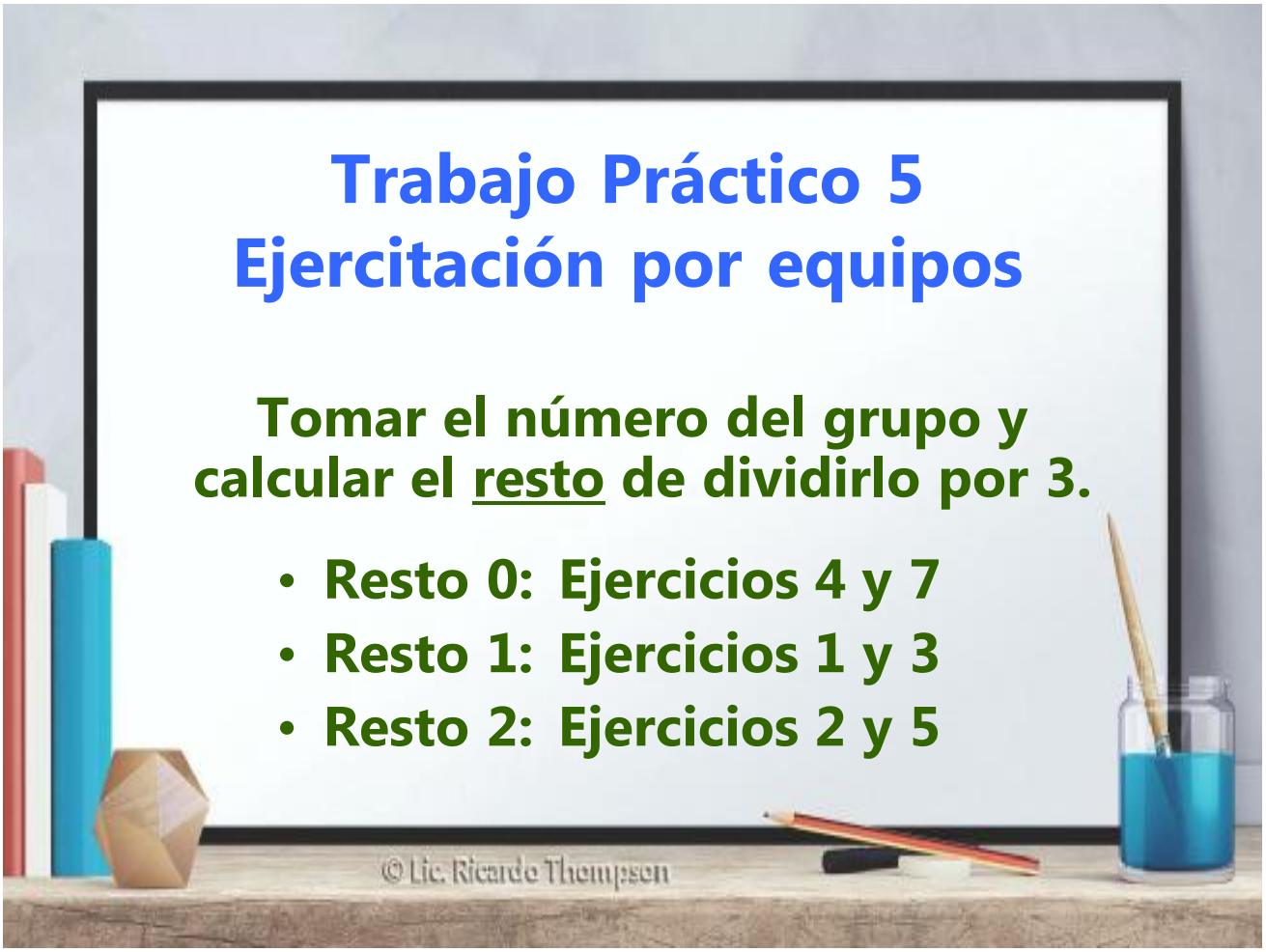
Instrucción assert

- **Si se incluyen varios *assert* en el mismo programa, puede usarse un mensaje para diferenciarlos:**

```
palabra  = input("Ingrese una palabra: ")
try:
    assert len(palabra)>5, "Palabra muy corta"
    assert palabra.isalpha( ), "Sólo se permiten letras"
except AssertionError as mensaje:
    print(mensaje)
```


Ejercitación

- **Práctica 5: Completa**


Trabajo Práctico 5
Ejercitación por equipos
Tomar el número del grupo y calcular el resto de dividirlo por 3.
• Resto 0: Ejercicios 4 y 7
• Resto 1: Ejercicios 1 y 3
• Resto 2: Ejercicios 2 y 5
© Lic. Ricardo Thompson