UTN AVELLANEDA
Tecnicatura Universitaria en Programación

Excepciones

Programación y Laboratorio I

Versión ‘22

UTN AVELLANEDA
Tecnicatura Universitaria en Programación

Excepciones

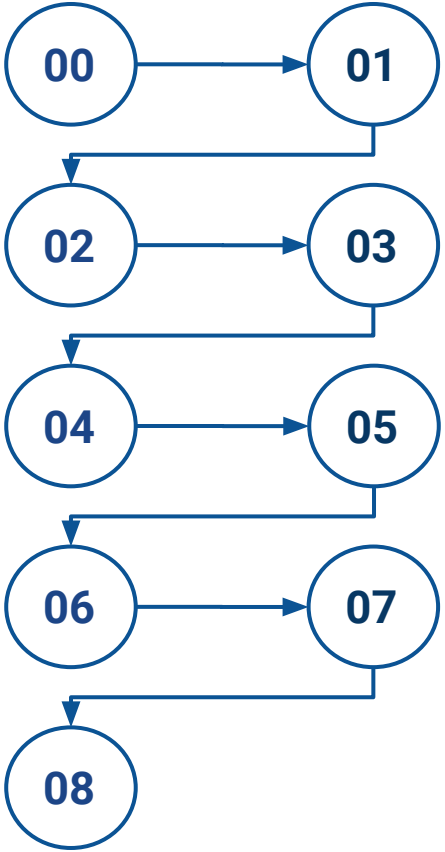
¿Qué es una excepción?

Capturar excepciones

Bloque else

Excepciones propias

Validar con excepciones



Tipos de excepciones

Lanzar excepciones

Bloque finalizador

Assert

¿Qué es una excepción?

Los errores detectados durante la ejecución se llaman excepciones y salvo que tratemos este error, el programa se cerrará.

```
a = 4; b = 0
print(a/b)

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

Tipos de excepciones

Existen diferentes tipos de excepciones para los distintos tipos de errores e inclusive la posibilidad de declarar nuevas excepciones.

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

UTN AVELLANEDA
Tecnicatura Universitaria en Programación

Tipos de excepciones

```
+-- Exception
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      |    +-- ModuleNotFoundError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      |    +-- UnboundLocalError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      |    +-- NotImplementedError
      |    +-- RecursionError
      +-- SyntaxError
      |    +-- IndentationError
      |         +-- TabError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      |    +-- UnicodeError
      |         +-- UnicodeDecodeError
      |         +-- UnicodeEncodeError
      |         +-- UnicodeTranslateError
```

Capturar excepciones

Cuando una porción de código quiere indicar que ocurrió un error, debe lanzar una excepción.

```
int("¡Hola, mundo!")
ValueError: invalid literal for int() with base 10: '¡Hola, mundo!'
```

Si una excepción es lanzada y ningún código la captura, el programa finaliza

Capturar excepciones

Es posible definir múltiples bloques de código para las distintas excepciones y por último capturar cualquier excepción.

```
try:
    int("¡Hola, mundo!")
except ValueError:
    print("No puede convertirse a un entero.")
except TypeError:
    print("No es una cadena.")
except:
    print("Es otro tipo de error")
```

Lanzar excepciones

Al escribir funciones propias surge la necesidad de poder lanzar excepciones.

La declaración **raise** permite forzar a que ocurra una excepción específica.

```
raise ValueError
```

Lanzar excepciones

Al escribir funciones propias surge la necesidad de poder lanzar excepciones.

La declaración **raise** permite forzar a que ocurra una excepción específica.

```
raise ValueError
```

Bloque else

El bloque **else** se ejecutará si no ha ocurrido ninguna excepción.

```
def divide(x, y):
    try:
        resultado = x / y
    except ZeroDivisionError:
        print("No es posible dividir por cero!")
    else:
        print("El resultado es", resultado)

divide(2,2) #El resultado es 1.0
divide(2,0) #No es posible dividir por cero!
```

Bloque finalizador

Como se puede ver, el bloque **finally** siempre se ejecuta. Puede ser útil para liberar recursos externos.(como archivos o conexiones de red).

```
def divide(x, y):
    try:
        resultado = x / y
    except ZeroDivisionError:
        print("No es posible dividir por cero!")
    else:
        print("El resultado es", resultado)
    finally:
        print("Se ejecuto finally")

divide(2,2) #El resultado es 1.0  Se ejecutó finally
divide(2,0) #No es posible dividir por cero! Se ejecutó finally
```

Excepciones propias

Los programas pueden nombrar sus propias excepciones creando una nueva clase excepción. Las excepciones, típicamente, deberán derivar de la clase Exception, directa o indirectamente.

```
# Creamos una excepción
class MiExcepcion(Exception):
   pass

# Lanzamos
raise MiExcepcion
```

Assert

Si la expresión contenida dentro del **assert** es False, se lanzará una excepción. Es posible añadir un texto con información relevante

```
assert 1==2, "El assert falló"
```

cuidado, ya que la expresión anterior no es equivalente a la siguiente

```
assert (1==2, "El assert falló")
```

Validar con excepciones

Esta función intenta obtener un valor entero ingresado por el usuario.

```
def leer_entero(intentos):
    retorno = None
    for i in range(intentos):
        valor = input("Ingrese un número entero: ")
        try:
            valor = int(valor)
            retorno = valor
            break
        except ValueError:
            print("Error se debe ingresar un número entero")
    return retorno
```