# Tema 2. Tratamiento de errores, excepciones.

Vicente Benjumea García

Programación-II

Departamento de Lenguajes y Ciencias de la Computación.
E.T.S.I. Informática. Univ. de Málaga.

## Tema 2. Tratamiento de errores, excepciones.

- Software tolerante a fallos. El concepto de excepción.
- Excepciones predefinidas.
- Elevación y lanzamiento de excepciones.
- Propagación de excepciones.
- Captura y tratamiento de excepciones.
- El flujo de ejecución en presencia de excepciones.
- Definición de nuevas excepciones.
- Comportamiento en situaciones excepcionales.

# Software tolerante a fallos. El concepto de excepción

## Errores

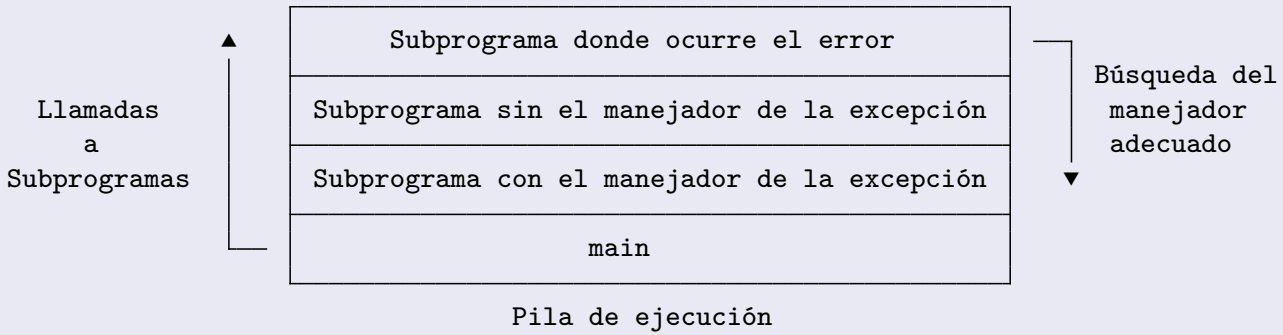
- Durante la ejecución de un programa se pueden producir **errores** con diversos **niveles** de severidad:
  - Índice fuera de límites, división por cero, fichero que no existe, conexión de red que falla, etc.
- Estos **errores**, producidos durante la ejecución de un programa, pueden provocar que el programa en su totalidad, o en parte, **no pueda llevar a cabo su cometido**. También es posible que se pueda **recuperar** la situación de error, y finalmente **completar su tarea** de forma satisfactoria.

## El sistema de excepciones

- El mecanismo de **excepciones** proporciona una forma clara de **detectar**, **informar** y **gestionar** los posibles **errores** que surjan durante la ejecución del programa, sin oscurecer el código.
- El sistema de excepciones nos permite **diferenciar el código** de procesamiento que resuelve un determinado problema, del código encargado de la gestión y recuperación de errores.

# Software tolerante a fallos. El concepto de excepción

- Una **excepción** es un evento (una señal) que **interrumpe**, como consecuencia de un **error** (situación excepcional), el flujo normal de ejecución de las instrucciones durante la **ejecución** de un programa.
- Cuando ocurre un error, o una situación anómala, durante la ejecución de un programa, entonces se **crea un objeto** excepción, se **eleva** (`raise`) y **lanza** al sistema de ejecución, y se **interrumpe** la ejecución del programa.
  - Este objeto excepción contiene **información sobre el error** producido, incluido su tipo y el estado del programa donde ocurrió.
- La excepción se va **propagando** a través de todos los subprogramas (funciones y métodos) correspondientes de la **pila de ejecución**, hasta encontrar un bloque de código adecuado que **capture** y maneje la excepción, y se **reanude** la ejecución normal del flujo de instrucciones del programa.
- Si el sistema **no encuentra** un manejador adecuado, entonces el programa **aborta** (**termina**) su ejecución (mostrando información detallada sobre ella).



# Software tolerante a fallos. El concepto de excepción

- Las excepciones se deben analizar desde **tres puntos de vista**:

  1. El que **eleva** y **lanza** la excepción:
     Durante la ejecución de código, se encuentra una **situación anómala** que **no sabe cómo resolver** e impide que el código **cumpla** su tarea, por lo tanto **crea** una excepción y la **eleva** y **lanza**, para informar al *nivel superior* de la situación anómala y de que la tarea no pudo completarse.

  2. El que **propaga** la excepción:
     Durante la ejecución de código, **recibe** una excepción, **no sabe como tratarla** y recuperar el error, por lo que el código no puede completar su tarea, y por lo tanto, **propaga la excepción** para informar al *nivel superior* de la situación anómala y de que la tarea no pudo completarse.

  3. El que **captura** y **maneja** la excepción.
     Durante la ejecución de código, **recibe** una excepción, y **sí sabe cómo resolver** la situación que provoca ese **tipo** de excepción, por lo que **dispone de un tratamiento** para ella, y recupera el error para poder completar su tarea.

- Un mismo código puede actuar desde los tres puntos de vista
  - Captura unas excepciones, propaga, eleva y lanza otras.

# Excepciones predefinidas (I)

## Jerarquía con las excepciones más habituales (I)

- Estas excepciones se elevan como consecuencia de **errores de programación** en nuestro código, por lo que usualmente, **no debemos capturarlas**, sino que **deberemos arreglar el error en nuestro código**.

```
BaseException                   ->  clase base de todas las excepciones
 ├── KeyboardInterrupt          ->  interrupción de ejecución del teclado (Ctrl-C)
 ├── SystemExit                 ->  fin del programa/sistema
 └── Exception                  ->  clase base de todas las excepciones capturables
      ├── AssertionError        ->  error, aserto no válido                          assert(x > 0)
      ├── AttributeError        ->  error, acceso a atributo/miembro no existente    xxx.zzz
      ├── ImportError           ->  error de importación
      │    └── ModuleNotFoundError ->  error de importación, módulo no encontrado    import zzz
      ├── MemoryError           ->  error, no hay memoria suficiente
      ├── NameError             ->  error, nombre no definido/encontrado             print(zzz)
      ├── SyntaxError           ->  error sintáctico                                 xxx zzz
      │    └── IndentationError ->  error sintáctico de indentación
      │         └── TabError    ->  error sintáctico de indentación por tabulador
      └── Warning               ->  clase base de todos los avisos de alerta
           ├── RuntimeWarning   ->  aviso de alerta de ejecución
           ├── SyntaxWarning    ->  aviso de alerta sintáctico
           └── UserWarning      ->  aviso de alerta de usuario
```

# Excepciones predefinidas (II)

## Jerarquía con las excepciones más habituales (II)

- Estas excepciones se elevan debido a **errores de ejecución** en nuestro código.
  - Si corresponden a errores de programación, deberemos **arreglar el error en nuestro código**.
  - En otro caso, **capturaremos** el error para **recuperar** la ejecución del programa.
- Además, cuando nuestro código se encuentre una **situación de error**, nuestro código **creará**, **elevará** y **lanzará** alguna de estas excepciones.

```
BaseException                -> clase base de todas las excepciones
└─ Exception                 -> clase base de todas las excepciones capturables
   ├─ ArithmeticError        -> error aritmético
   │  ├─ OverflowError       -> error aritmético, desbordamiento        16.0**300
   │  └─ ZeroDivisionError   -> error aritmético, división por cero     5/0
   ├─ EOFError               -> no quedan datos que extraer para input
   ├─ LookupError            -> acceso por clave o índice no adecuado
   │  ├─ IndexError          -> acceso por índice fuera de límites      lista[20]
   │  └─ KeyError            -> acceso por clave no encontrada          dicc["hola"]
   ├─ OSError                -> error del sistema operativo
   │  ├─ FileExistsError     -> el fichero ya existe                    open("datos.txt", "x")
   │  ├─ FileNotFoundError   -> fichero no encontrado                   open("datos.txt", "r")
   │  └─ PermissionError     -> error, no tiene permiso adecuado        open("datos.txt", "w")
   ├─ RuntimeError           -> error general en tiempo de ejecución
   │  ├─ NotImplementedError -> error de ejecución, opción no implementada
   │  └─ RecursionError      -> error de ejecución, recursión infinita
   ├─ StopIteration          -> fin de iteración en iterador/generador  next(it)
   ├─ TypeError              -> el valor no es del tipo adecuado        "hola" + 30
   └─ ValueError             -> el valor no es adecuado                 int("xxx")
```

# Elevación y lanzamiento de excepciones

- Cuando nuestro código se encuentra una **situación de error** que no le permite realizar su tarea, entonces nuestro código **creará**, **elevará** y **lanzará** una excepción.
  - La sentencia `raise` permite **lanzar** un *objeto Excepción*. Para ello, crearemos un objeto del tipo *excepción* adecuado al tipo de error que queramos informar, proporcionando la información adecuada en el constructor del objeto excepción.
  - **Interrumpe** el flujo de ejecución normal de instrucciones y **se procede a la búsqueda** de un manejador que capture ese **tipo de excepción**.

```
def calc_media(lista: list[int|float]) -> float:
    """devuelve la media de los números de la lista"""
    if not isinstance(lista, list):
        raise TypeError("argumento no es de tipo lista")
    if len(lista) == 0:                  # si la lista está vacía
        raise ValueError("lista vacía") # entonces no se puede calcular la media
    suma = 0.0                           # por lo que el resto del código no se ejecuta
    for x in lista:
        suma += x
    return suma / len(lista)
```

# Propagación de excepciones

- Si durante la ejecución de un determinado código se lanza (o se recibe) una excepción, y la excepción **no se captura**, entonces la excepción será **automáticamente propagada** al nivel superior.
  - **Interrumpe** el flujo de ejecución normal de instrucciones y **se procede a la búsqueda** de un manejador que capture ese **tipo de excepción**.

```
import math

def calc_desv_tipica(lista: list[int|float]) -> float:
    """devuelve la desviación típica de los números de la lista"""
    media = calc_media(lista)          # Si se recibe una excepción, entonces la propaga
    suma = 0.0                         # y el resto del código no se ejecuta
    for x in lista:
        suma += (x - media)**2;
    return math.sqrt(suma / len(lista))
```

- Si la invocación a `calc_media` y su ejecución lanzase una excepción, como no sería capturada dentro de la función `calc_desv_tipica`, entonces la excepción se propagaría al nivel superior, y el resto del código de la función `calc_desv_tipica` **no se ejecutaría**.

# Captura y tratamiento de excepciones

## La sentencia `try-except-else-finally`

- Las excepciones que se lancen durante la ejecución del código del bloque `try` serán capturadas en el bloque `except` correspondiente, o serán propagadas.
- El bloque `except` define el código que se ejecuta si dentro del bloque `try` correspondiente **se ha lanzado una excepción** especificada en su lista de excepciones capturadas, o de cualquier clase derivada de ellas.
  - El bloque `except` puede mostrar información en consola, pero también puede **recuperar la situación de error**.
- El bloque `else` define el código que **se ejecuta si no se ha lanzado ninguna excepción** en la ejecución del bloque `try` correspondiente.
- El bloque `finally` define un código que **se ejecuta siempre** que se ejecuta el bloque `try` correspondiente, incluso aunque se lancen o propaguen excepciones.

- Un bloque `try` puede tener cero o múltiples bloques `except`, pero sólo uno o ningún bloque `else`, y sólo uno o ningún bloque `finally`.

- Los bloques `except` se **comprueban según el orden** especificado en el programa, según el tipo especificado y el tipo del objeto excepción a capturar.

# Captura y tratamiento de excepciones

## La sentencia try-except-else-finally

```
import logging

# logging.basicConfig(filename="errores.log", filemode="w", level=logging.WARNING)

try:
    # Las excepciones que se lancen podrían ser capturadas si el tipo es adecuado
except (TypeError, ValueError) as exc:
    # Captura y tratamiento de excepciones TypeError, ValueError y derivadas
    logging.exception(exc)                # registra el error y la traza de ejecución
except ArithmeticError as exc:
    # Captura y tratamiento de excepciones ArithmeticError y derivadas
    logging.error(f"Error: [{exc!r}]")    # registra el error
except LookupError as exc:
    # Captura y tratamiento de excepciones LookupError y derivadas
    print(f"Error: [{exc!r}] [{exc.args}]")    # muestra el error y tupla con argumentos
except OSError as exc:
    # Captura y tratamiento de excepciones OSError y derivadas
except RuntimeError as exc:
    # Captura y tratamiento de excepciones RuntimeError y derivadas
except StopIteration as exc:
    # Captura y tratamiento de excepciones StopIteration y derivadas
else:
    # Código que se ejecuta si no se lanza excepción en la ejecución del bloque try
finally:
    # Código que se ejecuta siempre que se ejecuta el bloque try (limpieza de recursos)
```

# Captura y tratamiento de excepciones

## El módulo `logging`

- El módulo `logging` proporciona varias funciones que permiten registrar mensajes con niveles de error, crítico, aviso, información y depuración, en varios medios de registro, tales como la consola y ficheros.

- `logging.exception(excepcion)`: bloque `except`: error por excepción y traza de ejecución.
- `logging.critical(mensaje)`: error crítico, el programa no puede continuar su ejecución.
- `logging.error(mensaje)`: error, el programa no puede cumplir con su tarea correctamente.
- `logging.warning(mensaje)`: informacion de algo inesperado o problema futuro.
- `logging.info(mensaje)`: información sobre funcionamiento correcto.
- `logging.debug(mensaje)`: información detallada para depuración.

- La configuración del registro de mensajes de error se debe hacer en el bloque de selección que comprueba que se ejecuta como **módulo principal**.

```
if __name__ == "__main__":
    logging.basicConfig(filename="err.log", filemode="w") # salida "err.log", nivel WARNING
    logging.basicConfig(level=logging.DEBUG)              # salida CONSOLA, nivel DEBUG
    logging.basicConfig(filename="err.log", filemode="w", level=logging.DEBUG) # salida "err.log"
    main()
```

# Captura y tratamiento de excepciones

## Ejemplo

- Leer de teclado una lista de números. En caso de datos incorrectos (lista vacía o desviación típica mayor que 1) volver a leer la lista.

```
def leer_lista(mensaje: str) -> list[float]:
    """devuelve una lista de números reales leídos de teclado"""
    datos = input(mensaje)
    return [float(x) for x in datos.split()]                # float(x) puede lanzar excepciones

def leer_datos(mensaje: str) -> list[float]:
    """devuelve una lista de números válidos (con desv. típica entre cero y uno) leídos de teclado
    ok = False
    while not ok:
        try:
            lista = leer_lista(mensaje)                     # puede lanzar excepciones
            desv_tip = calc_desv_tipica(lista)              # puede lanzar excepciones (vacía)
        except (TypeError, ValueError, ArithmeticError) as ex: # captura la excepción
            ok = False                                      # error, volver a generar la lista
        else:                                               # si no se ha lanzado ninguna excep
            ok = (0 <= desv_tip <= 1)                       # si desv_tipica no adecuada, gene
    return lista

def main() -> None:
    datos = leer_datos("Inrtroduce numeros: ")
    print("Datos: ", datos)

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

# El bloque `finally` (I)

- El bloque **`finally`** **se ejecuta siempre** que se ejecuta el bloque **`try`** correspondiente. Su cometido es la **recuperación de recursos** y **restablecer un estado correcto** en la ejecución.

```python
import logging
from dataclasses import dataclass
@dataclass
class Frigorifico:
    capacidad: int
    contenido: list[str]
    puerta_abierta: bool = False

def abrir_puerta(frigo: Frigorifico) -> None:
    frigo.puerta_abierta = True
    logging.debug("Puerta abierta")
def cerrar_puerta(frigo: Frigorifico) -> None:
    frigo.puerta_abierta = False
    logging.debug("Puerta cerrada")

def almacenar(frigo: Frigorifico, producto: str) -> None:
    if not frigo.puerta_abierta:
        raise ValueError("frigorifico cerrado")
    if len(frigo.contenido) >= frigo.capacidad:
        raise ValueError("frigorifico lleno")
    frigo.contenido.append(producto)
    logging.debug(f"Producto [{producto}] almacenado")

def mostrar(frigo: Frigorifico) -> None:
    print("Contenido:", frigo.contenido)
    print("Puerta abierta:", frigo.puerta_abierta)
```

```python
def main() -> None:
    frigo = Frigorifico(capacidad=3,
                        contenido=[])
    try:
        abrir_puerta(frigo)
        almacenar(frigo, "yogur")
        almacenar(frigo, "helado")
        almacenar(frigo, "fruta")
        almacenar(frigo, "agua")
    except ValueError as exc:
        logging.exception(exc)
    else:
        print("productos almacenados")
    finally:
        mostrar(frigo)
        cerrar_puerta(frigo)
    mostrar(frigo)

if __name__ == "__main__":
    logging.basicConfig(level=logging.DEBUG)
    main()

# Este código SE PUEDE HACER MEJOR con
# clases, como veremos en el tema 3, y
# con gestores de contexto en el tema 5
```

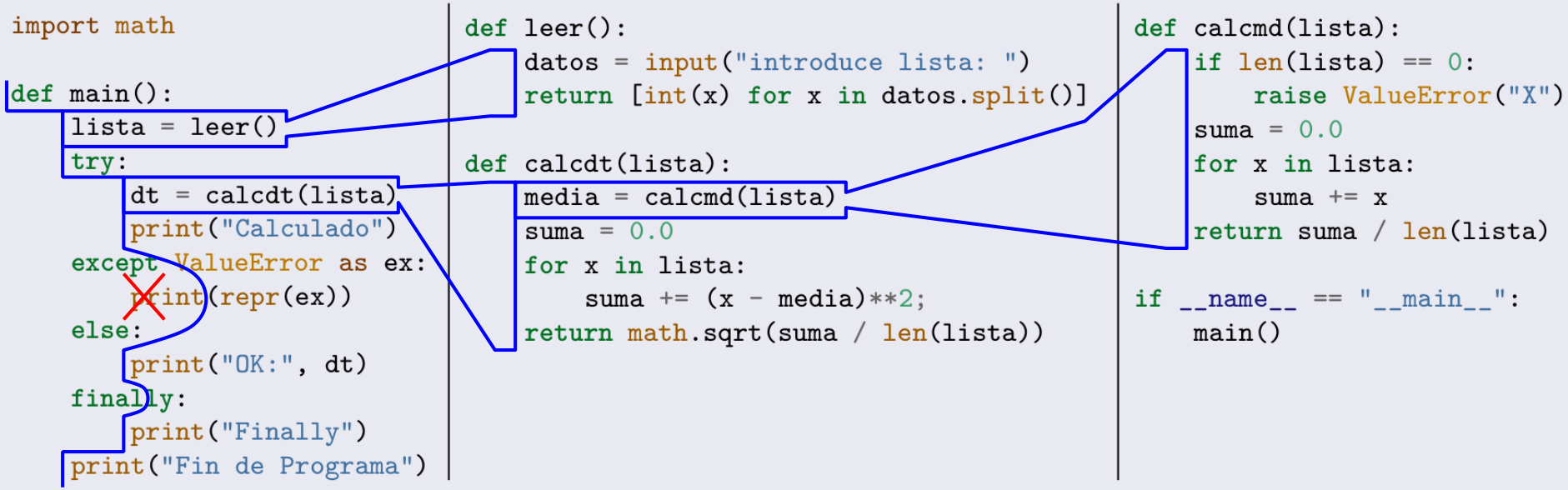
# El bloque `finally` (II)

## Resultado de la ejecución

```
DEBUG:root:Puerta abierta
DEBUG:root:Producto [yogur] almacenado
DEBUG:root:Producto [helado] almacenado
DEBUG:root:Producto [fruta] almacenado

ERROR:root:frigorifico lleno
Traceback (most recent call last):
  File "frigorifico.py", line 37, in main
    almacenar(frigo, "agua")
  File "frigorifico.py", line 21, in almacenar
    raise ValueError("frigorifico lleno")
ValueError: frigorifico lleno

Contenido: ['yogur', 'helado', 'fruta']
Puerta abierta: True

DEBUG:root:Puerta cerrada

Contenido: ['yogur', 'helado', 'fruta']
Puerta abierta: False
```

# El flujo de ejecución en presencia de excepciones (I)

## Escenario 1: Ejecución correcta (no se lanzan excepciones)




## Ejecución correcta (no se lanzan excepciones)

```
introduce lista: 1 2 3 4 5
Calculado
OK: 1.4142135623730951
Finally
Fin de Programa
```

# El flujo de ejecución en presencia de excepciones (II)

## Escenario 2: Excepción ValueError no capturada (lanzada fuera del bloque try)

```python
import math

def main():
    lista = leer()
    try:
        dt = calcdt(lista)
        print("Calculado")
    except ValueError as ex:
        print(repr(ex))
    else:
        print("OK:", dt)
    finally:
        print("Finally")
    print("Fin de Programa")
```

```python
def leer():
    datos = input("introduce lista: ")
    return [int(x) for x in datos.split()]

def calcdt(lista):
    media = calcmd(lista)
    suma = 0.0
    for x in lista:
        suma += (x - media)**2;
    return math.sqrt(suma / len(lista))
```

```python
def calcmd(lista):
    if len(lista) == 0:
        raise ValueError("X")
    suma = 0.0
    for x in lista:
        suma += x
    return suma / len(lista)

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

## Ejecución: excepción ValueError no capturada (lanzada fuera del bloque try)

```
introduce lista: a b c
Traceback (most recent call last):
  File "prueba.py", line 36, in <module>    main()
  File "prueba.py", line 4, in main         lista = leer()
  File "prueba.py", line 18, in leer        return [int(x) for x in datos.split()]
  File "prueba.py", line 18, in <listcomp>  return [int(x) for x in datos.split()]
ValueError: invalid literal for int() with base 10: 'a'
```

# El flujo de ejecución en presencia de excepciones (III)

## Escenario 3: Excepción ValueError capturada en cláusula except

```
import math

def main():
    lista = leer()
    try:
        dt = calcdt(lista)
        print("Calculado")
    except ValueError as ex:
        print(repr(ex))
    else:
        print("OK:", dt)
    finally:
        print("Finally")
    print("Fin de Programa")
```

```
def leer():
    datos = input("introduce lista: ")
    return [int(x) for x in datos.split()]

def calcdt(lista):
    media = calcmd(lista)
    suma = 0.0
    for x in lista:
        suma += (x - media)**2;
    return math.sqrt(suma / len(lista))
```

```
def calcmd(lista):
    if len(lista) == 0:
        raise ValueError("X")
    suma = 0.0
    for x in lista:
        suma += x
    return suma / len(lista)

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

## Ejecución: excepción ValueError capturada en cláusula except

```
introduce lista:
ValueError('X')
Finally
Fin de Programa
```

# El flujo de ejecución en presencia de excepciones (IV)

## Escenario 4: Excepción RuntimeError no capturada (cláusula except captura ValueError)

***Modificación:*** *ahora la sentencia* `raise` *lanza* ***RuntimeError***.

```
import math

def main():
    lista = leer()
    try:
        dt = calcdt(lista)
        print("Calculado")
    except ValueError as ex:
        print(repr(ex))
    else:
        print("OK:", dt)
    finally:
        print("Finally")
    print("Fin de Programa")
```

```
def leer():
    datos = input("introduce lista: ")
    return [int(x) for x in datos.split()]

def calcdt(lista):
    media = calcmd(lista)
    suma = 0.0
    for x in lista:
        suma += (x - media)**2;
    return math.sqrt(suma / len(lista))
```

```
def calcmd(lista):
    if len(lista) == 0:
        raise RuntimeError("X")
    suma = 0.0
    for x in lista:
        suma += x
    return suma / len(lista)

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

## Ejecución: excepción RuntimeError no capturada (cláusula except captura ValueError)

```
introduce lista:
Finally
Traceback (most recent call last):
  File "prueba.py", line 36, in <module>   main()
  File "prueba.py", line 6, in main        dt = calcdt(lista)
  File "prueba.py", line 21, in calcdt     media = calcmd(lista)
  File "prueba.py", line 29, in calcmd     raise RuntimeError("X")
RuntimeError: X
```

# Captura de una excepción y lanzamiento de otra excepción

- A veces resulta útil capturar un tipo de excepción y lanzar otro tipo de excepción. En este caso, las excepciones se encadenan automáticamente. Por ejemplo:

```
try:
    ...
    ... raise RuntimeError("esto es un error")
    ...
except RuntimeError as exc:
    raise ValueError(str(exc))  # eleva ValueError con el mismo mensaje de exc
```

# Definición de nuevas excepciones

- El programador puede **definir nuevas excepciones** como subclases de excepciones previamente definidas en el sistema (estudiaremos más sobre clases y herencia en el **Tema 3**).
  - Definir nuevas excepciones en nuestros módulos y paquetes permite que sea **más fácil gestionar y diferenciar** entre las excepciones lanzadas por el sistema, y las excepciones lanzadas por nuestros propios módulos y paquetes.
  - En nuestro caso, **crearemos** nuevas excepciones definiendo **clases**, con nombre terminado en **Error**, que hereden de la excepción **RuntimeError**, sin necesidad de definir nuevos atributos ni métodos (**pass**).
  - La nueva excepción definida puede **lanzarse** como las demás excepciones.

```
# Definición de la nueva excepción denominada NuevoError

class NuevoError(RuntimeError):  # Hereda todas las características de RuntimeError
    """Explicacion ..."""
    pass                         # No es necesario definir nada más

try:
    ...
    if situacion_erronea:
        raise NuevoError("Situación errónea")   # lanza la excepción
    ...
except NuevoError as exc:                       # captura la excepción
    print(f"Error: [{exc!r}]")
```

# Comportamiento en situaciones excepcionales

- Cuando la **comprobación** para evitar la situación excepcional es **poco costosa y** hay situaciones habituales en las que es **probable** que se produzca la excepción.
  - Entonces, se suele introducir código para **evitar** que la situación errónea excepcional pueda suceder.

```
elemento = int(input("Introduce número: "))
conjunto = {1, 2, 3, 4}
if elemento in conjunto:          # comprobación eficiente en conjunto
    conjunto.remove(elemento)     # lanza excepción si elemento no está
```

- Cuando la **comprobación** para evitar la situación excepcional es **costosa o** es **raro** encontrar situaciones donde se puede producir la excepción.
  - Entonces, **no** se suele **evitar** que la situación errónea excepcional pueda suceder, y se prepara el código para su captura y recuperación de forma adecuada.

```
elemento = int(input("Introduce número: "))
lista = [1, 2, 3, 4]
try:
    posicion = lista.index(elemento)  # lanza excepción si elemento no está
except ValueError as ex:
    posicion = -1                     # elemento no encontrado
```