Parte I: Programación en un lenguaje orientado a objetos

1. Introducción a los lenguajes de programación
2. Datos y expresiones
3. Clases
4. Estructuras algorítmicas
5. Estructuras de Datos
6. *Tratamiento de errores*

- Excepciones. Tratamiento de excepciones. Patrones de tratamiento de excepciones. Lanzar Excepciones. Documentación de las excepciones. Usar nuestras propias excepciones. Acciones de limpieza.

7. Entrada/salida
8. Herencia y polimorfismo

6.1. Excepciones

Son un mecanismo especial para ***gestionar errores***

- Permiten separar el tratamiento de errores del código normal
- Evitan que haya errores que pasen inadvertidos
- Permiten propagar de forma automática los errores desde las funciones o métodos más internos a los más externos
- Permiten agrupar en un lugar común el tratamiento de errores que ocurren en varios lugares del programa
- En Python son clases especiales

Las excepciones se ***lanzan*** para indicar que ha ocurrido un error:

- automáticamente, cuando el sistema detecta un error
- o explícitamente cuando el programador lo establezca

Están presentes en los lenguajes más modernos

Propagación de excepciones

main
invoca a
función
contiene a
bloque interno
invoca a
método
contiene a
bloque interno
invoca a
función

Objetivos de las excepciones

Los errores *nunca deben pasar inadvertidos*

Los *errores previsibles*:

- Deben ser *detectados lo antes posible*
- Deben ser *notificados* a la función o método invocante (y quizás también al usuario)
- Su efecto debe ser *corregido* por la aplicación (siempre que sea posible)

Los *errores imprevistos*

- es preferible que *finalicen* la aplicación (con un mensaje que permita su *diagnosis*),
- a que pasen inadvertidos causando un mal funcionamiento del sistema, de difícil diagnóstico

Conceptos asociados a las excepciones

Lanzar

- La excepción se lanza para avisar de que hay un error
  - automáticamente
  - o explícitamente con la instrucción `raise`

Propagar

- La excepción se propaga de un bloque al siguiente hasta que se trata

Manejador

- Instrucciones que se escriben para resolver un error

Tratar

- Ejecutar las instrucciones de un manejador de excepción
  - para resolver la situación de error: instrucciones `try-except`

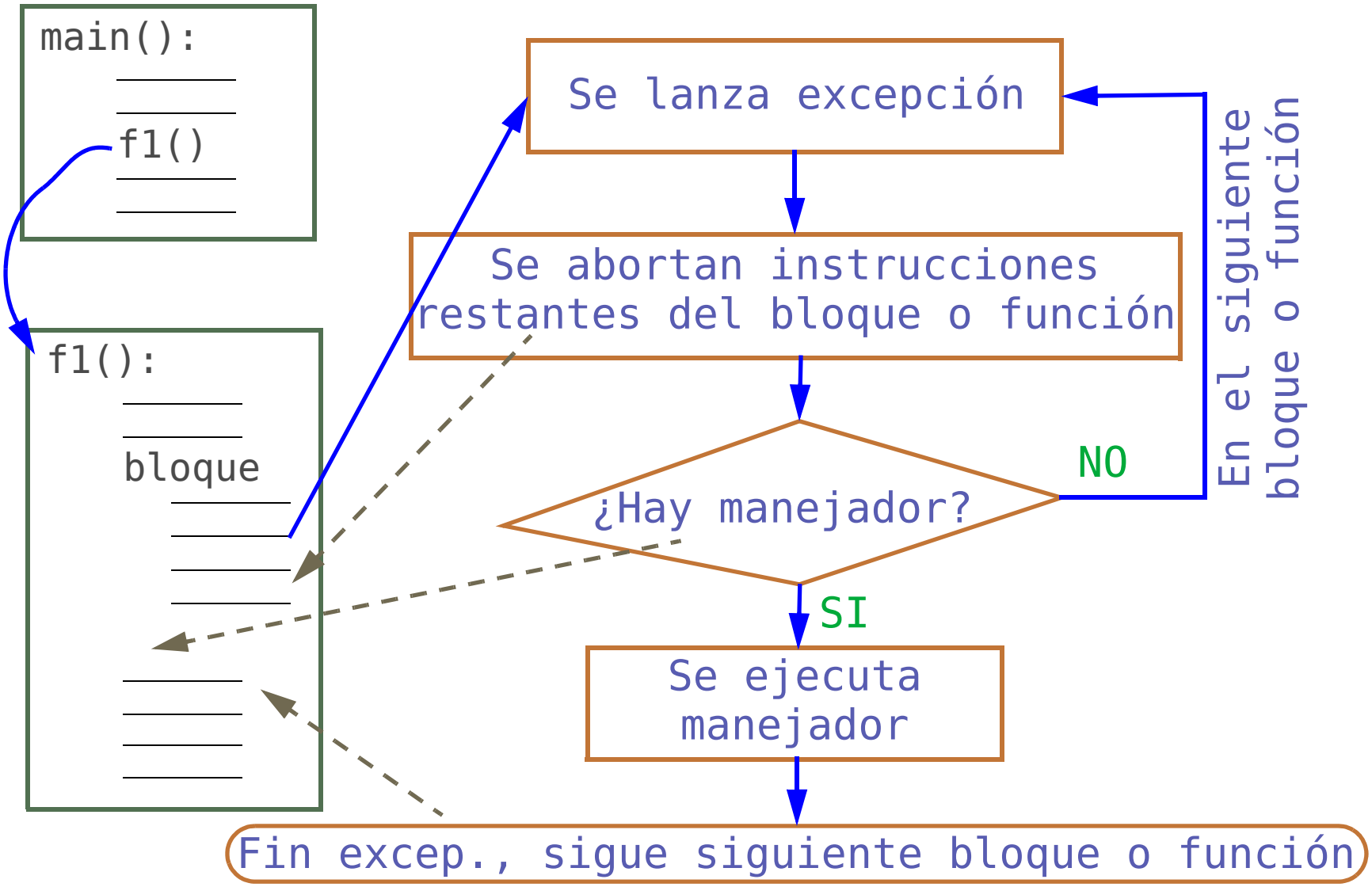
Ejemplo de lanzamiento automático: División por cero

```
def main():
    """
    Lee números del teclado y muestra su cociente
    """
    i: int = int(input("i="))
    j: int = int(input("j="))
    div: float = i/j
    print(f"Cociente de {i}/{j}= {div}")
```

cuando **j** vale 0 se lanza la excepción **ZeroDivisionError**

cuando se lanza la excepción esta línea no se ejecuta, y aparece un mensaje de error

Propagación de excepciones: en detalle



Propagación de excepciones

Una línea de código ***lanza*** una excepción

El bloque, método o función que contiene esa línea de código se aborta en ese punto

Si el bloque ***trata*** esa excepción (es decir, si tiene un ***manejador*** para ella), el manejador se ejecuta

- la "vida" de la excepción finaliza en este punto

Si no tiene manejador, la excepción se ***propaga*** al bloque, método o función superior

- que, a su vez, podrá tratar o dejar pasar la excepción

Si la excepción alcanza el bloque principal (`main`) y éste tampoco trata la excepción, el programa finaliza con un mensaje de error

Ejemplo de propagación de excepciones

```
def divide(a_0: int, b_0: int) -> int:
    print("divide: antes de dividir")
    div: int = a_0//b_0
    print("divide: después de dividir")
    return div

def intermedio():
    print("intermedio: antes de divide")
    div: int = divide(2, 0)
    print(f"intermedio: resultado: {div}")

def main():
    print("main: antes de intermedio")
    intermedio()
    print("main: después de intermedio")
```

Ejemplo de propagación de excepciones

Puesto que hay división por cero la salida generada será:

```
main: antes de intermedio
intermedio: antes de divide
divide: antes de dividir
Traceback (most recent call last):

  File "...", line 1, in <module>
    main()

  File "...", line 22, in main
    intermedio()

  File "...", line 17, in intermedio
    div: int = divide(2, 0)

  File "...", line 11, in divide
    div: int = a_0/b_0

ZeroDivisionError: division by zero
```

a continuación se lanza la excepción `ZeroDivisionError`

Se muestra la secuencia de llamadas que llevaron al error

Se muestra el nombre de la excepción

6.2. Tratamiento de excepciones

La forma general de escribir un *bloque* en el que se tratan excepciones es:

```
try:
    instrucciones
except ClaseExcepción1:
    instrucciones de tratamiento
except (ClaseExcepción2, ClaseExcepción3):
    instrucciones de tratamiento
```

Los *manejadores* se evalúan por orden:

- una excepción se trata en el primer "`except`" para esa excepción o para una de sus *superclases*

Se permiten múltiples excepciones en un "`except`"

Ejemplo: propagación con bloque `try-except`

En el ejemplo "propagación de excepciones" anterior, añadimos un bloque `try-except` al método intermedio:

```
def intermedio():
    try:
        print("intermedio: antes de divide")
        div: int = divide(2, 0)
        print(f"intermedio: resultado: {div}")
    except ZeroDivisionError:
        print("Ocurrió una división por cero")
```

Ejemplo: propagación con bloque `try-except`

La salida por consola que obtenemos ahora es:

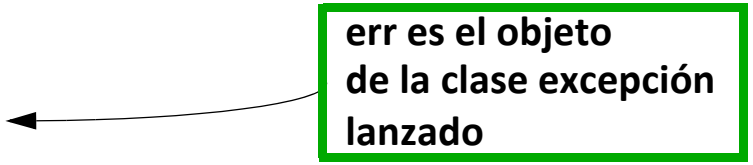
```
main: antes de intermedio
intermedio: antes de divide
divide: antes de dividir
Ocurrió una división por cero
main: después de intermedio
```

- en este caso la excepción es tratada, por lo que
  - el programa NO finaliza de forma abrupta
  - NO aparece un mensaje *del sistema* indicando que se ha producido una excepción

Mensajes de error

Se puede obtener el objeto de la clase excepción para mostrarlo como parte de un mensaje de error y para otros usos

```
try:
    ...
except ZeroDivisionError as err:
    print(err)
```



Tratamiento específico o general

Tratamiento ***únicamente*** de la excepción `ZeroDivisionError`

```
try:
    ...
except ZeroDivisionError:
    ...
```

Es posible poner un ***tratamiento común*** para cualquier excepción

```
try
    ...
except:
    ...
```

- es cómodo pero *en general está fuertemente desaconsejado*, ya que puede ocurrir un tratamiento inadecuado para una excepción no prevista

6.3. Patrones frecuentes de tratamiento de excepciones

Según la gravedad del error:

- ***leve***: se notifica el error, pero la aplicación continúa
- ***grave***: se notifica el error y se finaliza una parte de la aplicación, o la aplicación completa
- ***recuperable***: se reintenta la operación

Esquema de tratamiento de un *error leve*

```
try:
    instrucciones
except ClaseExcepción:
    notificación del error leve
```

A veces no queremos hacer ningún tratamiento, pero sí dejar de propagar la excepción:

```
try:
    instrucciones
except ClaseExcepción:
    pass # esta instrucción no hace nada
```

Es preciso diseñar bien el ámbito del `try-except`, pues las instrucciones posteriores al error se abandonarán

- Incluir dentro del `try-except` aquellas instrucciones que no tiene sentido ejecutar si el error ocurre, pero no más

Esquema de tratamiento de un *error grave*

```
import sys


try:
    instrucciones
except ClaseExcepción:
    notificación del error grave
    sys.exit(1) # finaliza la aplicación con un
                # código de error = 1
```

En otras ocasiones se finaliza solo el método o función (con `return`), o se lanza otra excepción (con `raise`)

Esquema de tratamiento de *error recuperable*

```
correcto: bool = False
while not correcto:
    try:
        instrucciones a reintentar
        correcto = True
    except ClaseExcepción:
        tratamiento
```

Si queremos limitar el número de reintentos podemos usar un contador y relanzar la misma excepción u otra si se alcanza el máximo

Ejemplo de error recuperable

```
def lee_dos_notas() -> tuple[int, int]:
    """
    Lee de teclado dos notas, reintentando
    hasta que sean correctas
    """
    correcto: bool = False
    while not correcto:
        try:
            nota1: int = int(input("Nota 1: "))
            nota2: int = int(input("Nota 2: "))
            correcto = True
        except ValueError as exc:
            print("Nota errónea: ", exc)
    # Bucle finalizado
    return nota1, nota2
```

Ejemplo de error recuperable con reintentos limitados

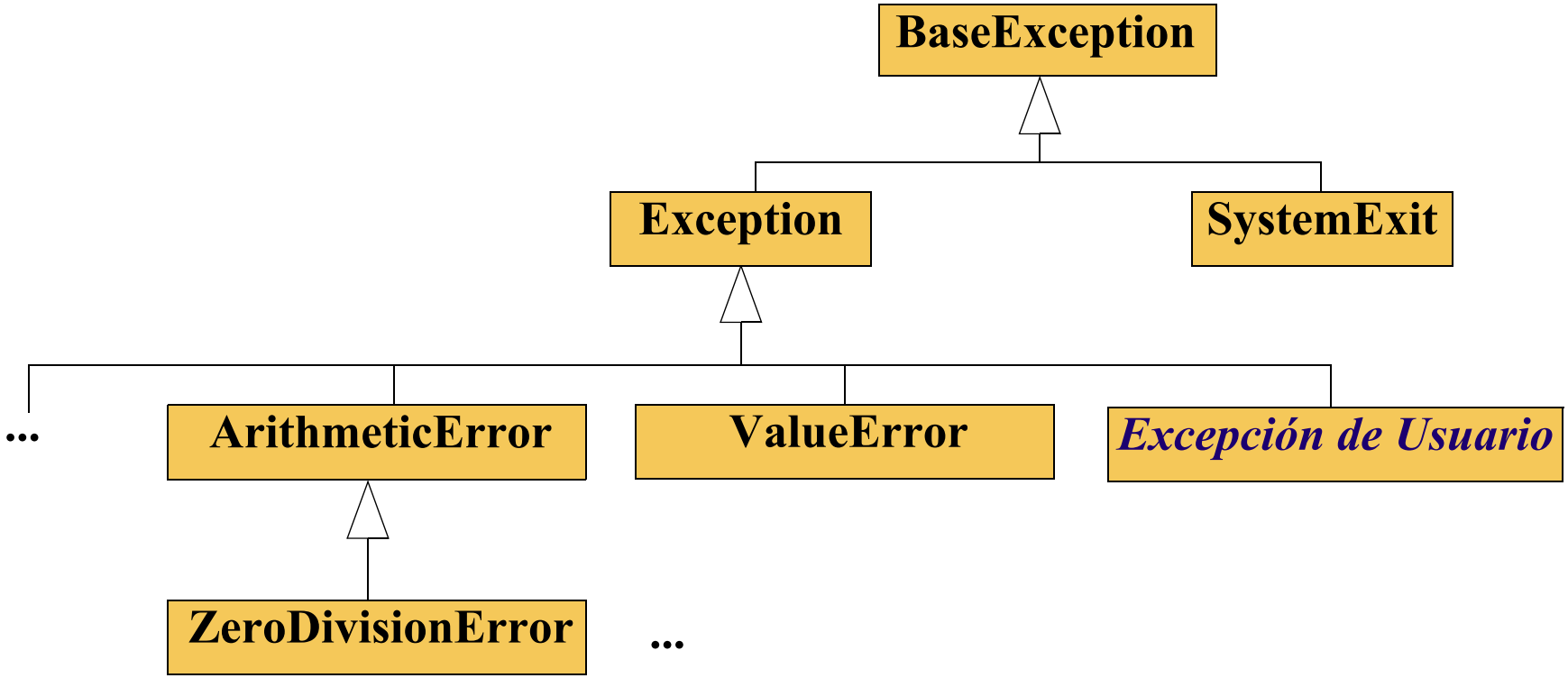
```
def lee_dos_notas() -> tuple[int, int]:
    """
    Lee de teclado dos notas, reintentando un máximo de 10
    veces hasta que sean correctas
    Returns:
        las dos notas
    Raises:
      ValueError: si se supera el máximo número de 10 reintentos
    """
```

Ejemplo de error recuperable con reintentos limitados (cont.)

```
correcto: bool = False
contador: int = 0
while not correcto and contador < 10:
    try:
        contador += 1
        nota1: int = int(input("Nota 1: "))
        nota2: int = int(input("Nota 2: "))
        correcto = True
    except ValueError as exc:
        print("Nota errónea: ", exc)
if correcto:
    return nota1, nota2
raise ValueError("Demasiados reintentos")
```

Jerarquía de las excepciones

Cada excepción pertenece a una *superclase*



https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Algunas excepciones predefinidas

`ZeroDivisionError`	División por cero
`ValueError`	Error aritmético (p.e. $\sqrt{-x}$, o valor incorrecto, p.e., float("12z")
`IndexError`	Índice de secuencia (lista, tupla, ...) fuera de límites (índice<-len o índice>=len)
`AttributeError`	El objeto no dispone del atributo solicitado
`MemoryError`	La memoria se ha agotado
`TypeError`	Se ha aplicado una función a un objeto del tipo incorrecto (p.e., al objeto None)
`RecursionError`	Se ha alcanzado el máximo número de niveles de recursión
`EOFError`	Se ha intentado leer un fichero pasado su final

https://docs.python.org/3/library/exceptions.html#concrete-exceptions

6.4. Lanzar excepciones

Se lanzan con la instrucción **raise**

```
raise ClaseExcepción
```

Habitualmente el constructor de la excepción admite parámetros

```
raise ClaseExcepción("mensaje")
```

- que sirven para dar información adicional sobre la causa de la excepción

Ejemplo:

```
if s is None:
    raise TypeError("La variable s vale None")
```

Lanzar la misma excepción

En algunas ocasiones un manejador puede volver a lanzar la misma excepción con la instrucción `raise` sola:

```
except ClaseExcepción
    parte del tratamiento de la excepción;
    raise
```

- puede ser útil cuando se desea realizar parte del tratamiento de la excepción de forma local
  - y dejar que el resto del tratamiento lo haga un bloque o función superior

6.5 Documentación de las excepciones

Cada función o método debe documentar las excepciones que propaga al exterior. No las que trata, pues estas ya no se propagan

Se documentan en una cláusula "Raises", que se pone después del "Returns". Ejemplo:

```
def duplica_primer_caracter(s: str) -> str:
    """
    Si s no es nulo retorna el primer carácter de s duplicado

    Returns:
        el primer carácter de s duplicado
    Raises:
        TypeError: cuando s vale None
    """
    if s is None:
        raise TypeError("Error: no debes poner s a None")
    return s[0]+s[0]
```

6.6. Usar nuestras propias excepciones

El programador puede crear sus propias excepciones y utilizarlas para indicar errores

- Una excepción propia es una clase que extiende a Exception

Excepción mínima, sin constructor, métodos, ni atributos:

```
class MiError(Exception):
    """
    Descripción del error
    """
```

Habitualmente se le pone la palabra `Error` al final del nombre

Ejemplo de excepción propia

Clase que define la excepción, con un atributo que es un mensaje de error

```
class NoQuieroTrabajarError(Exception):
    """Indica que un trabajador no quiere trabajar

    Attributes:
        dia: día de la semana en que ocurre el error
    """
```

Ejemplo de excepción propia (cont.)

```
    def __init__(self, dia: str):
        """
        Constructor que copia el parámetro en el
        atributo del mismo nombre

        Args:
            dia (str): El dia de la semana en que
            ocurre el error
        """
        self.dia: str = dia
        super().__init__()
```

El atributo es público, para poder verlo desde fuera

Llamada al constructor de la superclase (ver Cap. 8)

Ejemplo de excepción propia

Clase con una operación que lanza la excepción

```
class Operador:
    """
    Representa un trabajador
    """
```

Ejemplo de excepción propia (cont.)

```
    def trabaja(self, dia_semana: str) -> str:
        """
        Pedimos al trabajador que trabaje

        Args:
            dia_semana (str): día de la semana en que
                              pedimos trabajar

        Returns:
            str: Un mensaje de confirmación

        Raises:
            NoQuieroTrabajarError: Indica que el trabajador
                                   no quiere trabajar
        """
        if dia_semana == "miércoles":
            return "OK. voy a trabajar"
        # si no es miércoles
        raise NoQuieroTrabajarError(dia_semana)
```

Ejemplo de excepción propia (cont.)

Operación que invoca a `trabaja()`, y no trata la excepción

```
def manda_trabajar_viernes(trabajador: Operador):
    """
    Operación que invoca a trabaja(), y no trata la excepción.
    Por tanto, la propaga hacia el exterior.
    Esto lo indicamos en la cláusula "Raises"

    Args:
        trabajador (Operador): el objeto de la clase Operador.

    Returns:
        None.

    Raises:
        NoQuieroTrabajarError: El trabajador no quiere trabajar
    """

    print("Vamos a trabajar en viernes")
    trabajador.trabaja("viernes")
```

Ejemplo de excepción propia (cont.)

Operación que invoca a `trabaja()` y trata la excepción

```
def manda_trabajar_jueves(trabajador: Operador):
    """
    Operación que invoca a trabaja(), y trata la excepción

    No ponemos cláusula "Raises" pues la excepción se trata,
    y no se propaga hacia el exterior

    Args:
        trabajador (Operador): El objeto de la clase Operador
    """

    try:
        print("Vamos a trabajar en jueves")
        trabajador.trabaja("jueves")
    except NoQuieroTrabajarError as err:
        print("Se ha producido un error:",
              "No quiero trabajar en", err.dia)
```

Permite conocer el motivo de la excepción

6.7. Acciones de limpieza

La cláusula **`finally`** permite crear un bloque de código de "*limpieza*", que se ejecuta siempre después del bloque `try-except`

- haya habido excepción o no,
- incluso si se sale a causa de `return`, `break` o `continue`
- si hay excepción se ejecuta el `finally`; además, si no ha sido tratada se relanza la misma excepción

Ejemplo

```
try:
    instrucciones
finally:
    print("Esto se hace sí o sí")
```

Habitualmente se usa para liberar recursos (memoria, ficheros, ...) que se hayan reservado en el bloque de instrucciones

La cláusula `else`

Permite crear un bloque de código que se ejecuta después del bloque `try-except` (pero antes del `finally`) si *no* ha habido excepción

Ejemplo

```
try:
    instrucciones
except ZeroDivisionError:
    print("Ocurrió una división por cero")
else:
    print("Todo fue bien")
finally:
    print("Esto se hace sí o sí")
```

Es raro necesitar la cláusula `else`, pues su código se puede poner dentro de las instrucciones del `try`