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

Notas:

Una parte importante de un programa de calidad es que sepa gestionar los errores que pueden ocurrir.

Hay muchos errores que pueden ocurrir, como por ejemplo:

- Teclear mal un dato numérico y poner letras o algo que no sea un número.
- Escribir mal el nombre de un archivo o directorio.
- Realizar un cálculo aritmético incorrecto, por ejemplo una división por cero.
- Intentar invocar un método de un objeto inexistente, que vale `None`.

Habitualmente, si no hacemos nada, al ocurrir un error el programa se detiene con un mensaje de error.

- Si hemos estado trabajando un rato con el programa, por ejemplo metiendo datos, perderemos nuestro trabajo.
- Esto es inaceptable en un programa profesional.

Las excepciones constituyen un mecanismo avanzado para gestionar errores. La gestión tiene dos roles:

- *lanzar* la excepción: para notificar que ha ocurrido un error.
- *tratar* la excepción: ejecutar instrucciones de manejo, que resuelven la situación de error.

La parte de lanzar el error puede ser:

- automática, cuando el sistema detecta el error,
- o explícita, cuando es el propio programador el que escribe código para detectar el error.

Propagación de excepciones

main

invoca a

función

contiene a

bloque interno

invoca a

método

contiene a

bloque interno

invoca a

función

Notas:

Cuando ocurre un error y se lanza una excepción, el sistema busca un tratamiento para esa excepción, donde se resolverá la situación del error.

- Esta "búsqueda del tratamiento" se llama *propagación* y es automática (no hay que hacer nada).

Cuando una función (o método) invoca a otra decimos que la segunda función está en un nivel más interno.

La figura trata de ilustrar este proceso de propagación:

- 1. En un método, función o bloque interno ocurre un error y se notifica lanzando una excepción.
  - Esto lo representamos en la figura como una burbuja que se crea.
- 2. La excepción se propaga al método, función o bloque más externo, en busca de un tratamiento.
  - En la figura esto sería el ascenso de la burbuja por el líquido.
  - Si no se encuentra tratamiento, la propagación sigue: la burbuja sigue ascendiendo.
- 3. Cuando en un método, función o bloque superior se encuentra un tratamiento de excepción, ésta se resuelve ejecutando las instrucciones de tratamiento.
  - En la figura esto se representa por la burbuja que se detiene al alcanzar la superficie del líquido.
  - Eventualmente el error se resuelve: la burbuja desaparece.

Pero, ¿Qué es un *bloque* en este contexto?

- Es un conjunto de instrucciones con tratamiento de excepción, situadas entre las instrucciones `try` y `except`. Lo veremos más adelante.

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

Notas:

El mecanismo de propagación automática nos asegura que un error no pase inadvertido.

- Hay lenguajes, como C, que no tienen excepciones: en ellos es posible que un error pase inadvertido; al final el programa funcionará mal, pero no sabremos por qué.
- Afortunadamente, Python tiene excepciones y los errores son notificados.

Algunos errores son previsibles y podremos tratarlos escribiendo instrucciones en un manejador de excepción.

Otros errores son imprevistos y no los tratamos. Preferimos que el programa se detenga con un mensaje de error que nos dice dónde ocurrió el error.

- Tras su diagnóstico, corregiremos el programa para que no vuelvan a ocurrir.

Tras una batería completa de pruebas, el programa debería estar libre de estos errores imprevistos.

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`

Notas:

Es muy importante no confundir los dos roles que juegan las excepciones:

- *Lanzar*: para notificar el error.
  - Algunas excepciones se lanzan automáticamente, sin que tengamos que hacer nada.
  - Otras se lanzan por el programador con la instrucción `raise`.
- *Tratar*: para resolver el error.
  - Con las instrucciones `try` y `except`.

Para no equivocarse al interpretar un enunciado, es muy importante dominar los conceptos descritos arriba.

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

Notas:

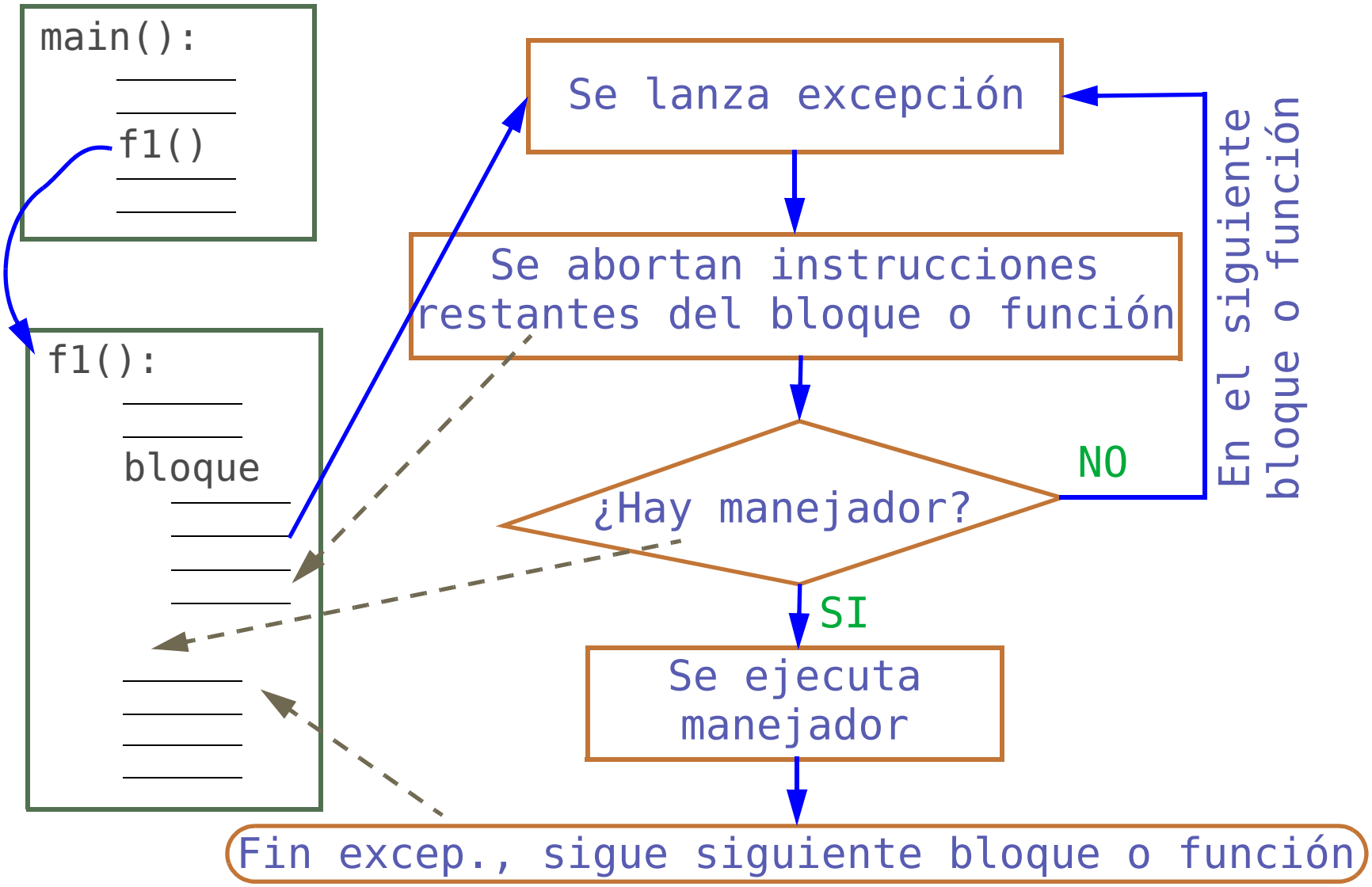
En este ejemplo se ilustra el lanzamiento automático de una excepción llamada `ZeroDivisionError`, que ocurre al dividir un número entre cero.

- El error ocurre si para la variable `j` se teclea el valor `0`.

Al lanzarse la excepción comienza el proceso de propagación, en el que:

- Se abandona la función o bloque actual:
  - En este caso, no se ejecuta el `print()`.
- Se busca un manejador de excepción: en este caso no hay ninguno:
  - Por tanto, el programa se detiene con 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

Notas:

Las dos transparencias anteriores muestran el proceso de propagación de una excepción.

Antecedentes:

1. El programa comienza a ejecutar instrucciones del `main()`.
2. El main() invoca a la función `f1()` y el computador comienza a ejecutar sus instrucciones.
3. `f1()` entra en un bloque (con instrucciones encerradas entre `try` y `except`) y sigue ejecutando sus instrucciones.
4. Antes de acabar el bloque ocurre un error y se lanza una excepción para notificarlo. Aquí comienza el proceso de propagación.

Propagación:

5. Se abortan las instrucciones restantes del bloque.
6. Se mira si en ese bloque hay un manejador para esa excepción.
7. Caso A: Si hay manejador se ejecutan sus instrucciones y la "vida" de esta excepción acaba.
   - Se continuaría ejecutando por el siguiente bloque o función, como si nada hubiese pasado.
8. Caso B: Si no hay manejador se lanza la misma excepción en el siguiente bloque o función.
   - La propagación continúa hasta que se encuentra un manejador o, si se llega al final del `main()`, se abandona el programa.

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

Notas:

En el ejemplo que aparece en las dos transparencias anteriores se muestra la propagación de una excepción a través de tres funciones. El proceso es automático y tiene estos pasos:

Antes de la excepción:

1. El programa comienza ejecutando las instrucciones del `main()`.
2. El `main()` pone un mensaje en pantalla y luego invoca la función `intermedio()`.
3. La función `intermedio()` pone un mensaje en pantalla y luego invoca a la función `divide()`.
4. La función `divide()` pone un mensaje en pantalla y luego intenta hacer una división. Sin embargo, el denominador es cero, por lo que se lanza `ZeroDivisionError` automáticamente.

Propagación:

5. Se abandonan las instrucciones restantes de la función `divide()`.
6. Como `divide()` no tiene manejador para la excepción, se retorna a la función superior, que es `intermedio()`. Allí se lanza la misma excepción: `ZeroDivisionError`.
7. Se abandonan las instrucciones restantes de la función `intermedio()`: es decir, las posteriores a la invocación de `divide()`.
8. Como `intermedio()` no tiene manejador para la excepción, se retorna a la función superior, que es el `main()`. Allí se lanza la misma excepción: `ZeroDivisionError`.
9. Se abandonan las restantes instrucciones del `main()`: es decir, el último `print()`. Como no hay manejador, el programa se detiene con un mensaje de error.

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`"

Notas:

El tratamiento de una excepción consiste en ejecutar instrucciones para resolver o paliar el error.

Estas instrucciones se escriben en un *manejador*.

Las instrucciones que pueden fallar se encierran, sangradas, en un bloque `try` - `except`, como vemos en la transparencia anterior.

Cada manejador contiene instrucciones Python normales escritas tras la instrucción `except`

- el ámbito del manejador se define mediante un nivel de sangrado.

Puede haber varios manejadores, para diversas excepciones.

Un único manejador puede servir a varias excepciones: en ese caso, se expresan los nombres de las excepciones en una tupla, separados por comas.

Las excepciones tienen una jerarquía, que veremos más adelante.

- Esto quiere decir que algunas excepciones son descendientes o "*subclases*" de otras.
  - El antecesor de una excepción se llama su "*superclase*".
- Cuando ponemos un manejador para una excepción, el tratamiento incluye a esa excepción o cualquiera de sus descendientes.
- Por ello, el orden en que escribimos los manejadores es importante: se comienza a buscar el manejador en el orden en que aparecen.

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

Notas:

El ejemplo que aparece en las dos transparencias anteriores es similar al de la página 15. Se muestra la propagación de una excepción a través de tres funciones, pero en este caso hay un manejador:

Antes de la excepción:

1. El programa comienza ejecutando las instrucciones del `main()`.
2. El `main()` pone un mensaje en pantalla y luego invoca a la función `intermedio()`.
3. La función `intermedio()` pone un mensaje en pantalla y luego invoca a la función `divide()`.
4. La función `divide()` pone un mensaje en pantalla y luego intenta hacer una división. Sin embargo, el denominador es cero, por lo que se lanza `ZeroDivisionError` automáticamente.

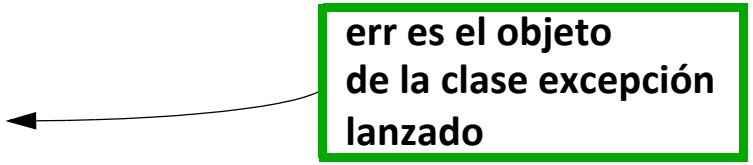
Propagación:

5. Se abandonan las instrucciones restantes de la función `divide()`.
6. Como `divide()` no tiene manejador para la excepción, se retorna a la función superior, que es `intermedio()`. Allí se lanza la misma excepción: `ZeroDivisionError`.
7. Se abandonan las instrucciones restantes del bloque contenido en la función `intermedio()`: es decir, las posteriores a la invocación de `divide()`.
8. Como `intermedio()` sí tiene manejador para la excepción, se ejecutan sus instrucciones (en este caso se pone un mensaje de error en pantalla) y finaliza la "vida" de la excepción.
9. Se retorna al `main()` y se ejecutan con normalidad las instrucciones posteriores a la invocación de `intermedio()`: es decir, el último `print()`. Finalmente el programa acaba normalmente.

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)
```



Notas:

Las excepciones son objetos de una clase especial que representa una excepción.

- Por ejemplo, `ZeroDivisionError` es una clase.
- Cuando hay una división por cero el sistema automáticamente crea un objeto de esta clase y lo "lanza" para notificar el error.

El objeto se puede obtener con la notación indicada arriba, y se puede usar, por ejemplo, para mostrar su nombre en pantalla.

- En este caso hemos llamado `err` al objeto excepción.

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

Notas:

Recordamos que para los errores no previstos en general no debemos escribir manejadores.

- Es preferible que el programa se detenga con un mensaje de error.
- Este hecho debemos detectarlo en una prueba exhaustiva de nuestro programa, y corregir el programa para que no vuelva a ocurrir.

Por ello, un tratamiento general que sirva para cualquier excepción está muy desaconsejado.

- Con la única excepción de usarlo para poner un mensaje de "últimas voluntades" antes de "morir". Por ejemplo, este mensaje podría indicar con quién hay que contactar para cambiar el programa y que así no vuelva a ocurrir.

En general debemos poner tratamientos para excepciones específicas.

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

Notas:

Hay muchas formas de tratar excepciones.

Pero hay algunos patrones que se repiten con frecuencia, y que veremos a continuación.

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

Notas:

En el caso de un error leve, que no afecta al funcionamiento normal del programa, es frecuente:

a. Notificar al usuario de la ocurrencia del error con un mensaje en pantalla, o

b. ignorar el error, poniendo un tratamiento de excepción que no hace nada (instrucción `pass`), pero detiene la propagación de la excepción.

La notificación del error, si se hace, debe ser suficientemente explicativa. Mensajes como "Error" o "Ha ocurrido un error" son poco útiles.

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`)

Notas:

En el caso de un error grave, que afecta al funcionamiento normal del programa, es frecuente:

a. Finalizar el programa, usualmente con una notificación al usuario de la ocurrencia del error con un mensaje en pantalla, o

b. finalizar una parte del programa, por ejemplo una función o método completo.

- A veces se pone un mensaje de error y se lanza la misma excepción u otra, para propagarla al nivel superior (instrucción `raise`).

La notificación del error, si se hace, debe ser suficientemente explicativa.

Para finalizar el programa podemos hacerlo con `sys.exit(1)`.

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

Notas:

Este tercer patrón de tratamiento de excepciones es el más complejo.

Se usa cuando la situación de error puede corregirse reintentando la operación.

- Por ejemplo, si estamos leyendo un dato de teclado y el usuario se equivoca, podemos pedirle que vuelva a escribir el dato.

No usar este patrón para errores que no se puedan corregir reintentando la operación.

- Por ejemplo, si una estructura de datos de tamaño limitado está llena (y no hay nadie vaciándola) no tiene sentido seguir reintentando meter en ella un dato nuevo.

Como puede observarse, en este patrón usamos un bucle para repetir la operación muchas veces.

- El bloque `try-except` se pone dentro del bucle.
- Si la operación va bien ponemos la variable `correcto` a `True`, para finalizar el bucle.
- Si hay un error y se lanza una excepción, las instrucciones restantes del `try-except` se abandonan, por lo que la variable `correcto` mantiene su valor `False` y el bucle continúa tras ejecutarse el manejador.

Si queremos un número limitado de reintentos, podríamos usar un bucle `for` o un bucle `while` con un contador.

- Hay un ejemplo de esto más adelante.

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
```

Notas:

En el ejemplo se muestra una función que lee dos notas enteras del teclado, reintentando la operación si el usuario se equivoca escribiendo algo que no sea un número.

Agrupamos la lectura de las dos notas en un solo bloque `try-except`.

- Esto hace que si fallamos al leer una nota, ya sea la primera o la segunda, se reintenta la lectura de ambas.
- Un diseño alternativo hubiera sido leer primero una nota, reintentando hasta que fuese correcta, y luego hacer lo mismo con la segunda.
  - Esto hubiese implicado una estructura más compleja con dos bucles de reintento, cada uno con un bloque `try-except` en su interior.
  - O, mejor, hacer una segunda función para leer un solo número entero.

En este ejemplo la excepción salta al realizar al conversión a entero con `int()`. Si el string leído del teclado es incorrecto (por ejemplo, tiene letras) se lanza `ValueError`.

En el tratamiento de la excepción ponemos un mensaje de error para que el usuario sepa que se ha equivocado.

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")
```

Notas:

En el ejemplo se muestra la misma función del ejemplo de la página 35, que lee dos notas enteras del teclado reintentando la operación si el usuario se equivoca.

- Sin embargo, en esta ocasión limitamos el número de intentos a un máximo de 10.
- Como puede que nunca leamos valores correctos, tras 10 intentos la función lanza `ValueError`, con un mensaje explicativo: "Demasiados Reintentos". Esto se hace con la instrucción `raise`, que veremos con detalle más adelante.

Podemos escoger dos diseños

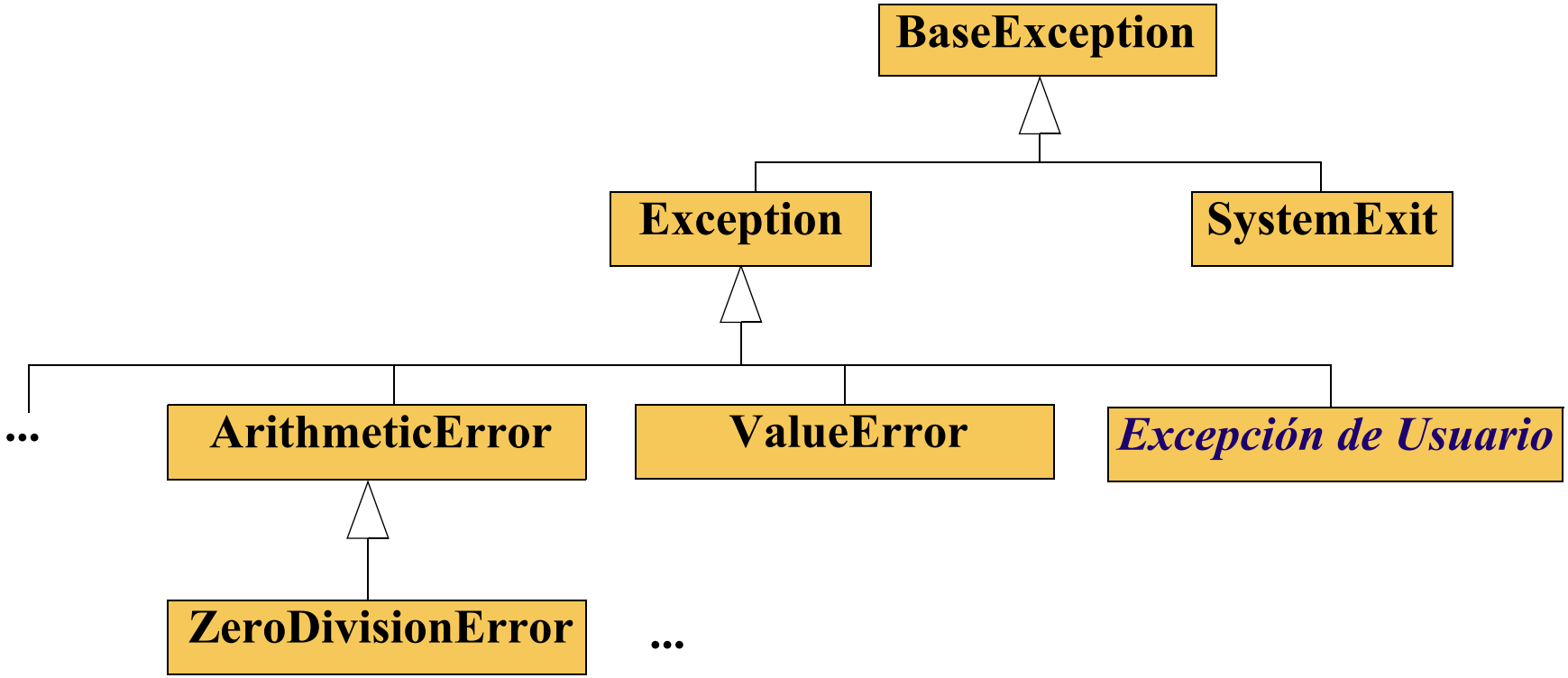
- Bucle `while`, con dos condiciones de permanencia: que aún no hayamos leído los valores correctos y que aún no hayamos alcanzado el límite de 10 intentos.
  - Esta es la opción elegida.
  - Necesitamos un contador de intentos: variable que inicializamos a cero y aumentamos a cada intento.
- Bucle `for` que se hace un máximo de 10 veces. En su interior, si la lectura es correcta abandonamos el bucle prematuramente con `break` o `return`.

Al finalizar el bucle es necesario distinguir las dos situaciones que pueden haber ocurrido:

- Los números se leyeron correctamente: los retornamos como una tupla de dos valores.
- Se alcanzó el máximo número de reintentos sin ninguna lectura correcta: se lanza `ValueError` para avisar del error a la función superior. No podemos ignorar alegremente esta situación, pues no hemos cumplido el objetivo y no tenemos las notas leídas.

Jerarquía de las excepciones

Cada excepción pertenece a una *superclase*

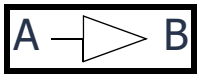


https://docs.python.org/3/library/exceptions.html#exception-hierarchy

Notas:

En la figura se muestra la jerarquía de algunas excepciones comunes.

Como hay muchas más excepciones, se indica una dirección de internet (URL) donde se amplía el esquema a todas las excepciones.

En los diagramas de clases, la flecha hueca 

indica que A es una *subclase* de B. O inversamente, que B es una *superclase* de A.

- Podemos interpretar que A es hija de B.
- También decimos que A es una *extensión* de B.

Por ejemplo, vemos que `ZeroDivisionError` es hija de `ArithmeticError`, que a su vez es hija de `Exception`.

`Exception` contiene las excepciones normales, que representan situaciones de error.

`SystemExit` es una excepción especial que se usa para abandonar el programa.

Más adelante veremos que podremos crear nuestras propias excepciones, y lo haremos haciéndolas hijas de `Exception`.

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")
```

Notas:

Hemos visto que muchas excepciones se lanzan automáticamente.

Sin embargo, nosotros podemos también lanzarlas desde nuestro programa.

- Lo haremos si detectamos un error y queremos aprovechar el mecanismo de propagación para avisar del error a otra parte del programa.

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]
```

Notas:

La documentación de las excepciones permite saber qué puede ocurrir al invocar una función o método.

Cuando invocamos a una función desde otra, esta documentación nos ayudará en la decisión de si tratar esas excepciones o dejarlas pasar propagándolas a un nivel superior.

La documentación se pone en una cláusula `"Raises"` donde se documentan una por una las excepciones que se pueden lanzar o propagar, junto a las causas por las que ocurren.

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

Notas:

Es habitual que identifiquemos situaciones de error especiales, relacionadas con nuestra aplicación.

- En ese caso podemos crear nuestras propias excepciones, con nombres que expresen la situación de error.
- Por ejemplo, en una aplicación de biblioteca: `LibroNoEncontradoError`.

La excepción es una clase que extiende a la clase `Exception`. Esto se manifiesta poniendo la clase `Exception` entre paréntesis, tras el nombre de la clase.

Basta crear la clase con su encabezamiento y el comentario de documentación que explique el error.

- No es preciso añadir constructor ni métodos, aunque si fuese conveniente podríamos hacerlo.
- El constructor podría añadir atributos que contuviesen información asociada al error
- y los métodos podrían trabajar con esa información.
- A continuación veremos un ejemplo con un constructor que crea un atributo.

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)

Notas:

En este ejemplo creamos una excepción propia para ilustrar su uso. Lo hacemos extendiendo la clase `Exception`.

- Hemos optado por añadirle un atributo, que es el día de la semana en que ocurre el error.
- El atributo se crea en el constructor, como siempre.
- Hemos usado un atributo público, para poder usarlo desde fuera, con la notación `objeto.atributo`
  - Por ello su nombre no comienza por "__".
- Al final del constructor hemos llamado al constructor de la superclase. En este caso la superclase es `Exception`. Nos referimos a ella con `super()`.
  - Esto es habitual al extender clases.
  - Lo veremos en el capítulo 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)
```

Notas:

El ejemplo continúa. Aquí se muestra el lanzamiento de la excepción desde otra clase.

- El método `trabaja()` lanza la excepción, usando la instrucción `raise`.
- Como el constructor de la excepción `NoQuieroTrabajarError` requiere un string que indica el día de la semana en que ocurre el error, al lanzar la excepción aportamos el parámetro `dia_semana`.
- Puede observarse que este trabajador solo quiere trabajar los miércoles. En otros días de la semana lanza la excepción para notificar que no quiere trabajar.
- Observar también la documentación de la excepción lanzada, en la cláusula `"Raises"`

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")
```

Notas:

El ejemplo continúa con una función que invoca al método `trabaja()`, pero no trata la excepción.

Como se pide al trabajador trabajar en viernes, éste lanzará la excepción, pero como aquí no se trata, será propagada al bloque superior.

Observar que como la excepción se propaga, la incluimos en la cláusula `"Raises"`

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

Notas:

Finalizamos aquí el ejemplo. En este caso escribimos otra función que invoca al método `trabaja()`, pero a diferencia de la anterior trata la excepción con un bloque `try` - `except`.

En el tratamiento de la excepción ponemos un mensaje de error en pantalla.

- Como parte del mensaje de error incluimos el atributo `dia` del objeto excepción, que se llama `err`.
  - El atributo, por tanto, es `err.dia`
  - Esto permite una mejor explicación de la causa del error.

Observar que no ponemos cláusula `"Raises"` pues la excepción se trata dentro del método, y no se propaga hacia el exterior

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

Notas:

Un bloque `try` puede tener una cláusula `finally`, que sirve para ejecutar unas instrucciones al finalizar el bloque, tanto si ha habido error o no.

No hay que abusar de esta cláusula.

- A veces se usa incorrectamente para indicar que queremos hacer unas instrucciones después del bloque `try`.
- Pero esto se puede hacer simplemente poniendo las otras instrucciones después del bloque, fuera:

```
try:
      instrucciones
except MiError:
      print("Notificación de error")
otras_instrucciones
```

- Las "`otras_instrucciones`" se hacen después del bloque `try-except`; no es necesario el `finally`.
- Entonces ¿para qué es útil el `finally`?
  - Para el caso en que queremos que las "`otras_instrucciones`" se ejecuten incluso en caso de un error no manejado, en este caso un error diferente a `MiError`.
  - Esto es útil para acciones imperativas de limpieza, por ejemplo liberar memoria o recursos que de no liberarse quedarían bloqueados para siempre.

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`

Notas:

La instrucción que se muestra arriba se puede reescribir así:

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

El funcionamiento es casi el mismo, excepto que en el primer caso (con `else`) las instrucciones de la cláusula `else` no quedarían amparadas por el manejador de excepción.