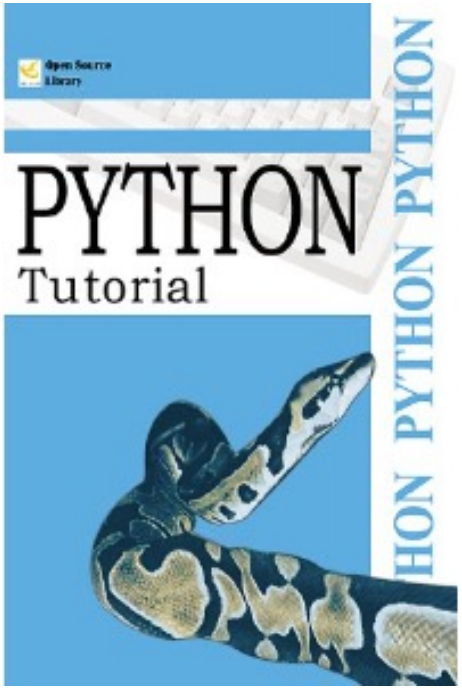
Exceptions



Dra. Mª Dolores Rodríguez Moreno

Objectives

Specific Objectives

- Understanding how exceptions are generated
- Understanding how to generate exceptions
- Understanding how to intercept and handle exceptions

Source

- https://docs.python.org/3/reference/
- https://ellibrodepython.com/
- Python Tutorial - Tapa blanda. GuidoVan Rossum (2012)

Outline

- Introduction
- Definition
- Raising exceptions
- Handling exceptions
- else block
- finally block
- assert
- Custom exceptions
- Execution Flow
- Context Managers

Introduction

- An error is an unexpected condition that halts program execution
- It refers to any issue that prevents a computer from functioning properly
- Errors can occur in both software and hardware
- There are classified into:

1. Syntax Errors
2. Semantic Errors
3. Run Time Errors
4. Logical Errors

Syntax Errors

- Involve the violation of the formal rules that define how valid statements are constructed in a programming language
- They occur when the grammatical rules of the language are not followed

Example:

```
def myfunction
   if (a == 3
      print(“Equal to 3)
```

- Can you see any error in the code?

Semantic Errors (I)

- Semantic errors occur when a statement is logically incorrect, even if it follows the syntax rules
- Semantics are the rules that give meaning to code, ensuring it makes sense in context
- Example: *The car drives the driver*
  - While syntactically correct, this statement is illogical
  - It has no clear meaning
- Example: *A*B + C = D*
  - *Correct??*

Semantic Errors (I)

- Semantic errors occur when a statement is logically incorrect, even if it follows the syntax rules
- Semantics are the rules that give meaning to code, ensuring it makes sense in context
- Example: *The car drives the driver*
  - While syntactically correct, this statement is illogical
  - It has no clear meaning
- Example: *A*B + C = D*
  - *It provides a semantical error as an expression cannot be on the left*

Runtime Errors

- They occur during the program's execution, often caused by invalid operations
- Examples
  - Attempting to open a non-existent or corrupted file
  - Dividing a number by zero

Logical Errors

- They happen when a program runs without crashing but produces incorrect or unintended results
- What is my intention in this example?

Example:

```
total = 0

for i in range(5):

    total = total + 10

print(total)
```

Outline

- Introduction
- **Definition**
- Raising exceptions
- Handling exceptions
- else block
- finally block
- assert
- Custom exceptions

Definition

- **Exceptions** are events that disrupt the normal flow of a program
- Importance:
  - Handling exceptions is crucial to prevent programs from crashing
  - Without handling errors, the program may stop unexpectedly, which is unacceptable in many applications (e.g., aircraft, trains, or ATMs)
- Common Scenarios:
  - Dividing by zero
  - Accessing an invalid index in a list
  - Working with files that don't exist

Definition (I)

- Consequence of an abnormal or special situation that may occur during the ***execution*** of a program
- Python incorporates native support to handle these types of situations
- Exception Handling in Python:
  - Avoid using return codes to signal errors, reducing the need to evaluate such values with if or similar structures, simplifying your code
  - It offers a clean way to separate error-handling code from the main flow of the application, improving its readability and maintainability
  - Allows functions that call other functions to not need to check return values: if a function terminates without throwing exceptions, the caller can assume that no problem occurred

Examples:

```
23 * (1/0)

      ZeroDivisionError: division by zero

5 + nodefinida*7

      NameError: name 'nodefinida' is not defined

'2' + 2

      TypeError: can only concatenate str (not "int") to str
```

Outline

- Introduction
- Definition
- **Raising exceptions**
- Handling exceptions
- else block
- finally block
- assert
- Custom exceptions
- Context Managers

Raising exceptions

- Using the keyword *raise* in Python
- The *raise* statement allows you to trigger exceptions in Python
- This is useful when you want to enforce specific conditions or signal errors in your code

Syntax:

```
raise [ExceptionType [args [traceback]]]
```

Raise parameters

1. ExceptionType (Required)
   - Specifies the exception class to be raised
   - Can be a built-in exception (e.g., ValueError, TypeError) or a user-defined custom exception
   - If not specified, the last captured exception is re-raised
2. args (Optional)
   - A list or tuple of arguments passed to the exception constructor
   - Useful for providing additional details about the error
- 3. traceback (Optional)
   - Allows specifying a traceback object containing information about the call stack at the time the exception occurred
   - Useful for reusing or customizing stack traces
   - Less common but powerful for advanced use cases like custom debugging or error logging

Syntax Examples

Raising Built-in Exceptions

```
raise ZeroDivisionError("Custom exception message")

raise NameError("Custom exception message")

#The string provided is displayed along the exception
```

Raising Without Parameters

```
raise ZeroDivisionError
```

Ways Exceptions are Raised (I)

1. Automatically by Python: when an invalid operation occurs (e.g., division by zero)

Example

```
result = 10 / 0  # Raises ZeroDivisionError automatically
```

Ways Exceptions are Raised (II)

2. Manually using *raise*: to enforce custom error handling

Example

```
if condition_is_invalid:

    raise ValueError("Condition is invalid!")
```

Ways Exceptions are Raised (III)

3. Custom Exceptions: define and raise your own exceptions (using your own class)

Example

```
# Creamos una excepción personalizada

class MiExcepcionPersonalizada(Exception):

    pass

#Y ya podríamos lanzarla con raise cuando queramos

raise MiExcepcionPersonalizada()
```

Outline

- Introduction
- Definition
- Raising exceptions
- **Handling exceptions**
- else block
- finally block
- assert
- Custom exceptions
- Context Managers

Handling exceptions

- By default, the interpreter handles exceptions by stopping the program and printing an error message
- However, we can override this behavior by *catching* the exception

Syntax:

```
try:

    #Something that produce exception

except SomeSpecificException:

    #Something to inform/handle the exception
```

Example: (no handle)

```
fin = open('a_file')

for line in fin:

    print line

fin.close()
```

Example: handle

```
try:
    fin = open('bad_file')
    for line in fin:
        print line
    fin.close()
except:
    print 'Something went wrong.'
```

Steps

- First, the *try* clause (the statement(s) between the *try* and *except* keywords) is executed
- If no exception occurs, the *except* clause is skipped, and execution of the *try* statement is finished
- If an exception occurs during execution of the *try* clause, the rest of the clause is skipped
- If the exception matches the *except* clause, the *except* clause is executed
- If no handler is found, execution stops with an error message
- Better to specify the type of exception

Exceptions in Python

- IndexError: Raised when trying to access an index that is out of the range
- KeyError: Raised when trying to access a dictionary key that does not exist in the dictionary
- SyntaxError: Raised when the parser encounters a syntax error in the code
- IOError: Raised when a file operation fails
- ImportError: Raised when a module cannot be imported
- ValueError: Raised when a function receives the correct type but inappropriate value
- KeyboardInterrupt: Raised when the user interrupts program execution
- EOFError: Raised when an end-of-file condition occurs without reading any data
- More exceptions…. Next slide shows some examples…

Example I: type of exceptions

```
my_list = [1, 2, 3]

print(my_list[10])  # Raises IndexError because index 10 does not exist in the list



my_dict = {'a': 1, 'b': 2}

print(my_dict['c'])  # Raises KeyError because 'c' is not a key in the dictionary



if True

    print("Missing colon in the if statement")  # Raises SyntaxError
```

Example II: type of exceptions

```
data = input("Enter something (or hit Ctrl+D to simulate EOF): ")
# Raises EOFError: End of file reached without reading any data

with open("non_existent_file.txt", "r") as file:

        data = file.read() # Raises IOError: The file could not be opened or read

import non_existent_module  # Raises ImportError

number = int("not_a_number") # Raises ValueError

print("Running... Press Ctrl+C to interrupt.") #Raises KeyboardInterrupt
```

Catching several exceptions

Example I: type of exceptions

```
try:

   c = 5/0     # Si descomentas esto entra en ZeroDivisionError

   d = 2 + "Hola" # Si descomentas esto entra en TypeError

except ZeroDivisionError:

   print("No se puede dividir entre cero!")

except TypeError:

   print("Problema de tipos!")
```

Catching several exceptions in same block

- Both exceptions are handled in the same block.
- Useful when multiple exceptions require similar handling

Example:

```
try:

    #c = 5/0        # Si comentas esto entra en TypeError

    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError

except (ZeroDivisionError, TypeError):

    print("Excepcion ZeroDivisionError/TypeError")
```

Catching all Exceptions

- If you don't know the exception, you can used "Exception"
- It is the base class for all exceptions
- Handles any error but is less specific

Example:

```
try:
    #c = 5/0        # Si comentas esto entra en TypeError

    d = 2 + "Hola" # Si comentas esto entra en ZeroDivisionError

except Exception:

    print("Ha habido una excepción")
```

Knowing the type of Exceptions

- Python provides the exact exception type:

```
type (the_exception_variable)
```

- Accessing Exception Details & capturing Exception Information
- Use ***as*** to assign the exception object to a variable
- You can also use the *type()* attributes
- Benefits: provides detailed error information

Example of *type*

Example:

```
try:

    d = 2 + "Hola"

except Exception as ex:

    print("Ha habido una excepción del tipo", type(ex))

    print("Exception Args:", ex.args)  # A tuple: ('division by zero',)

# Ha habido una excepción del tipo <class 'TypeError'>
```

Outline

- Introduction
- Definition
- Raising exceptions
- Handling exceptions
- **else block**
- finally block
- assert
- Custom exceptions
- Context Managers

else block

- A special block that executes only if no exception occurs in the try block
- Runs only if no exception occurs
- Useful for separating "normal flow" code from error-handling code
- Placed after the exception block

Example:

```
try:

    # Forzamos una excepción al dividir entre 0

    x = 2/0

except:

    print(”Visualiza: Entra en except, ha ocurrido una excepción")

else:

    print(”Visualiza: Entra en else, no ha ocurrido ninguna excepción")

#Visualiza: ???????
```

Example:

```
try:

    # Forzamos una excepción al dividir entre 0

    x = 2/1

except:

    print(" Visualiza: Entra en except, ha ocurrido una excepción")

else:

    print(" Visualiza: Entra en else, no ha ocurrido ninguna excepción")

# Visualiza: ?????
```

Outline

- Introduction
- Definition
- Raising exceptions
- Handling exceptions
- else block
- **finally block**
- assert
- Custom exceptions
- Context Managers

finally block

- Executes no matter what happens (exception or not)

Example:

```
try:
    file = open("example.txt", "r")
except FileNotFoundError:
    print("File not found")
finally:
    print("Cleanup or close resources")
```

else and *finally*

- The else block executes only if the try block does not raise an exception
- Why Use It?
- To keep the *try* block clean by separating error-free execution from error-handling.
- Relation to finally:
- The *else* block runs before the *finally* block (if present) [if no exceptions occur]

Exercise

Cree una calculadora básica con manejo de Excepciones que:

- Solicite al usuario dos números
- Solicite al usuario una operación matemática: suma, resta, multiplicación o division
- Utilice los bloques try, else, finally y maneje excepciones como:
  - División por cero (ZeroDivisionError)
  - Introducción de datos inválidos (ValueError) si la operación no se encuentra (+-x/)
  - Capture detalles de la excepción y los muestre usando type() y sus atributos

```
def calculadora():
    # Solicitar dos números al usuario
    num1 = float(input("Introduce el primer número: "))
    num2 = float(input("Introduce el segundo número: "))
    # Solicitar la operación
    operacion = input("Introduce la operación (+, -, *, /): ")
    # Realizar la operación
    if operacion == "+":
        resultado = num1 + num2
    elif operacion == "-":
        resultado = num1 - num2
    elif operacion == "*":
        resultado = num1 * num2
    elif operacion == "/":
        resultado = num1 / num2
    else:
        print("Operación no válida")
        return
    print(f"El resultado es: {resultado}")
    print("Gracias por usar la calculadora")
```

Outline

- Introduction
- Definition
- Raising exceptions
- Handling exceptions
- else block
- finally block
- **assert**
- Custom exceptions
- Execution Flow
- Context Managers

assert

- A debugging aid used to check conditions during execution
- If the condition is False, an *AssertionError* is raised

Syntax:

```
assert condition
```

Adding info to *assert*

Example:

```
assert 1 == 2  # Raises AssertionError

#Equivalent to

if not condition: # In this case condition is → (1==2)

    raise AssertionError()

#Output       AssertionError:
```

Adding info to *assert*

- We can provide Additional Information:

Example:

```
assert False, "Assertion failed”

#Output:   AssertionError: Assertion failed

x = "ElLibroDePython"

assert x == "ElLibroDePython"

#output:??
```

Be careful with ()

Example:

```
assert False, "Assertion failed”

#Output:   AssertionError: Assertion failed
```

Example: with ( )

```
# INCORRECT: Evaluates to True the tuple

assert(False, "Assertion failed") # Evaluates to True
```

Outline

- Introduction
- Definition
- Raising exceptions
- Handling exceptions
- else block
- finally block
- assert
- **Custom exceptions**
- Execution Flow
- Context Managers

Customize exceptions

- Why Define Custom Exceptions?
- Built-in exceptions may not cover all use cases
- Custom exceptions provide:
  - Specific error identification
  - Custom error messages
  - Improved code clarity

Customize exceptions

Syntax:

```
class MyCustomException(Exception):

    pass



raise MyCustomException()
```

Customize exceptions

- We can provide parameters to the exception

Example:

```
class MiExcepcionPersonalizada(Exception):

    def __init__(self, parametro1, parametro2):

        self.parametro1 = parametro1

        self.parametro2 = parametro2
```

Example:

```
try:
    raise MiExcepcionPersonalizada("ValorPar1", "ValorPar2")
except MiExcepcionPersonalizada as ex:
    p1, p2 = ex.args
    print(type(ex))
    print("parametro1 =", p1)
    print("parametro2 =", p2)
#<class '__main__.MiExcepcionPersonalizada'>
#parametro1 = ValorPar1
#parametro2 = ValorPar2
```

Customize exceptions with parameteres

- We can pass parameters as a dictionary

Example: Use a Dictionary

```
class MiExcepcion(Exception):
    pass

try:
    raise MiExcepcion({"mensaje":"Mi Mensaje", "informacion":"Mi Informacion"})

except MiExcepcion as e:

    detalles = e.args[0]

    print(detalles)

    print(detalles["mensaje"])

    print(detalles["informacion"])

#{'mensaje': 'Este es el mensaje', 'informacion': 'Esto es la informacion'}

# Mi Mensaje

# Mi Informacion
```

Other way to do it: accessing with attibutes

```
class MiExcepcion(Exception):

    def __init__(self, mensaje, informacion):

        self.mensaje = mensaje

        self.informacion = informacion

try:

    raise MiExcepcion("Mi Mensaje", "Mi Informacion")

except MiExcepcion as e:

    print(e.mensaje)

    print(e.informacion)
```

Comparison

Dictionary Parameters	Attribute Parameters
Access with []	Access with .
Easy to add/remove keys	Requires updating class
Flexible structure	Strict definition

Exercise

- Vamos a desarrollar un sistema que calcula la nota promedio de un estudiante a partir de una lista de calificaciones
- Para asegurarte de que las notas ingresadas son válidas (números entre 0 y 10), debes crear una excepción personalizada que se dispare cuando alguna calificación sea inválida
- Cree una excepción personalizada llamada *InvalidGradeError* que debe:
  - Recibir como argumento la calificación no válida
  - Proporcionar un mensaje que explique el error
- Implementar una función *calcula_promedio(lista_calificaciones)* que:
    - Valide que todas las calificaciones estén entre 0 y 10
    - Lance la excepción *InvalidGradeError* si encuentra una nota inválida
    - Devuelva el promedio si todas las calificaciones son válidas
- Manejar la excepción con un bloque try-except: e.g. Entrada: [8, 9, 10] [8, 9, 15]

Outline

- Introduction
- Definition
- Raising exceptions
- Handling exceptions
- else block
- finally block
- assert
- Custom exceptions
- **Execution Flow**
- Context Managers

Execution Flow (I)

- Cuando se lanza una excepción:
  - Se crea un objeto de la clase de excepción. Se pueden pasar argumentos a través del init de la clase
  - Se interrumpe el flujo de ejecución
  - Se retorna por la pila de llamadas hasta encontrar una función/método que sepa tratar la excepción
  - Se ejecuta el manejador, si lo hay
  - Sólo se ejecuta el manejador correspondiente a la excepción, el resto se ignoran

Example: class Exception y Prueba (See Google Collab)

```
class MiExcepcion(Exception):

    def __init__(self, desc):

        self.descripcion = desc

    def what(self):

        return self.descripcion

class Prueba:

    def __init__(self, nombre):

        self.MiNombre = nombre

        self.Mensaje(f"Creando Prueba:")

    def Mensaje(self, msg):

        print(f"{msg} {self.MiNombre}")
```

Example: FuncionB y C

```
def FuncionC(i):

    p = Prueba("C")

    print("Entrando a FuncionC")

    print("Se lanza la excepción.")

    raise MiExcepcion("Error Función C.")

    print("Saliendo de FuncionC")  # Nunca se ejecutará

def FuncionB(i):

    p = Prueba("B")

    print("Entrando a FuncionB")

    FuncionC(i + 1)

    print("Saliendo de FuncionB")  # Nunca se ejecutará si hay excepción
```

Example: FuncionA

```
def FuncionA(i):

    p = Prueba("A")

    print("Entrando a FuncionA")

    FuncionB(i + 1)

    print("Saliendo de FuncionA")  # Nunca se ejecutará si hay excepción
```

Example: main

```
if __name__ == "__main__":

    print("Entrando a main")

    try:

        p = Prueba("M")

        FuncionA(1)

    except MiExcepcion as e:

        print(f"Capturada una excepción: {type(e).__name__}")

        print(f"Descripción: {e.what()}")

    print("Saliendo de main")
```

Example: output

```
Entrando a main

Creando Prueba: M

Creando Prueba: A

Entrando a FuncionA

Creando Prueba: B

Entrando a FuncionB

Creando Prueba: C

Entrando a FuncionC

Se lanza la excepción.

Capturada una excepción: MiExcepcion

Descripción: Error Función C.

Saliendo de main
```

Execution Flow (II)

- Una excepción se considera manejada desde el momento en que se entra en su manejador, por lo tanto:
  - De la lista de manejadores, sólo se ejecutará el correspondiente a la excepción lanzada
    - Si en la pila de llamadas quedan otras funciones que puedan capturarla, no serán tenidas en cuenta.
  - Cualquier otra excepción lanzada desde su propio manejador deberá ser capturada por alguna otra función cuya llamada se encuentre en el camino de vuelta por la pila

Cuando utilizar excepciones

- Es un buen estilo de programación utilizarlas en módulos
  - En este caso, estamos obligados a evitar las situaciones anómalas que se puedan producir cuando el código de la misma sea ejecutado por cualquier programa.

- No utilizarlas en casos obvios; por ejemplo:
  - Podemos utilizar la excepción *IndexError* para verificar si el índice está dentro de los límites, pero es óptimo utilizar una sentencia if para prevenir que esto no suceda

Example: BAD use of exceptions

```
try:
   print(my_list[5])
except IndexError:
   print("Índice fuera de rango")
# Solución a utilizar
if 5 < len(my_list):
   print(my_list[5])
else:
   print("Índice fuera de rango")
```

Outline

- Introduction
- Definition
- Raising exceptions
- Handling exceptions
- else block
- finally block
- assert
- Custom exceptions
- Execution Flow
- **Context Managers**

Context Managers

- Allow managing resources efficiently
- Automatically execute tasks when entering and exiting a ***with*** block
- Useful for operations like opening files, database connections, or acquiring locks

Example: Context Manager

```
with open('file.txt', 'w') as file:

    file.write('Hello!')

# The file is automatically closed after the block
```

Example: NO Context Manager

```
file = open('file.txt', 'w')

try:

    file.write('Hello!')

finally:

    file.close()
```