# Errors and exception handling

Julia Clemente

*Máster en Tecnologías de la Información Geográfica*
Universidad de Alcalá

## Objectives

1. To be aware of the error handling problem.
2. Understand exceptions.
3. Handle, create and raise exceptions in Python.

# Table of contents

# Types of errors

## Types of errors (I)

- Errors happen during the development of a program.
- We need a mechanism to handle errors.
- Types of errors:
  - **Syntax errors**: detected by the interpreter (or by the compiler) when processing the source code.
    - Usually, they are the result of mistakes in spelling, punctuation, or indentation.
    - In Python Sintax indicated by message: `SyntaxError`.
    - Example: write `Clas` insteaf of `Class`.
  - **Semantic errors**: despite error message are not generated, the program in execution not produce the expect result.
    - Example: use of an incorrect algorithm.
    - Using IDE's built-in debugger!
  - **Runtime errors**: only detected during the execution.
    - Runtime error are also called **exceptions**.
    - Example: An function on execution tries to access the fourth position of list of three elements.

# Types of errors

## Example

```
>>> while True print('Hello world')
  File "<stdin>", line 1
    while True print('Hello world')
                    ^
  SyntaxError: invalid syntax
```

# What is an exception?

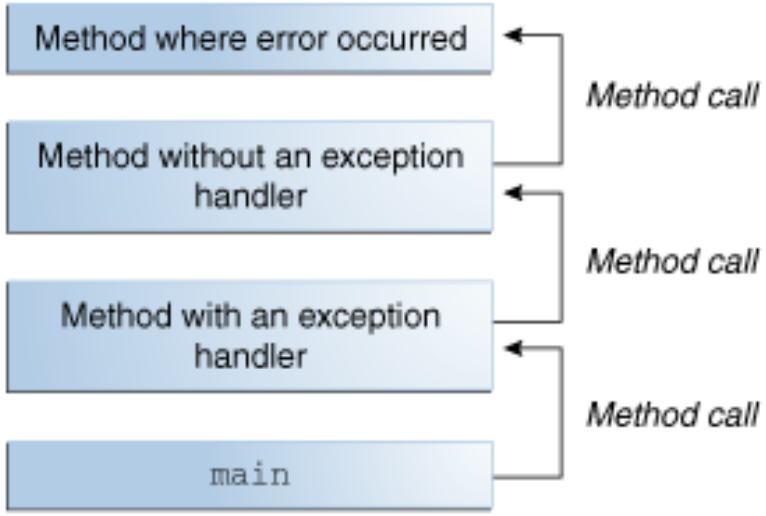
## Definition

- Exception: An error that disrupts the normal execution flow.
  - Code cannot be executed.
  - File not found, division by zero, invalid argument, etc.
  - Exception errors can come in two forms:
    - Exceptions due to errors in the code → the programmer needs to repair them.
    - Exceptions due to outside factors (e.g., user input) → the programmer should write code to anticipate and handle the error.
- **Elegant solution to handle errors**.
  - They are objects (whatever it means). See exception hierarchy at the end of this link: `https://docs.python.org/3/library/exceptions.html`

# What is an exception?

## Exception propagation (I)

Call stack:



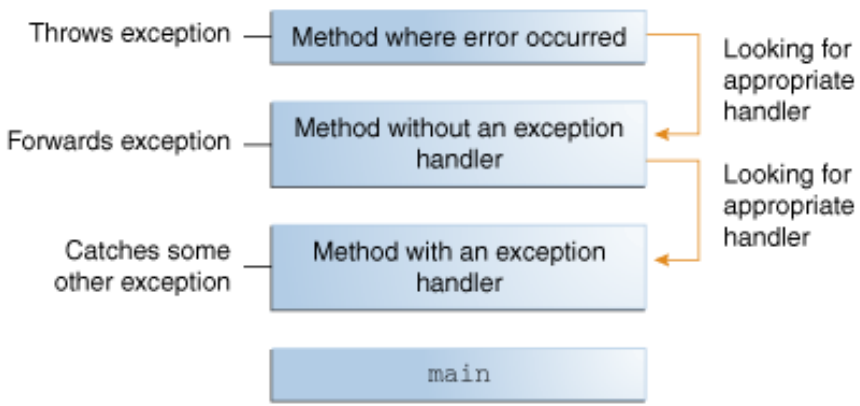

*Call stack*: Sequence of invoked methods

Source: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

# What is an exception?

## Exception propagation (II)

Exception handling:




When an error happens ...

1. Code execution is stopped.
2. An exception is thrown.
3. The interpreter goes back in the call stack.
4. When the interpreter finds an exception handler, it is executed.

The exception handler catches the exception. Otherwise, the program finishes.

Source: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

# What is an exception?

## Examples

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

`ZeroDivisionError`, `NameError` and `TypeError` are Python *built-in* identifiers. They are not reserved keywords.

# Handling exceptions

## try/except statement

Handling an exception requires a try-except statement.

- `try`: Encloses the vulnerable code.
- `except`: Allows the exception can be caught and handled.
  - What to do in case of trouble.
  - A more intelligible message to print or it can perform some alternative action.

**try-except statement**

```
try:
    # Risky code
except ExceptionType1:
    # Handle error
except ExceptionType2:
    # Handle error
except:
    # Handle errors
```

# Handling exceptions

## try/except statement: named exceptions

### Named exceptions example

```
try:
    x = int(input("Please enter a number: "))
except ValueError:
    print("Oop!, that was not a number!")
except KeyboardInterrupt:
    print("Got Ctrl-C, good bye!")
```

The name of the exception behind `except` to handle specific errors:

See `https://docs.python.org/3/library/exceptions.html`

Or creating simple code examples that cause an specific exception in Python IDLE to identify its correct name.

# Handling exceptions

## try/except statement: multiple except blocks

### Multiple except example

```
try:
    f = open('file.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("I/O error: {0}".format(err))
except ValueError:
    print("Could not convert data to integer")
except:
    print("Unexpected exception")
    raise
```

New Python element

- `raise`

# Handling exceptions

## try/except statement: else statement

Similar to a conditional statement.

### Multiple except example

```
while True:
    try:
        x = input("Enter first number: ")
        y = input("Enter second number: ")
        print(x/y)
    except:
        print("Please, try again.")
    else:
        break
```

# Handling exceptions

## Handling geoprocessing exceptions

If a geoprocessing tool fails to run:

- An `ExecuteError` exception is thrown.
- `ExecuteError` is generated **by Arcpy**.

### Example handling geoprocessing exceptions

```
import arcpy
arcpy.env.workspace = "C:/DataGIS"
infeatures = "world.shp"
outfeatures = "world.shp"
try:
    arcpy.CopyFeatures_management(infeatures, outfeatures)
except arcpy.ExecuteError:
    print arcpy.GetMessages(2)
except:
    print "There, a nontool error has ocurred."
```

# Exceptions with arguments

## Other use of raise (I)

raise statement: raise exceptions yourself, generic or not:

### Generic exception: examples

```
>> raise Exception
Runtime error
Exception
>> raise Exception(Ïnvalid workspace")
Runtime error
Exception: Invalid workspace
```

### Propagating exceptions

```
1 def divide(dividend, divider):
2     try:
3         result = dividend / divider
4         return result
5     except ZeroDivisionError:
6         raise ZeroDivisionError("The divider cannot be zero!")
```

# Exceptions with arguments

## Other use of raise (II)

### Propagating exceptions

```
def read_int():

    num_attempts = 0
    while num_attemps < 6:
        number_in = raw_input("Enter a integer number: ")
        try:
            number_in = int(number_in)
            return number_in
        except ValueError:
            num_attemps +=1
    raise ValueError, "Incorrect value entered"
```

# Exceptions with arguments

## Accessing to exception context

### *Exception arguments*: When we need more info

```
try:
    raise Exception("spam", "eggs")
except Exception as inst:
    print(type(inst))
    print(inst.args)
    print(inst)

    x, y = inst.args
    print('x =', x)
    print('y =', y)
```

```
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

And, also useful: exc_info function (sys module):
https://docs.python.org/es/3.10/library/sys.html
https://pythonprogramming.net/headless-error-handling-intermediate-python-tutorial/

# Clean-up actions

## Use of `try-except-finally` statement (I)

Sometimes we need to execute code under all circumstances:

- Typically *clean-up* actions: close files, database connections, sockets, etc.
- The **finally** clause solves this problem.

**Example 1**

```
try:
    raise KeyboardInterrupt
finally:
    print("Goodbye, world!")
```

# Clean-up actions

## Use of `try-except-finally` statement (II)

### Example 2

```
try:
    file1 = open("mifile.txt")
    # process file1
except IOError:
    print "Input/Output error."
except:
    # process the exception
finally:
    # if file1 is not closed, it should be closed
    if not(file1.closed):
        file1.close()
```

# Error handling with glitchs

## Use of traceback module

Catching exceptions can hide an error that should be corrected.

- Traceback messages are suppressed.
- The Python built-in `traceback` module can force traceback messages to print.
- The trackeback `print_exc()` method prints the most recent exception.

### Example 1

```
import sys, traceback
try:
    number_input = float(sys.argv[1])
    square = number_input ** 2
    print "The squared number is {0}".format()
except:
    print "Input must be number."
    traceback.print_exc() # what if this line (and import traceback) are removed?)
print "Bye"
```

# Error handling with glitchs

## Use of traceback module (II)

### Example 2: Find out what that does . . .

```
try:
    import arcpy
    import sys
    import traceback
    < code lines including geoprocessing tools >
except:
    error = sys.exc_info()[2]
    errorInfo = traceback.format_tb(error)[0]
    pyErrors = "PYTHON ERRORS: \nTraceback info:\n" + \
               errorInfo + "\nError Info:\n" + str(sys.exc_type) + \
               ":" + str(sys.exc_value) + "\n"

    arcpy.AddError(pyErrors)
    arcpyErrors = "ArcPy ERRORS:\n" + arcpy.GetMessages(2) + "\n"
    arcpy.AddError(arcpyErrors)
    print pyErrors + "\n"
    print arcpyErrors

print "Bye"
```

https://pro.arcgis.com/es/pro-app/arcpy/get-started/error-handling-with-python.htm

# Bibliographic references I


- [van Rosum, 2021] G. van Rossum, Jr. Fred L. Drake.
  *Python Tutorial Release 3.9.7, chapter 8*.
  Python Software Foundation, 2021.
  https://docs.python.org/3/download.html
- [Lutz, 2013] M. Lutz.
  *Learning Python*.
  O'Reilly, 2013.
- [Pilgrim, 2005] M. Pilgrim.
  *Dive into Python*.
  Ed. Apress, 2005.