

# More Python for Videogames

Videogames Technology  
Asignatura transversal

Departamento de Automática

## Objectives

1. Being able to manipulate files in Python.
2. Understand and apply Python serialization (pickles and JSON).
3. Being able to handle exceptions.

## Bibliography

- The Python Tutorial. Section 7.2: Reading and writing files. ([Link](#))
- The Python Tutorial. Chapter 8: Errors and Exceptions. ([Link](#))

# Table of Contents

## 1. Reading and writing files

- Path
- Opening files
- Reading files
- Writing files
- Useful methods
- With

## 2. Serialization

- The `pickle` module
- The `pickle` module: Examples
- The JSON module
- The JSON module: JSON format
- The JSON module: Examples

## 3. Exceptions

- Motivation
- Definition
- Handling exceptions

# Reading and writing files

## Path

### Path

A string that identifies a file in a file system

On Windows, the path is denoted by:

`path = 'C:\Windows\Temp'`

On Linux, the path is denoted by:

`path = '/tmp/prueba.txt'`

And it is represented in Python by:

`path = 'C:\\Windows\\Temp'`

But by also using raw string:

`path = r'C:\\Windows\\Temp'`

# Reading and writing files

## Opening files

All file operations are made through a file object

- First of all: Call the `open()` function

### `open()`

```
open(filename[, mode])
```

Return: An object file

- `filename`: Path
- `mode`: Characters describing how the file will be used
  - `r`: Read mode, `w`: Write mode (overwrites file)
  - `r+`: r/w mode, no truncation; `w+`: r/w mode, truncation
  - `a`: Write, appending mode
  - `b`: Binary mode, text mode by default

Always, always, always close the file: `f.close()` (unless in a `with` clause)

# Reading files (I)

## The `read()` function

`f.read([size])`

Return: the specified number of bytes

- size: The number of bytes to be read from the file. Default reads the whole file

### Option 1: Read the entire file (`f.read()`)

```
>>> f = open("/tmp/file", 'r')
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
>>> f.close()
```

## Reading files (II)

**Option 2:** Read a single line (`f.readline()`)

```
>>> f = open("/tmp/file2", 'r')
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'This is the second line of the file\n'
>>> f.readline()
 ''
>>> f.close()
```

# Reading files (III)

## Option 3: Read lines as list(f.readlines())

```
>>> f = open("/tmp/file2", 'r')
>>> f.readlines()
['This is the first line of the file.\n',
 'This is the second line of the file\n']
>>> f.close()
```

## Option 4: Read in a loop

```
f = open("/tmp/file2", 'r')
for line in f:
    print(line, end='')
f.close()
```

# Example

Calculating the average of characters per line of file `example.txt`

```
1 f = open('example.txt', 'r')
2 num_total_char = 0
3 count_line = 0
4
5 for line in f:
6     count_line += 1
7     num_total_char += len(line)
8
9 file_ex.close()
10 print('average', float(num_total_char) / float(count_line))
```

# Writing files (I)

## The `write()` function

`f.write(string)`

Return: Number of written bytes

- `string`: String to write

### Example 1: Write a line

```
>>> f = open("/tmp/file", 'w+')
>>> f.write('This is a test\n')
15
>>> f.read()
''
>>> f.close()
```

# Writing files (II)

## Example 2: Write a number

```
>>> f = open("/tmp/file", 'w+')
>>> f.write(str(42))
2
>>> f.close()
```

# Useful methods

METHOD	DESCRIPTION
f.tell()	Returns the pointer's position
f.seek(n)	Moves the pointer n bytes
f.close()	Closes a file. Use it always!

```
>>> f = open("/tmp/file", 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)
5
>>> f.read(1)
b'5'
>>> f.close()
```

# With (I)

## The with clause

```
with open(path, mode) as file:  
    ...
```

It simplifies file operations

- No need to close files
- Better exception handling

```
f = open('file')  
print(f.read())  
f.close()
```



```
with open('file') as file:  
    print(file.read())
```

# With (II)

## Hello, world

```
1 with open("file.txt", "w") as file:  
2     file.write("Hello, world.\n")  
3     file.write("This is another file.\n")
```

# With (III)

## Reading a line each time

```
1 count_line = 0
2 with open('nombres.txt') as arch_names:
3     for line in arch_names:
4         count_line += 1
5         print(f'{count_line}: {line.rstrip()}')
```



names.txt

```
1 Juan
2 Laura
3 Pablo
4 Enrique
5 Javier
```

Output

```
1: Juan
2: Laura
3: Pablo
4: Enrique
5: Javier
```

# Serialization

## Introduction

What happens if we need to store complex data structures?

- Think about lists, dictionaries or even objects ...

What happens if we need to transmit complex data structures?

### Serialization

Converting a data object into a sequence of bytes

### Deserialization

Converting a sequence of bytes into a data object

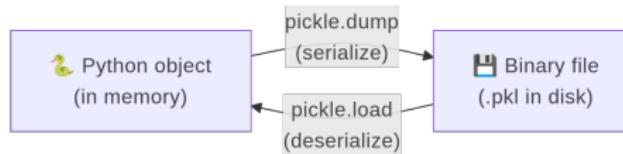
We can easily store and even transmit sequences of bytes ...

- ... and also reconstruct our original data

There are several serialization technologies: Pickles, JSON, XML, YAML, ...

# Serialization

## The pickle module



Given an object `x` and a file object `f` ...

- `pickle.dump(x, f)`: Serializes object `x` and writes it to file `f`
- `pickle.load(f)`: Reads and deserializes object from file `f`
- `x` may be a dictionary, list or even an object

Pickle uses a binary format

# Serialization

## The pickle module: Examples

### Save a list to a file

```
1 import pickle  
2  
3 numbers = [2, 5, 7, 8]  
4  
5 f = open('list.pkl', 'wb')  
6  
7 pickle.dump(numbers, f)  
8  
9 f.close()
```

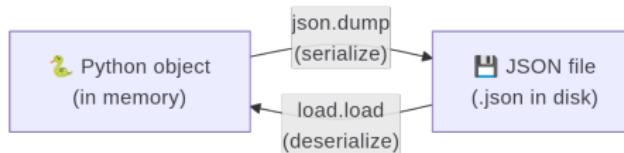
### Load a list from a file

```
1 import pickle  
2  
3 f = open('list.pkl', 'rb')  
4  
5 my_numbers = pickle.load(f)  
6  
7 print(my_numbers)  
8  
9 f.close()
```



# Serialization

## The JSON module



Given an object `x` and a file object `f` ...

- `json.dump(x, f)`: Serializes object `x` and writes it to file `f`
- `json.load(f)`: Reads and deserializes object from file `f`
- `x` may be a dictionary but **not an object**

JSON is a text format

# Serialization

## The JSON module: JSON format

### JSON: JavaScript Object Notation

- Data format for hierarchical data
- Created in 2001 for stateless client-server communication
- Text-based
- Interoperable (pickles only for Python)
- Complex data structures

filename.json

```
{  
    "firstName": "John",  
    "isAlive": true,  
    "age": 27,  
    "address": {  
        "streetAddress": "21 2nd Street",  
        "city": "New York",  
        "state": "NY",  
    },  
    "phoneNumbers": [ "111", "333" ]  
}
```

# Serialization

## The JSON module: Examples

### Save a list to a file

```
1 import json
2
3 mylist = [ "John" , 42 , "Smith" ]
4
5 myfile = open( "myfile.json" , "w" )
6
7 json.dump( mylist , myfile , indent = 4 )
```

### Load a list from a file

```
1 import json
2
3 mylist = json.load(open( 'myfile.json' ))
4
5 print( mylist )
```



# Exceptions

## Motivation

Errors happen

- We need a mechanism to handle errors
- Some errors happen before execution (syntax errors)

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

```

SyntaxError: invalid syntax

- Others are only detected in execution (runtime errors)

```
>>> int("hola")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    ValueError: invalid literal for int() with base 10: 'hola'
```

⇒ We need tools to handle errors: Exceptions

# Exceptions

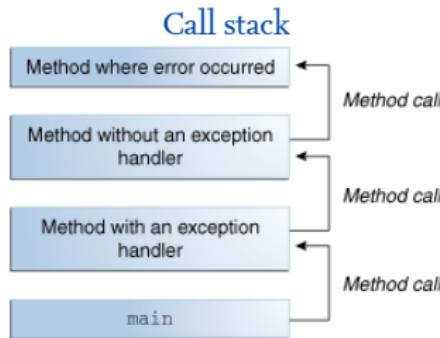
## Exception definition (I)

**Exception:** An error that disrupts the normal execution flow

- File not found, division by zero, invalid argument, etc
- Code cannot be executed
- Elegant solution to handle errors

# Exceptions

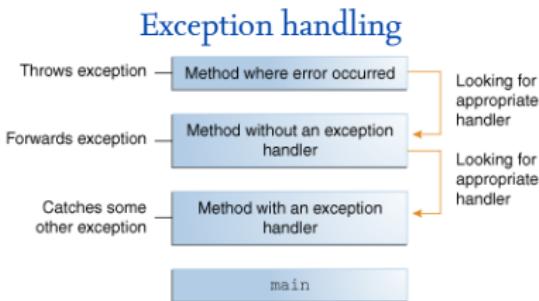
## Exception definition (II)



Call stack: Sequence of invoked methods

# Exceptions

## Exception definition (III)



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, the program finishes otherwise

# Exceptions

## Exception definition (IV)

```
Traceback (most recent call last):
  File "r2p2.py", line 57, in <module>
    start_simulation(args.scenario)
  File "r2p2.py", line 41, in start_simulation
    u.load_simulation(config)
  File "/home/david/repositorios/r2p2/r2p2/utils.py", line 175,
    in load_simulation
      with open(json_file, 'r') as fp:
FileNotFoundException: [Errno 2] No such file or directory: 'foo.json'
```

# Exceptions

## Handling exceptions (I)

Handling an exception requires a try-except statement

- **try:** Encloses the vulnerable code
- **except:** Code that handles the exception (catch in Java/C++)

### try-except statement

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

# Exceptions

## Handling exceptions (II)

### try-except example

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

The exception type contains the error

# Exceptions

## Handling exceptions (III)

### try-except example

```
1 try:
2     f = open('file.txt')
3     s = f.readline()
4     i = int(s.strip())
5 except IOError as err:
6     print("I/O error: " + err)
7 except ValueError:
8     print("Could not convert data to integer")
9 except:
10    print("Unexpected exception")
11    raise
```

### New Python elements

- Raise
- Exception as object