More Python for Videogames

Videogames Technology
Asignatura transversal

Departamento de Automática

Objectives

1. Being able to manipulate files in Python.
2. Being able to understand the usefulness of 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

Reading and writing files
Path

Path: A string that identifies a file in a file system

- On Linux, the path is denoted by:

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

- On Windows, the path is denoted by:

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

And it is represented in Python by:

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

But by also using *raw string*:

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

Universidad
de Alcalá

Reading and writing files
Opening files

- All file operations are made through a *file object*.
- A file is a sequence of bytes. But ..., it's often useful to treat it as a sequence of lines.
- First of all: Call the `open()` function.

The `open()` function

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

Description: The function returns an object file.
- *filename*: String with the file name.
- *mode*: Characters describing how the file will be used:
  - r: Reading mode, *w*: Writing mode, +: Reading/Writing mode.
  - b: Binary mode, *a*: Appending mode.

Remember: Always, always, always close the file: `f.close()`

Methods of file objects

Reading files (I)

The `read()` function

`f.read([size])`
- size: The number of bytes to be read from the file.
- Return value: The bytes read in string.

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()
```

Universidad
de Alcalá

Methods of file objects
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()
```

Methods of file objects
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()
```

Methods of file objects
Writing files (I)

The `write()` function

`f.write(string)`
- string: String to write in file.
- Return value: The number of written bytes.

Example 1: Write a line

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

Methods of file objects
Writing files (II)

Example 2: Write a number

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

Others file management methods

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'
```

Example 1

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

```python
file_ex = open('example.txt', 'r')
num_total_char = 0
count_line = 0

for line in file_ex:
    count_line += 1
    num_total_char += len(line)
file_ex.close()
print('average', float(num_total_char) / float(count_line))
```

Example 2

Reading a line each time

```
1  count_line = 0
2  with open('/Users/julia/code/names.txt') as arch_names:
3      for line in arch_names:
4          count_line += 1
5          print('{:<10}{}'.format(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

- What happens if we need to store complex data structures?
    - Think about lists, dictionaries or even objects ...
    - The `pickle` module comes to help.
- Pickling: Transform an object to string representation.
- Unpickling: Reconstruct an object from its string representation.
- Given an object `x` and a file object `f`.

```
>>> pickle.dump(x, f)
>>> x = pickle.load(f)
```

Universidad
de Alcalá

Serialization

The `pickle` module: Examples

Save a list to a file

```
1  import pickle
2
3  list_number = [2, 5, 7, 8]
4
5  pickle.dump(list_number, open('list.pickle', 'wb'))
```

Load a list from a file

```
1  import pickle
2
3  list_number = pickle.load(open('list.pickle', 'rb'))
4  print(list_number)
```

Universidad
de Alcalá

Serialization

The JSON module: Introduction

JSON: JavaScript Object Notation

- Data format for hierarchical data
- Created in 2001 for stateless client-server communication
- Text-based
- Interoperable
- 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  print(mylist)
```

Exceptions

Errors happen

- We need a mechanism to handle errors
- Some errors happen before execution (*syntax errors*)
- Others are only detected in execution (*runtime errors*)
  - We need tools to handle errors: Exceptions

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

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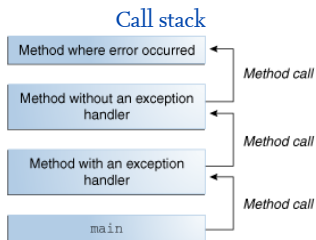
Exceptions
Exception definition (I)

Exception: An error that disrupts the normal execution flow

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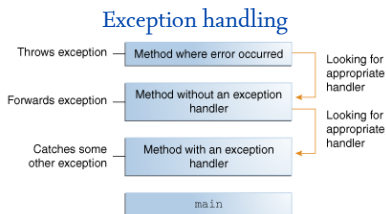- Elegant solution to handle errors

Exceptions
Exception definition (II)

Call stack

Call stack: Sequence of invoked methods

Exceptions
Exception definition (III)

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, 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:
FileNotFoundError: [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
- `catch`: Code that handles the exception

try-catch statement

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

Exceptions
Handling exceptions (II)

try-catch example

```python
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

Universidad
de Alcalá

Exceptions

Handling exceptions (III)

try-catch example

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

New Python elements

- Raise
- Exception as object

Exceptions

Clean-up actions

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

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