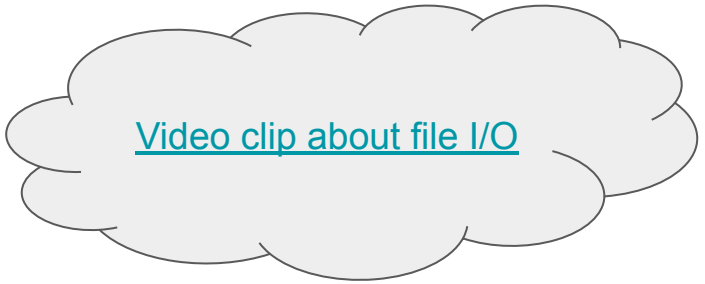
Fundamentos de Programação

António J. R. Neves
João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Summary

- File input/output
- File paths and directories
- Command line arguments
- Exceptions
- Assertions



Text Files

- Most of the programs we have seen so far are transient in the sense that they run for a short time, take input and produce output, but when they end, everything disappears.
- One of the simplest ways for programs to maintain their data is by reading and writing text files.
- A text file is a sequence of characters stored on a persistent medium like a hard drive, flash memory, or CD-ROM.
- Characters are *encoded* in bytes according to a standard coding table such as ASCII, Latin-1 or UTF-8, for instance.

Opening and closing files

- We must prepare a file before reading or writing. This is called **opening** the file.
- The built-in function `open` takes the name of the file and returns a `file` object that we can use to access it.

```
fileobj = open(file_name, 'r')  # open for reading
fileobj = open(file_name, 'w')  # open for writing
```

- More modes: `'r'`, `'w'`, `'a'`, `'r+'`, `'w+'`, `'a+'`, `'rb'`, ...
- After using the file, remember to **close** it.

```
fileobj.close()
```

- Better: use **`with`** statement. It automatically closes files.

```
with open(file_name, mode) as fileobj:
    statements to read/write fileobj
# fileobj.close() not required!
```

Text *versus* binary mode

- Normally, files are opened in ***text mode***. This means:
  - You write/read *strings* of characters (type `str`).
  - Newline characters (`'\n'`) are converted to/from platform-specific *line endings*: LF in Unix, CRLF in Windows. (About CRLF in stackoverflow.)
  - Characters are *encoded/decoded*: each character is converted to/from one or more bytes. (For example, `'á'` → 195, 161 in UTF-8).
  - You may specify the encoding with the optional `encoding=` argument.

```
fileobj = open(file_name,'r', encoding='utf-8')
```

- For files that don't contain text, you should use `'wb'` or `'rb'` to open in ***binary mode***. This means:
  - You write/read strings of *bytes* (type `bytes`, not `str`).
  - No conversions occur.

Reading a file

- We can use a `for` loop to read a file *line by line*.

```
fin = open('words.txt')
for line in fin:          # for each line from the file
    print(repr(line))     # do something with it
fin.close()
```

- Another way is using the `readline` method.

```
while True:
    line = fin.readline()     # returns line to the end
    if line == "": break      # empty means end-of-file
    print(repr(line))
```

- We can also read the entire file as string.

```
text = fin.read()  # read as much as possible (up to EOF)
```

- Or read at most N characters.

```
str = fin.read(10) # read upto 10 chars (empty means EOF)
```

Write a file (1)

- To write to a file, open it with mode `'w'` (or `'a'`).

```
fout = open('output.txt', 'w', encoding='utf-8')
```

- Opening it in `'w'` mode creates a new file or *truncates* an existing one, *i.e.* it deletes the old data and starts from scratch. The `'a'` mode does not truncate, it appends to the end of the file.
- The `write` method puts data into the file.

```
line1 = "To be or not to be,\n"
fout.write(line1)
```

- Again, the file object keeps track of where it is, so if you call `write` again, it adds the new data to the end.

```
line2 = "that is the question.\n"
fout.write(line2)
```

Write a file (2)

- The argument of `write` has to be a string, so we have to convert other types of values.

```
x = 0.75
fout.write('X: ' + str(x))
```

- Or use the string format method.

```
fout.write('{} costs {:.2f}€.'.format('tea', x))
```

- You may also use `print` with the `file=` argument.

```
print('X:', x, file=fout)
print('{} costs {:.2f}€.'.format('tea', x), file=fout)
```

- When you are done writing, remember to close the file!

```
fout.close()   # OR use the with statement
```

Moving the file object's position

- We generally read and write sequentially, from start to end.
- But sometimes we need to "jump" around.
- The `tell()` method tells you the current position within the file.
- The `seek(offset)` method changes the current file position to `offset` bytes from the *start*. (An optional argument can specify a different reference point).

```
a0 = f.readline()   # read a line
pos = f.tell()      # store position
a1 = f.readline()   # read second line
f.seek(pos)         # return to stored position
a2 = f.readline()   # read second line again (a2==a1)
```

Filenames and paths

- The `os` module provides functions for working with files and directories (`os` stands for “operating system”).

  `os.getcwd()` returns the name of the current directory.

- A string that identifies a file is called a ***path***.

```
os.getcwd()          #->    '/home/jmr/FP'
```

- An **absolute path** starts with **/** (the topmost directory).
- A **relative path** starts from the current directory.

```
'aula06/aula06.pdf'
```

- You may find the absolute path to a file:

```
os.path.abspath('aula06/aula06.pdf')
             #-> '/home/jmr/FP/aula06/aula06.pdf'
```

File properties and listing directories

- There are functions to check existence and type of files.
  - `os.path.exists(path)` checks whether a file exists.
  - `os.path.isdir(path)` checks whether a filename is a directory.
  - `os.path.isfile(path)` checks whether it's a regular file.
- And a function to get the contents of a directory.
  - `os.listdir()` returns a list of the files (and other directories) in the given directory.

Example

- The method `walk()` generates the file names in a directory tree by walking the tree either top-down or bottom-up.

```
import os
for root, dirs, files in os.walk(".", topdown=False):
    for name in files:
        print(os.path.join(root, name))
    for name in dirs:
        print(os.path.join(root, name))
```

Command Line Arguments

- The `sys` module provides access to any command-line arguments via the `sys.argv` variable.
  - `sys.argv` is the list of command-line arguments;
  - `len(sys.argv)` is the number of command-line arguments;
  - `sys.argv[0]` is the program (script) name.

```
import sys
print('Number of args:', len(sys.argv), 'arguments.')
print('Argument List:', sys.argv)
```

- Run above script as follows:

```
python3 test.py arg1 arg2 arg3
```

- Produces:

```
Number of arguments: 4 arguments.
Argument List: ['test.py', 'arg1', 'arg2', 'arg3']
```

- Explore `getopt` module

Exceptions

- Python provides an important feature to handle any unexpected events in your program: **exceptions**.
- You've seen exceptions before.

```
int("one")    #-> ValueError: invalid literal for int()
open("foo")   #-> FileNotFoundError: No such file…
```

- When Python encounters a situation that it cannot cope with, it ***raises*** an exception.
- That interrupts the normal flow of execution: the current function is interrupted, then the one that called it, etc., until the main program itself is interrupted.
- Information about the event is transmitted all the way through in an *exception object*.

Handling exceptions

- You can intercept selected exceptions and resume normal execution with the **`try`** statement.
- Example: handle errors accessing files:

```
try:
    fh = open("testfile", "r")
    content = fh.read()
except IOError:
    print("Error: could not open file or read data")
else:
    print("This executes iff no exception occurred")
    fh.close()
```

- The **`except`** clause may name multiple exceptions.
- An **`except`** clause naming no exception, catches all types.

Exception information

- An exception can have an *argument*, which is a value that gives additional information about the problem.

```
def convert(var):
    try:
        return int(var)
    except ValueError as e:
        print("Not numeric:", e)
        return None

m = convert("123")
n = convert("xyz")
```

Play

Raising exceptions

- We can raise exceptions (of any type) by using the **`raise`** statement.

```
def checkLevel( level ):
    if level < 1:
        raise Exception(f"level={level} is too low!")
    # code here is not executed if we raise the exception
    return level

try:
    v = checkLevel(-1)
    print("level = ", v)
except Exception as e:
    print("Error:", e)
```

Play

Assertions

- An **assertion** is a condition that the programmer *knows* (or *believes*) to be true at some point in a program.
- To check an assertion, use **`assert`** *`condition`*.
- This evaluates the condition and, if false, raises an exception of type `AssertionError`.
- If that happens, the programmer learns that there is a **bug**. He/she must find out *why* that assertion failed, and fix the problem.
- If users are confident that the program is correct, they can turn off assertion checking when running program: `python3 -O prog.py`.

Assertions: when to use?

- Assertions at the start of a function, to check if arguments are within the *domain* of the function. (Check preconditions.)

```
def nextDay(y, m, d):
    assert dateIsValid(y, m, d)
    d += 1
    ...     # rest of solution here
    ...
    assert dateIsValid(y, m, d)
    return (y, m, d)
```

- Assertion at the end of a function, to check postconditions.
- Assertions after calling functions for testing results.

```
def testNextDay():
    assert nextDay(1920, 2, 28) == (1920, 2, 29)
    assert nextDay(1920, 2, 29) == (1920, 3, 1)
    assert nextDay(1920, 12, 31) == (1921, 1, 1)
    print("ALL OK!")
```