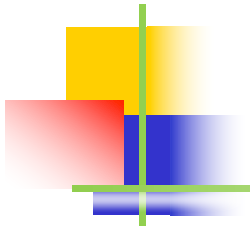

deti universidade de aveiro
departamento de electrónica, telecomunicações e informática

Fundamentos de Programação

António J. R. Neves

João Rodrigues

Departamento de Electrónica, Telecomunicações e Informática

Universidade de Aveiro

an@ua.pt / jmr@ua.pt

http://elearning.ua.pt/

Summary

- Files
- Command line arguments
- Exceptions and 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.

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()
```

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(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(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 up to 10 chars (empty means EOF)
```

Moving the file cursor

- We generally read and write files 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*. (But an optional parameter can specify a different reference point).

```
f.seek(0)
while True:
    part = f.read(2)
    if part == ''
        break
    print(part)
```

Write a file (1)

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

```
fout = open('output.txt', 'w')
```

- If the file already exists,
- Opening it in `'w'` mode creates a new file or *truncates* an existing one, i.e.: it clears out the old data and starts from scratch.
- 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)
```

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

```
fout.close()
```

Write a file (2)

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

```
x = 52
fout.write(str(x))
```

- An alternative is to use the string format method.
- The following example uses the 3 replacement fields to format an integer, a floating-point number, and a string:

```
>>> 'In {:d} years I have spotted {:g} {:s}.'.format(3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

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. A **relative path** starts from the current directory; an **absolute path** starts from the topmost directory (the *root*) of the file system.
- To find the absolute path to a file, you can use:

```
os.path.abspath(path)
```

- `os.path.exists` checks whether a file or directory exists.
- `os.path.isdir` checks whether a filename is a directory.
- `os.path.isfile` checks whether it's a regular file.
- `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 *Python* `sys` module provides access to any command-line arguments via the `sys.argv`:
  - `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

Handle with errors

- Python provides two very important features to handle any unexpected error in your Python programs and to add debugging capabilities: **Exceptions** and **Assertions**.
- An **exception** is an event that occurs during the execution of a program, which disrupts the normal flow of execution.
- In general, when a Python script encounters a situation that it cannot cope with, it *raises* an exception. An exception is a Python object that represents an error.
- When a Python script raises an exception, it must either handle the exception or else it will terminate.

Exceptions (1)

- Handle errors accessing files:

```
try:
    fh = open("testfile", ”r")
    fh.read()
except IOError:
    print("Error: can\'t find file or read data”)
else:
    print("Written content in the file successfully”)
    fh.close()
```

- The `except` statement can also be used with no exceptions or with more than one.

Exceptions (2)

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

```
# Define a function here.
def temp_convert(var):
  try:
    return int(var)
  except ValueError as x:
    print (”Argument does not contain numbers\n", x)

# Call above function here.
temp_convert("xyz")
```

Exceptions (3)

- We can raise exceptions by using the `raise` statement.
- An exception can be a `string`, a `class` or an `object`.

```
def functionName( level ):
  if level <1:
    raise Exception(level)
    # The code here is not executed if we raise the exception
  return level


try:
  l = functionName(-10)
  print ("level = ",l)
except Exception as e:
  print ("error in level argument",e.args[0])
```

Assertions

- An **assertion** is a condition that we know (or require) to be true at some point in a program.
- Use the **`assert`** statement for checking assertions.
- It evaluates the condition and, if false, raises an exception.
- We can turn off assertion checking when we are done with testing of the program (call Python with –O flag).
- We can place assertions at the start of a function to check for valid input, or after a function call to check for valid output.

```
def KelvinToFahrenheit(Temperature):
    assert Temperature >= 0, "Colder than absolute zero!"
    return ((Temperature-273)*1.8)+32
print KelvinToFahrenheit(-5)
#-> AssertionError: Colder than absolute zero!
```