python™

Interaction with files

Data and files

- Let’s go back to our "data-centric" vision of Python programming. We now know how to organize data in lists, and how to build complex structures of nested lists. But this is not sufficient: where are the data?
- Most of the time the data resides in a file, so you need to understand how to interact with them, and how to handle errors that inevitably happen when you must read and write from files.
- In Python there is a dedicated module for the interaction with the operating system, which is called… **os**!

os

The os module contains a variety of low-level functions for the input / output and manipulation of files and directories.

```
>>> import os
>>> os.chmod(path,mode)          # Change file permissions
>>> os.link(src,dst)             # Create a hard link
>>> os.listdir(path)             # Return a list of names in a directory
>>> os.mkdir(path[,mode])        # Create a directory
>>> os.remove(path)              # Remove a file
>>> os.rename(src,dst)           # Rename a file
>>> os.rmdir(path)               # Remove a directory
>>> os.stat(path)                # Return file information
>>> os.symlink(src,dst)          # Create a symbolic link
```

Open

- If you want to use the data contained in a file first you have to **open** it. When you are done with the computing you must **close** the file:

```
>>> the_file = open('filename.txt')
        #<data processing code>

>>> the_file.close()
```

- When the function `open()` is used, it automatically creates and returns an iterator that can loop over the file lines.

- Download the following example file:

`http://bit.do/lettera-txt`

and let's see an example of how to read from file.
In the following I will suppose that my example file is in the directory `/home/domenico` on a Linux workstation

```
>>> import os
>>> curdir = os.getcwd()
/home/domenico
>>> testdir=os.path.join(curdir,'Python')
>>> print testdir
/home/domenico/Python
>>> os.chdir(testdir)


>>> data = open('lettera.txt')
>>> print data.readline(),
TOTO': Giovanotto! Carta, calamaio e penna! Su, avanti, scriviamo...
>>> print data.readline(),
TOTO': Dunque, hai scritto?

>>> data.seek(0)
>>> for each_line in data:
        print each_line,

....

>>> data.close()
```

From the standard library

Gets the current directory

Builds the path of a new dir, and changes the working dir. **The OS module commands does not change with the workstation OS**

open() assigns the data object

readline() gets a line and prints it

seek() moves back the iterator

You can loop directly on the data obj

close() closes the connection with the file

1. A comma after `print` removes the newline
2. The path manipulation functions are independent from the OS used
3. The print lines in python 3.x would be written as:

```
print(each_line, end='')
```

Data processing

- The text that we are using as example seems to follow a specific schema: every line begins with the actor name, followed by a colon, a space, and the text.
- In a first approximation we say that this format is fine, and we process each row in order to extract its components (actor and gag).
- The string object **has a BIF**, `split()`, that seems to be the function we are looking for: it takes as input the character to be used as a separator, and returns a list of resulting strings. In this example we can say:

```
>>> (role, gag) = each_line.split(":")
```

```
>>> import os
>>> data=open('lettera.txt')
>>> for each_line in data:
        role, gag = each_line.split(":")
        print role,
        print " says: ",
        print gag,
....
TOTO'  dice:   Quale signorina?

Traceback (most recent call last):
  File "<pyshell#37>", line 2, in <module>
    role, gag=each_line.split(":")
ValueError: too many values to unpack
```

Note the double assignment

The input line

```
PEPPINO: Hai detto: "Signorina!"...
```

contains two separators, but on the left side of `split()` there are only two variables! Python cannot assign the third part of the string, and **raise an exception.** Let's check the `split()` documentation to see if there are a simple way to resolve this issue. We find that there is an additional parameter **maxsplit**

```
>>> data=open('lettera.txt')
>>> for each_line in data:
        role, gag = each_line.split(":", 1)
        print role,
        print " dice: ",
        print gag,
....
PEPPINO  dice:   Io scrivo.

Traceback (most recent call last):
  File "<pyshell#44>", line 2, in <module>
    role, gag=each_line.split(":", 1)
ValueError: need more than 1 value to unpack
```

Note the parameter

The ValueError is changed, there is another type of format mismatch.
The line
`(pausa)`
does not contains a separator

If you have some programming experience now surely you have recognized a common situation: the data format is not fully known at the time of implementation of the code.

There are two alternative options at this point:
1. continue to add code that deals with the specific cases
2. make errors happen and manage them appropriately

Ad-hoc code

```
>>> data=open('lettera.txt')
>>> for each_line  in data:
        if not each_line.find(":")  == -1:
            role, gag = each_line.split(":",  1)
            print role,
            print " dice: ",
            print gag    ,
```

`find()` is string method, that returns the position of the searched substring, or returns **-1 if the substring is not found**.
Other problems may arise:

1. If the data format changes, you must rewrite and adapt the code.
2. The if condition is not easily understandable
3. If there is a line with a data format slightly different again, the code will not be able to handle it

Errors management

- The second way to deal with the "changing data format" problem is based on the fact that, as we have seen, when Python finds a runtime error it shows the stack traceback followed by an error message: it **raises an exception**.
- Exceptions can be captured and managed, avoiding that an error may stop the program.
- In this way the code becomes more robust, and is able to survive unexpected inputs. You must obviously provide a code that will be executed in response to a raised exception.
- The language keyword to be used in the error management are `try` and `except`.

```
>>> data=open('lettera.txt')
>>> for each_line in data:
        (role, gag) = each_line.split(":",1)
        print role,
        print " says: ",
        print gag,
```

What code segment will you protect with the try statement?

Since we do not want that the print statements are executed if there is an error, we must protect the entire for loop code

For this particular code, we may safely do nothing if there is an error in input formatting, and use the **pass** statement.

```
>>> for each_line in data:
        try:
            (role, gag) = each_line.split(":",1)
            print role,
            print " says: ",
            print gag    ,
        except:
            pass
```

An interesting choice…

- Both the solutions work! The first adds code to handle errors, the second uses the exceptions management.
- There are advantages and disadvantages to both options: the first, however, is exposed to runtime errors if format of input data changes.
- To better understand we make another hypothesis: what if the data file is removed and the code does not find it anymore?
  Both versions stops with an IOError. How can we handle this problem?

IOError ad hoc code

Add extra logic to handle errors

```
>>> import os
>>> if os.path.exists('lettera.txt'):
        data=open('lettera.txt')
        for each_line in data:
            if not each_line.find(":") == -1:
                (role, gag) = each_line.split(":", 1)
                print role,
                print " dice: ",
                print gag    ,
        data.close()
    else:
        print "The datafile is missing!"
```

In words:
The os module is imported, the function `path.exists` checks that the file exists before trying to open it. Every file line is processed, but only after a format check: if we find the colon we can go on, breaking the line in two substring, otherwise the line is ignored. At the end the data file is closed.

IOError exception management

Using exceptions to handle errors

http://bit.do/Read1-py

```
>>> try:
        data=open('lettera.txt')
        for each_line in data:
            try:
                (role, gag) = each_line.split(":", 1)
                print role,
                print " says: ",
                print gag    ,
            except:
                pass
        data.close()
    except:
        print "The datafile is missing!"
```

In words:
A data file is opened, every line is splitted in two substrings, the data are extracted and printed, the data file is closed. Every possible error is managed by exceptions.

Comparison

- The logic complexity of the first option increases with the errors you have to deal with, and it becomes increasingly difficult to follow the actions performed by the code.
- In the second option it is always clear what the code are doing, and the complexity does not grows with the possible errors to manage.
- Using exception handling you can concentrate on what really the code has to do, delegating to the exceptions any error management.

Error types

- The code that we have written for the exception handling is too efficient: *each error type* is captured, and no matter what **type** of error we have, the code inside the `except` block is always run.

- This is not always what we want. One thing is to accept that a non-standard line is silently skipped, another thing however is to permit that any type of error is silenced. In that case we cannot ever know that an error occurred.

- You can make sure that the try-except code handles only one type of error, specifying it on the except line as follows:

```
>>> try:
        data=open('lettera.txt')
        for each_line in data:
            try:
                (role, gag) = each_line.split(":", 1)
                print role,
                print " says: ",
                print gag    ,
            except ValueError:
                pass

        data.close()
    except IOError:
        print "The datafile is missing!"
```

If at run time we have an error of different type, it is no longer managed by the code, and although there is a stop at least we are aware of the problem.

If you want to print the exact IOError message, instead of the custom string, you can use:

```
except IOError, errMessage:
    print errMessage
```

Write data

- Data processing not only needs data, but also produces data. Programs typically save data in file system files, show them on the screen or transfer them over the network.

- To give a meaningful example, with a data processing between data read and write, now we'll go back to the previous code and we'll change it adding some data modification

- Our task will be to create two different lists, one for each actor in the dialog, that contain only the relative actor lines.

http://bit.do/Read1-py

```
>> try:
        data=open('lettera.txt')
        for each_line in data:
            try:
                (role, gag) = each_line.split(":", 1)
                print role,
                print " says: ",
                print gag    ,
            except:
                pass

        data.close()
    except:
        print "The datafile is missing!"
```

Load it in IDLE and modify it:

1. Create two empty lists, `toto` and `peppino`, and remove the print instructions
2. Remove the unwanted white spaces from `gag`
3. Write the code that adds `gag` to the right list depending on the value of `role`
4. At the end print the two lists `toto` and `peppino` on the screen

Puzzle time…

```
elif role == 'PEPPINO':
if role == 'TOTO'
toto=[]
peppino.append(gag)
peppino=[]
```

```
print toto
print peppino
toto.append(gag)
gag=gag.strip()
```

Solution

```
toto = []
peppino = []

try:
    data=open('lettera.txt')
    for each_line in data:
        try:
            (role, gag) = each_line.split(":", 1)

            gag=gag.strip()

            if role == "TOTO'":
                toto.append(gag)
            elif role == 'PEPPINO':
                peppino.append(gag)
        except ValueError:
            pass
    data.close()
except IOError:
    print "The datafile is missing!"


print toto
print peppino
```

Create two empty lists, `toto` and `peppino`

`http://bit.do/Read2-py`

Remove the unwanted white spaces from `gag`

Add `gag` to the right list depending on the value of `role`

print the two lists `toto` and `peppino` on the screen

```
gag=gag.strip()
```

Here we have something it's worth noting.
Do you remember that the strings are **immutable** objects? How is it possible then that the `strip()` function removes the blank space from the beginning and the end of a string?

What really happens is that the `strip()` function actually creates a NEW string.
In the above code line, a reference to the new string is assigned to the name `gag`, replacing the reference to the old string.
If the old string does not have any reference pointing to it, is considered unused, and its memory space reallocated.

Warning: if you don't assign the reference to a name on the left, the original string remain unchanged!

```
>>> gag='    prova   '
>>> id(gag)
4480702912
>>> gag=gag.strip()
>>> id(gag)
4480702816
>>> gag
'prova'
```

Write a list in your file

- We have printed the new lists on the screen, but now we want to write it in a data file.

- The `open()` function, that we have already used, by default opens the file for reading, but it can also be used to open files in write or append mode. To see the help about `open()`, and for example about the `strip()` function seen before:

```
>>> help(open)
>>> help(str.strip)
```

- Once you have the file object, you can use its `write()` method.
- In **Python 3.x** there is a `print()` function, which can take as the first argument the string to be printed, and as the second argument the name of the object file to use::

```
>>> outFile=open("data.out", 'w')
>>> print("Python for dummies", file=outFile)
```

http://bit.do/Write1-py

```
toto = []
peppino = []
try:
    data=open('lettera.txt')
    for each_line in data:
        try:
            (role, gag) = each_line.split(":", 1)
            gag = gag.strip()
            if role == "TOTO'":
                toto.append(gag)
            elif role == 'PEPPINO':
                peppino.append(gag)
        except ValueError:
            pass
    data.close()
except IOError:
    print "The datafile is missing!"

try:
    toto_file=open('toto_data.txt','w')
    peppino_file=open('peppino_data.txt','w')
    toto_file.write(str(toto))
    peppino_file.write(str(peppino))
    toto_file.close()
    peppino_file.close()
except IOError:
    print 'File Error'
```

When you run it in IDLE, you cannot see any output on the screen

http://bit.do/Write2-py

```
toto = []
peppino = []

try:
    data=open('lettera.txt')
    for each_line in data:
        try:
            (role, gag) = each_line.split(":", 1)
            gag=gag.strip()
            if role == "TOTO'":
                toto.append(gag)
            elif role == 'PEPPINO':
                peppino.append(gag)
        except ValueError:
            pass
    data.close()
except IOError:
    print("The datafile is missing!")

try:
    toto_file=open('toto_data.txt','w')
    peppino_file=open('peppino_data.txt','w')
    print(toto, file=toto_file)
    print(peppino, file=peppino_file)
    toto_file.close()
    peppino_file.close()
except IOError:
    print('File Error')
```

Python 3.x version

But....

- What happens to our output file if a `write` fails causing an **IOError**?
- When you are reading data from a file, an IOError can be quite "boring", but rarely dangerous. The same issue is very different if you are writing to a file: you must handle the error BEFORE the file is closed, otherwise the written data can be left in an inconsistent state, or not be written at all.
- In the previous code if there is an exception on a `write()`, the execution jumps on the exception block and the `close()` on the open file are not executed: the data can potentially be corrupted.
- **You have to be sure that every open file is ALWAYS closed before the end of the script run, regardless of any errors that may occur when writing.**

try-except-finally

http://bit.do/Write3-py

```
try:
    toto_file=open('toto_data.txt','w')
    peppino_file=open('peppino_data.txt','w')
    toto_file.write(str(toto))
    peppino_file.write(str(peppino))
except IOError:
    print 'File Error'
finally:
    toto_file.close()
    peppino_file.close()
```

The `finally` is ALWAYS executed,regardless of whether or not an exception occurs.

But there's more: what happens if with a code like this we try to open a file that does not exist?

```
>>> try:
        mdata=open('missing.txt')
        print mdata.readline(),
    except IOError:
        print 'File Error'
    finally:
        mdata.close()
```

mdata is not created, and raises an exception **in** `finally:`

```
File Error
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
NameError: name 'mdata' is not defined
```

with

To avoid further complicating the code, for example inspecting the `local()` dictionary that contains the defined name, you can use the statement **with**

```
try:
    with open('toto_data.txt','w') as toto_file:
        toto_file.write(str(toto))
    with open('peppino_data.txt','w') as peppino_file:
        peppino_file.write(str(peppino))
except IOError as err:
    print 'File Error:' + str(err)
```

http://bit.do/Write4-py

Now you are **sure** that, whatever happens during the data writing, Python will correctly close any open file.

Note that this time we want to print the error message generated by the exception

```
>>> try:
        with open('missing.txt') as datam:
            print mdata.readline(),
    except IOError as err:
        print 'errore', err
```

```
errore [Errno 2] No such file or directory:
'missing.txt'
```

Pickle

With a `with` instruction you can now read the new file:

```
>>> with open('toto_data.txt')  as mdf:
        print mdf.readline()
```

Ooops! We had a list in memory, but we saved a string!
One possibility would be to write a code to re-interpret it as a list, but it would be much more easy to understand how to save directly a list and reload it in memory.
Python comes with a standard module called **pickle**, that can save and load any data object, including lists. When you load the data from file, the objects are recreated in memory with their original format.
Think about how it might be useful for a program that makes long calculations.

```
import pickle
try:
    with open('toto_data2.txt','wb')  as toto_file,
open('peppino_data2.txt','wb')  as peppino_file:
        pickle.dump(toto,toto_file)
        pickle.dump(peppino,peppino_file)
except IOError as err:
    print 'File Error:' + str(err)
except pickle.PickleError  as perr:
    print 'Pickling error:  ' + str(perr)
```

http://bit.do/Write5-py

Pickle Format

Now if you open the files that pickle has created, you will realize that pickle uses its own format, or protocol, to write data.

Let's see how to import the saved data into another program:

http://bit.do/Read3-py

```
>>> import pickle
>>> new_toto=[]
>>> new_peppino=[]
>>> try:
        with open('toto_data2.txt','rb') as toto_file,
open('peppino_data2.txt','rb') as peppino_file:
            new_toto=pickle.load(toto_file)
            new_peppino=pickle.load(peppino_file)
    except IOError as err:
        print 'File error: ' + str(err)
    except pickle.PickleError as perr:
        print 'Pickling error: ' + str(perr)
>>> new_toto[3]
'"Signorina!"...'
```

Python essentially takes care of all the details for saving and loading lists, and objects, in files.