Introduction to programming.

Lecture 8: File processing

Lecturer : Xavier Parent $\mathrm{xavier.parent@uni.lu}$

Paths

Working with files and interacting with the file system are important for many different reasons. The simplest cases may involve only reading or writing files, but sometimes more complex tasks are at hand. Maybe you need to list all files in a directory of a given type, find the parent directory of a given file, or create a unique file name that does not already exist.

A prerequisite of all this is the ability to handle **paths**. A path is the address where the file is stored.

Here we only give basic information to access a file from a .py file anywhere on your computer (provided that you actually have the authorization to access the said file).

Since paths are, for obvious reasons, specifically formated strings, you need to use a module to deal with these objects. This module is called $\mathbf{pathlib}$. The main class is called $\mathbf{Path}$. This is what you usually need.

In [ ]:

```
from pathlib import Path
```

Then, a path can be specified by calling $\mathbf{Path}(stringdirect)$ where $stringdirect$ is a string containing the address to a directory.

Interest of pathlib

One of programming's little annoyances is that Microsoft Windows uses a backslash character between folder names where other OS uses a forward slash. Pathlib was introduced in version 3 in order to handle this incongruity. You pass a path or filename into a new Path() object using forward slashes and it handles the rest.

The Path() object will convert forward slashes into the correct kind of slash for the current operating system. Nice! If you want to add on to the path, you can use the / operator directly in your code.

In [ ]:

```
p=Path('/Desktop')
print(p)
q=Path('notsocorrectlyformatted**address')# You don't really need it to be a path "per
 se"
print(q)
```

In [ ]:

```
#It is possible to add a further sub-directory to a path.
p=p/'Introduction_To_Programming'
print(p)
```

In [ ]:

```
from pathlib import Path, PureWindowsPath

p = Path("source_data/text_files/raw_data.txt")

# Convert path to Windows format

path_on_windows = PureWindowsPath(p)

print(path_on_windows)

# prints "source_data\text_files\raw_data.txt"
```

Some typical directories: cwd, home, root and parent

- current working directory
- home directory (created for you when you first logged in)
- root directory (at the top level of the directory)
- parent directory

In [ ]:

```
#p=Path.cwd() # The "cwd" method returns the Path of the current working directory
#p = Path(".") # dot means "this directory"
#p=Path("..") # dot dot means"the parent direcotry"
p=Path('/') # root directory
print(p)
```

In [ ]:

```
#Path.home() #Home directory
p=Path.home()/'momo'
print(p)
```

Basic use of pathlib

- list sub-directories and files
- list all the files with a given extension in a directory
- navigate inside a directory tree
- query path properties (does a path exist?)
- open a file
- create/delete a directory (for deletion, the directory must be empty)
- delete a file or a symbolic link
- convert a path-object to a string

rmdir

A methods that removes the directory for the given path. **A security has been added to this method, to remove a directory, the directory must be empty.** It raises an Error if the file cannot be found.

In [ ]:

```
p = Path(".") # dot means "this directory"
[x for x in p.iterdir() ]
```

In [ ]:

```
# glob(pattern) returns an iterator containing all paths whose format follows the patte
rn given
# A pattern is always specified with a *
p=Path.home()/'Desktop/'
[x for x in list(p.glob('*.txt'))]
```

In [ ]:

```
# navigating inside a tree directory

p = Path("/etc")
q = p / "init.d" / "reboot"
print(q)
```

In [ ]:

```
# checking if a path exists

p=Path('Nonexistingpath') # I am creating an object path for a path that does not exist
print(p.exists())  # Use the 'exists' method to make sure that a path actually leads so
mewhere
p=Path.home()/'Desktop' # Some path that exists
print(p.exists()) # Okay
```

In [ ]:

```
# asking if a path leads to a directory or a file

p=Path.home()/'Desktop'
print("\"the path {} is pointing to a directory  \" : {}".format(p,p.is_dir()))
print("\"the path {} is pointing to a file  \" : {}".format(p,p.is_file()))
p=p /'Introduction_To_Programming'/'Homework1.pdf' # For this to work,
                                                  # you need to have this path availab
le on your computer
print("\"the path {} is pointing to a directory  \" : {}".format(p,p.is_dir()))
print("\"the path {} is pointing to a file  \" : {}".format(p,p.is_file()))
```

In [ ]:

```
# opening a file

with p.open() as f:
    f.readline()
```

In [ ]:

```
p=Path.home()/'Desktop/Nowitexists/Test'
print(p.exists())
p.mkdir() # Creates a directory at a given address,
print(p.exists())
```

In [ ]:

```
p.mkdir() # prints an Error message if the directory already exists...
```

In [ ]:

```
# remove (empty) dir
p=Path.home()/'Desktop/Nowitexists/Test'
p.rmdir()
```

In [ ]:

```
# delete sumbolic link  or file ln -s bible.txt myfile
p=Path.home()/'Desktop/bible.txt'
p.unlink()
```

In [ ]:

```
#parts is an attribute of a path containing all upper directories of the path given
p.parts
```

In [ ]:

```
#convert a path-object to a string

p=Path.home()/'Desktop/Nowitexists/Test'
str(p)
```

In [ ]:

```
# convert a path-object to a string formatted for the present OS
import os
from pathlib import Path

c = str(Path("/")/'test')
os.system("sudo mkdir {c}")
```

Manipulating files

2 extra modules of interest:

- **os** .
- **shutil**

They have many features other than those covered in this lecture, in particular **os**

In [ ]:

```
import os
import shutil
```

Basic use of os

- delete a file or folder
- rename file
- move file

Basic use of shutil

- copy file from one directory to the other

There are different methods to copy a file with **shutil** : **copy**, **copy2** and **copyfile**. In any case the two arguments of these copy functions are an original path identifying the file to be copied and a path where the copy should go. In **copy** or **copy2** the goal path can either point toward an existing directory (in which case the copied file will go into this directory and have the same name as the original file's one) or point toward a maybe non-existing file in an existing directory in which case the copied file will be given the name of the maybe non-existing file. With **copyfile**, the second argument cannot be a path for a directory.

The difference between **copy** and **copy2** is that the second copy, as much as possible, the metadata as well. When you don't know which one to use, just use **copy2**.

copy2 is the same as the **copy** function except it copies the file metadata with the file. The metadata includes the permission bits, last access time, last modification time, and flags.

In [ ]:

```
from pathlib import Path
import os
p=Path.home()/'Desktop/Int2Prog'
#os.rename(p/'Homework2.pdf',p/'Homework1000.pdf') # rename file
#os.rename(p/'Homework1000.pdf',p/'Int2Prog'/'Homework1000.pdf')## move the file
#os.mkdir(p/'fakedir')    # create a folder
#[x for x in p.iterdir()]
os.rename(p/'fakedir', p/'fakedir2')    # rename a folder
[x for x in p.iterdir() ]
```

In [ ]:

```
os.rename(p/'Homework2.pdf',p/'Homework1000.pdf') # Change a name
```

In [ ]:

```
os.rename(p/'Homework1000.pdf',p/'Introduction_To_Programming'/'Homework1000.pdf')# mov
e the file
```

In [ ]:

```
import shutil
```

```
shutil.copy2(p/'Introduction_To_Programming'/'Homework1000.pdf',p)
```

Open and close a file

Once a file f is closed, f is still assigned but you cannot use the file anymore, to do it, you need to reopen it again. The reason is clear, while the file f is closed it only uses as much memory as needed to specify the path to the file but when the file is open the memory you use is as big as the file is. In particular, you will have an Error message if you attempt to open a file which is twice bigger as what your session is authorized to use.

Let us say that leaving open many files is a very bad practice since it will have a bad influence on the efficiency of your program. The Python documentation highlights a specific command to make sure that you only open a file when you need it. The command is $\mathbf{with}$. It is called a "context manager". The file is automatically closed when you exist the block. This helpt to circumvent memory issues.

Opening a file

Command 1 : file = open(path-to-filename, 'mode', 'encoding')

Command 2 : with open(path-to-filename, 'mode', 'encoding') as file:

Remarks:

- The mode argument is optional; 'r' (see below) will be assumed if it’s omitted.
- The default encoding for Python source code is UTF-8. If no encoding is specified, then the defult one is used.
- With command 1, the file must be closed explicitily: file.close()
- Command 2 does not require this.

Available access modes

- r: Opens the file in read-only mode. Starts reading from the beginning of the file and is the default mode for the open() function.
- rb: Opens the file as read-only in binary format and starts reading from the beginning of the file. While binary format can be used for different purposes, it is usually used when dealing with things like images, videos, etc.
- r: Opens the file in read-only mode. Starts reading from the beginning of the file and is the default mode for the open() function.
- rb: Opens the file as read-only in binary format and starts reading from the beginning of the file. While binary format can be used for different purposes, it is usually used when dealing with things like images, videos, etc.
- r+: Opens a file for reading and writing, placing the pointer at the beginning of the file.
- w: Opens in write-only mode. The pointer is placed at the beginning of the file and this will overwrite any existing file with the same name. It will create a new file if one with the same name doesn't exist.
- wb: Opens a write-only file in binary mode.

Available access modes (con't)

- w+: Opens a file for writing and reading.
- wb+: Opens a file for writing and reading in binary mode.
- a: Opens a file for appending new information to it. The pointer is placed at the end of the file. A new file is created if one with the same name doesn't exist.
- ab: Opens a file for appending in binary mode.
- a+: Opens a file for both appending and reading.
- ab+: Opens a file for both appending and reading in binary mode.

The file Object Attributes

Once a file is opened and you have one file object, you can get various information related to that file. Here is a list of all the attributes related to a file object

- file.closed: return true if the file is closed
- file.mode: returns access mode with which file was opened.
- file.mode: returns access mode with which file was opened.

In [ ]:

```
# opening a file
p=Path.home()/'Desktop/Nowitexists/'
print(p)
f=open(p/'trash.txt', 'r') #This creates a variable f which "contains" the specified fi
le
```

In [ ]:

```
print(type(f))
```

In [ ]:

```
print(f.read())#You can do many things with your file this is what we are going to see
 next
```

In [ ]:

```
f.close()# close the file
f.closed # Ask if the file is closed
```

In [ ]:

```
f.read()
```

In [ ]:

```
p=Path.home()/'Desktop/Nowitexists/'
with open(p/'trash.txt', 'r') as f:
    print(f.read()) #Do whatever you want with your file : the file is open
f.closed #then the file is closed
#As we can see, the file is only open in the indented block.
```

Text files versus binary files

As we have mentioned before you can either consider your files as text files or binary files. This is not an exhaustive list but here are some extensions that you should consider to be openable as text files :

- python files .py
- text files .txt
- LaTeX files .tex
- ...

By opening these files as text files, you will directly see the text written in it. Be careful with problems of endocage though especially if it is not in **utf-8**.

If you don't really know if your file should be read as a text file, it probably means that you should read it as a binary files. This is definitely not an exhaustive list but make sure you don't write on **.jpg**, **.png** or **.pdf** files as a text files, you will get an Error message.

Warning : modifying a binary file is a very delicate operation. Modifying a picture to make another picture requires a particular knowledge on how the picture is encoded.

All files (even text files) are encoded as binary files. Depending on wether you want your file to encode a music, a picture, a text, a video ; the binary file will be radically different. Text files end up being very simple binary files. However, other kinds of files are encoded in a very specific way so that if you add a random binary string at the end of it, you will end up with a non-readable file that will be recognized as corrupted by the software you use to read it.

Of course there are ways to temper with binary files of any source but you need to do it carefully. Next lecture will be on this particular topic.

Just remember as a general law that files which are not meant to be read/written as text files are most probably using very particular enconding. For the rest of this lecture we will only be interested in text files.

In [ ]:

```
```

In [ ]:

```
p=Path.home()/'Desktop/Int2Prog/'
#with open(p/'dummy.txt','r+') as f:
#    f.write('929838')

#    print(f.read())

with open(p/'test.jpg', 'r+b') as f:
    f.write(b'929838') #Remark that b'....' means bytes string, a particular type
                                            #to be used when you are writing on fi
les as bynary files.
    print(f.read(100))
```

In [ ]:

```
with open(p/'dummy.txt','w') as f:
    f.write('This is about everything I am willing to write')#Write on a file
bytedata=bytes(b'This is about everything I am willing to write') #Convert it as a bina
ry string
print(list(bytearray(bytedata))) # You see the characters encoded as integers between 0
and 2^8-1
with open(p/'dummy.txt','rb') as g:#Read the   file as a binary file
    databyte=g.read()

databyte==bytedata # Both are equal meaning that the encoding is quite clear for text f
iles
```

Read / write on text file

Between the opening and the closing of a file you can read it or write on it. Remember that when you open a file, you specifically say what you do with it, **read**, **write**, **both** or **write at the end**.

In this lecture we deal with long text files. We take the bible. It will be made available on the moodle. If you want to play with it, download it and move it to your desktop.

In [ ]:

```

```

Read() method

The read(size) method returns the specified number of bytes (as indicated by the size parameter) from the file. Default is -1 which means the whole file. It is returned as a single string.

Remember: One byte = 1 character.

When working with files in read mode, it is usually advised to open it first by specifying the encoding type.

Most of the text editors allow you to check the encoding used in a given file. You can also use Firebox or do it from terminal

- Firebox: Drag and drop the file into firefox Right click on the page Select "View Page Info" and the text encoding will appear on the "Page Info" window.
- Terminal. Go to directory where the file is, and type "file filename". On windows you need "Cygwin" installed (a linux emulator)

List of the standard encodings: https://docs.python.org/2.4/lib/standard-encodings.html (https://docs.python.org/2.4/lib/standard-encodings.html)

Looping over a file object

Finally, remark that the opened file can be seen as an iterator. It is an iterator of the lines contained in the text file. I

It is the best way to read/print the whole text as if it were in the text file.

When you want to read – or return – all the lines from a file in a more memory efficient, and fast manner, you can use the loop over method.

file = open(“testfile.txt”, “r”)

for line in file: do ...

Other read-related methods

readline(size)

If no argument is passed or None or -1 is passed, then the entire line (or rest of the line) is read.

readlines()

This reads the remaining lines from the file object and returns them as a list.

Tell

Remark that when you call $textfile.\mathbf{read}()$ twice the first call returns the whole text whereas the second call returns an empty string. The reason is that the $\mathbf{read}$ method does not only return the text file but also keeps on progressing through the file. So that when you read everything, the pointer is necessarily at the end of your file. To know where is the pointer inside the file, you can use the $\mathbf{tell}$ method.

The outputed number is in byte.

In [ ]:

```
from pathlib import Path
p=Path.home()/'Desktop/Int2Prog'
textfile = open(p/"words.txt",mode = 'r',encoding = 'utf-8') # try ascii
textfile.read(2)
```

In [ ]:

```
textfile=open(p/'bible.txt',mode='r',encoding='utf-8') #Let's open it once and for all
 in "reading mode" the file.
```

In [ ]:

```
from pathlib import Path
#textfile=open(p/'bible.txt',mode='r',encoding='utf-8')
#textfile.read(4)
#textfile.read(0)
#textfile.read(1)
#textfile.read(4)
```

In [ ]:

```
textfile.read() # It does not contain anything
```

In [ ]:

```
with open(p/'bible.txt',mode='r',encoding='utf-8') as f: #Let's open it once and for al
l in "reading mode" the file.
        f1=f.read()
        for x in f1:
            print(x, end='')

#if __name__=="__main__":
#    main()
```

In [ ]:

```
with open('bible.txt', 'r') as infile:
    for line in infile:
        print('> {}'.format(line))
```

In [ ]:

```
p=Path.home()/'Desktop/Int2Prog'
texfile=open(p/'bible.txt','r')
texfile.read.split('\n')
```

In [ ]:

```
textfile=open(p/'bible.txt','r')
print("Pointer before reading : ",textfile.tell())
textfile.read()
print("Pointer after reading : ",textfile.tell())
textfile.close()
```

Be careful with the **tell** method, especially if you want to use it as an index. As we can see in this small example.

In [ ]:

```
textfile=open(p/'Adumbfile.txt','w')
textfile.write('français')
textfile.close()
textfile=open(p/'Adumbfile.txt','r')
i=0
j=textfile.tell()
c=textfile.read(1)
while i<len('français'):
    i+=1
    j=textfile.tell()
    c=textfile.read(1)
    print("character read : ",c,"| index ",i, "| tell value",j)
```

It also happens in the big text we were looking at.

In [ ]:

```
textfile=open(p/'A_madcap_Cruise.txt','r')
data=textfile.read(20000) # Encodes the  first 20 000 characters
textfile.close()
textfile=open(p/'A_madcap_Cruise.txt','r')
i=0
j=textfile.tell()
c=textfile.read(1)
while i<len(data):
    i+=1
    j=textfile.tell()
    c=textfile.read(1)
    print("character read : ",c,"| index ",i, "| tell value",j) #Go to 19988

# Some special character are read as if they were two. It is related to an encoding pro
blem.
# It is due to the use of non-standard character (i.e. non-English characters mainly).
```

In [ ]:

```
# Exercise: write a script that prints the longest word in the file bible.txt
```

It is explicitely written in the Python documentation that the $\mathbf{tell}$ method works without surprise only with binary files. One should use carefully the $\mathbf{tell}$ method when dealing with text files.

In [ ]:

```
textfile=open(p/'A_madcap_Cruise.txt','r')
for x in textfile:
    print(x)
textfile.close()
```

Write mode

There are, a priori, three modes to write on a given file : **'w'**, **'r+'** and **a**. First let us do a copy of our text file.

If you directly open your file with the write mode then you will erase everything from your file and end up with an empty file.

Warning : by using the mode 'w' when you open a file you are simply deleting the file if it exists and creating a new empty file with the same name.

If you just want to modify a text file you want to use **'r+'** mode which is both reading and writing.

Remark that the sentence is written **over** the former text and not before.

If you want to write at the end of your file you should use the **'a'** mode.

In [ ]:

```

```