UNIVERSITY OF TWENTE.

Files and Python

External data containers; how to read and how to create them

1

Objectives

After this lecture, students will be able to:

- Work with files and directories (aka folders)
- Write and read text files

2

What is a file?

Files store data, which can be text, numbers, binary encoded data

Such data can be read and modified; we can also write data to a file

Files can be stored in different locations: a local hard-disk, a usb drive, a disk on the network, and …

More remotely on the internet, reachable through a url. (An internet address known as a *uniform resource locator*.)

Sometimes one will need to search programmatically for files, on a local disk, or on the internet.

File path

A location of a file is calls a file path. It is denoted as a string.

Normal string literals use \ as an escape character. This allows the inclusion of non-standard characters in a string. E.g., when you want to have a single quote inside a single-quoted string, you use \'

Thus, \\ stands for a single backslash:

'C:\\ITC\\ProgrammingSkills\\'

'C:/ITC/ProgrammingSkills/'

Raw string literals, flagged by an r, do *not* give \ a special meaning:

r'C:\ITC\ProgrammingSkills'

Raw string literals

- treat backslashes as literal characters
- cannot end with a backslash
- flag b denotes a bytestring literal

Dictionary as file container

Every file is held in some directory (aka folder). The location of a file is determined by the file path. *File path = directory path + file name*.

Can be absolute path
'C:/ITC/ProgrammingSkills/myfile.txt'

or relative path
'../GISModelling/myotherfile.txt'

'.' iscurrent directory, and '..' is parent directory.

Pragmatics: use of relative paths is good coding style as it allows to place your program anywhere on disk and continue to operate locally.

File path types

Absolute path

PROs**:** the location is exact

CONs**:** a change in the directory structure will change the absolute path

Relative path

PROs**:** moving the working directory will not create problems (if all the files are in the same directory or its children)

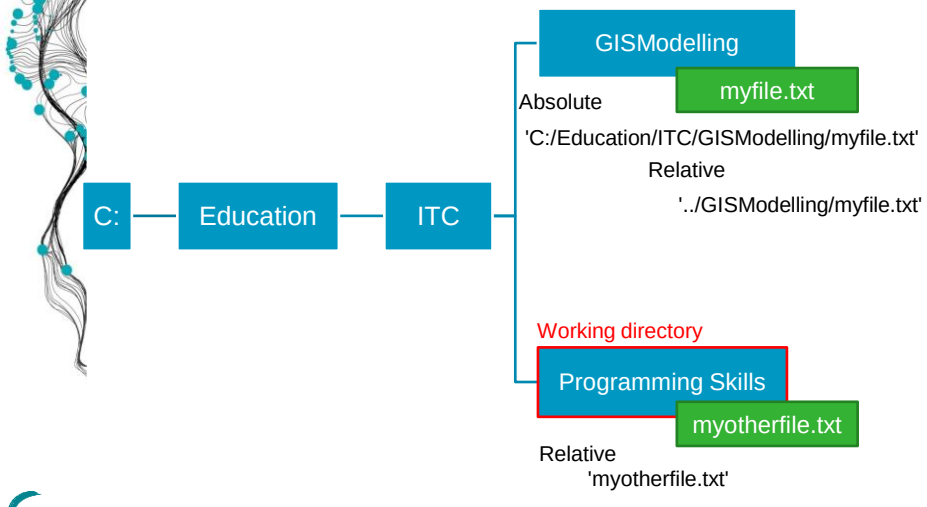
CONs**:** the exact location is unknown

Current working directory

The current working directory (CWD) is the directory where the program was started from.

Relative paths are relative to the CWD.

Files can be in different containers



Module *os* to work with the operating system

Use the os module to determine the CWD or to change it

```
import os

os.getcwd()
'C:\\Education\\ITC\\ProgrammingSkills'

os.chdir( r'C:\Education\ITC\GISModelling' )

os.getcwd()
'C:\\Education\\ITC\\GISModelling'
```


Module *os*

```
os.path.abspath( 'myfile.txt' )      # what is the absolute path?
os.path.exists( directory )          # does the directory/file exist?
os.path.isdir( directory )           # is this a directory?
os.path.isfile( 'myfile.txt' )       # is this a file?

os.listdir( directory )               # list of files and directories
os.walk( directory )                  # get (sub)files and (sub)directories

for rootdirectory, dirs, files in os.walk(directory):
    print ( rootdirectory, dirs, files )
```

… and many others …

Step 1: Open the file

A file must be opened before reading, updating, deleting data

You can read from the beginning of the file to the end, or you can skip some parts

When you finish, you need to close the file.

Pragmatics: always close files as a good programming practice.

File types

Different kinds of file have different types of content:

- Text files (.txt, .csv, .html, .xml …)
- Binary files (.doc, .png, .exe …)

We will focus mostly on text files.

Special case are .csv (comma-separated values) which hold one record of data per line. It is a data exchange workhorse file type.

Web harvesting techniques fetch .html or .xml by url and interpret the contents.

Text files

A text file contains printable characters, tabs and spaces, organized into lines that are separated by newline characters

- printable chars: a-z; A-Z; 0-9, '~!@#,;:', …
- newline: '\n' (line break, line terminator, end-of-line (EOL))
- whitespace characters: ' ', '\t'

How to open a file for *write* actions

In the program, create a Python file object *f* as follows

f = open('myfile.txt', 'w')

By default this is a text file. Object *f* "points to that file."

The second parameter is called *mode.* Mode *'w'*: we will write to the file. And loose already existing file contents.
Use mode *'x'* for exclusive creation # check file existence

print(f)
<_io.TextIOWrapper name='myfile.txt' mode='w' encoding='cp1252'>

by default the local character encoding is used

Performing file *write* actions

```
f.write( 'I write something\n' )          % Note we are adding a newline

f.write( 'I write something else\n' )

f.close()                                 % since we have opened,
                                          % we should also close the file at some point

f.write( 'I am writing again' )           % cannot do this after close()
ValueError: I/O operation on closed file
```

How to open a file for *writeln* actions

Writing multiple lines

```
f = open( 'myfile.txt', 'w' )

lines = [ 'I write something\n', 'I write something else\n' ]

f.writelines( lines )

f.close()
```

How to append to a file

Open a file object, and use mode 'a'

```
f = open( 'myfile.txt', 'a' )

print( f )
<_io.TextIOWrapper name='myfile.txt' mode='a' encoding='cp1252'>

f.write( 'I append this line of text.' )

f.close()
```

If the file does not exist, a new file will be created

How to open a file for *read* actions

Open a file object with mode 'r'

```
f = open( 'myfile.txt', 'r')

print( f )
<_io.TextIOWrapper name='myfile.txt' mode='r' encoding='cp1252'>
```

If you do not specify a mode, you will by default operate in mode ‘r’

How to read a text file

```
text = f.read()                                  # read the whole file

print( text )
I write something
I write something else
I append this line of text.
```


How to read character strings

```
f.seek( 0 )                                      # go back to the beginning

text = f.read( 18 )                              # read 18 characters
print( text )
I write something

text = f.read( 32 )                              # read another 32 characters
print( text )
I write something else
I append

text = f.read( 5 )                               # read 5 more
print( text )                                    # nothing read and nothing to print
```

The use of iteration in file operations

You can use iteration:

```
f = open( 'myfile.txt', 'r' )
for line in f:
    print (line, end = ' ')
I write something
I write something else
I append this line of text
```


Stripping strings

Stripping a string takes away the whitespace characters at the start and at the end

```
f.seek( 0 )
text = f.readlines()                    # reads all the lines and creates a list
print( text )
[ 'I write something\n', 'I write something else\n', 'I append this line of text' ]

for i in range( len(text) ):
    text[ i ] = text[ i ].strip()       #delete leading and trailing whitespaces

print( text )
[ 'I write something', 'I write something else', 'I append' ]
```

File operations: truncation

If you want to read and write *without truncation* use the mode 'r+'

```
f = open( 'myfile.txt', 'r+' )          # content will not be deleted
```

If you want to read and write *with truncation* use the mode 'w+'

```
f = open( 'myfile.txt', 'w+' )          # content will be deleted
```


File operation modes

Mode	Meaning
'r'	open for reading (default)
'w'	open for writing, truncating the file first
'x'	open for exclusive creation, failing when the file already exists
'a'	open for writing, appending to the end of the file if it exists
't'	text mode (default)
'b'	binary mode
'+'	open a disk file for updating (reading and writing)

The default mode is 'rt'
'w+t' or 'w+b' opens and truncates the file
'r+t' or 'r+b' opens the file without truncation

Make a file copy going by character

```
def copy_file(oldFile, newFile):
    f1 = open( oldFile, 'r' )
    f2 = open( newFile, 'w' )
    text = f1.read( 1 )
    while text != '':
            f2.write(text)
            text = f1.read( 1 )
    f2.close()
    f1.close()

copy_file( 'myfile.txt', 'mynewfile.txt' )
```

Note: Python module shutils offers several high-level functions to operate on files. Including copying files.

Make a file copy working by lines

```
def copy_file_byline( oldFile, newFile ):
    f1 = open( oldFile, 'r' )
    f2 = open( newFile, 'w' )
    for line in f1:
        f2.write( line )
    f2.close()
    f1.close()

copy_file_byline( 'myfile.txt', 'mynewfile.txt' )
```

Make a file copy in one go

```
def greedy_copy_file(oldFile, newFile):
    f1 = open( oldFile, 'r' )
    f2 = open( newFile, 'w' )
    text = f1.read()
    if not text == '':
        f2.write( text )
    f2.close()
    f1.close()

greedy_copy_file( 'myfile.txt', 'mynewfile.txt')
```

Obtaining a file through a url

```
import urllib.request as r
                                        % urllib allows us to work with internet URLs
f = r.urlopen('http://www.textfiles.com/adventure/impossiblemission2.sol')
f.read()
b'\n          "IMPOSSIBLE MISSION II" - HINTS, TIPS, SOLUTION (?)\n          By
Jaromir Krol - "Jerry King" - jaromirk@kki.net.pl\n
http://www.kki.net.pl/jaromirk/c64joke/\n\n\n Elvin  Atombender,  the  evil genius, is  at
it  again. What? Destroying the\n world, of course. And it\'s up to you to stop him.
(Actually, the storyline in\n the  manual is a nice read, so take a look at it at Project
64!). All right -\n Elvin  lives  in a complex of nine towers.  You must search the eight
of them\n one  by  one, and finally get access  to  Elvin\'s control room, hidden in the\n
last  tower…
```

Detection and count of characters by dictionary

```
def count_char(filePath):
    f = open(filePath)
    story = f.read().upper(). replace('\n', '').replace('\t', '').replace(' ', '')
    f.close()
    chars = { }                # dictionary in which (key:value) will be (character:count)
    for c in story:
        if c in chars:
            chars[c] = chars[c] + 1
        else:
            chars[c] = 1
     for c in chars:
        print (c, chars[c])
```


Reading a data *.csv* file

The *csv* module implements classes to read and write tabular data in CSV format. This is a very simple yet useful format for data science and data exchange.

The csv module's *reader* and *writer* objects read and write sequences of data

DictReader and *DictWriter* classes are used to convert data from .csv to dictionaries, and vice versa

Pragmatics: look at example code that uses *Dict*… at trustworthy sites on the internet

Reading a data *.csv* file

```
import csv
f = open( 'GPS.csv' )
reader = csv.reader( f )
for line in reader:
    print( line )
f.close()
```

```
['$GPRMC', '140053.00', 'A', '4454.1740', 'N', '09325.0143', 'W', '000.0', '128.7', '300508',
'001.1', 'E', 'A*2E']
['$GPGGA', '140053.00', '4454.1740', 'N', '09325.0143', 'W', '1', '09', '01.1', '00289.8', 'M', '-
030.7', 'M', '', '*5E']
['$GPGSA', 'A', '3', '21', '15', '18', '24', '26', '29', '06', '22', '', '03', '', '', '02.0', '01.1',
'01.7*04']
```


Other delimiters in a *.csv* file

Though they are called comma-separated, .csv files may use a different delimiter such as tab or ; or |

```
f = open( 'GPS.csv' )
reader = csv.reader( f )                                    # the default delimiter is ','

f = open( 'GPSSemicol.csv' )
reader = csv.reader( f, delimiter = ';' )

f = open( 'GPSTab.csv' )
reader = csv.reader( f, delimiter = '\t')
```

Dictreader *.csv* operation

Reading the contents of a .csv file into a Python dictionary:

```
import csv
f = open( 'GPSDict.csv' )
reader = csv.DictReader(f, fieldnames=['a','b','c','d','e','f','g','h','I','j','k','l','m'])
for line in reader:
    print (line)
f.close()
```

```
{'b': '140053.00', 'h': '000.0', 'c': 'A', 'a': '$GPRMC', 'f': '09325.0143', 'k': '001.1', 'g': 'W',
'l': '128.7', 'm': 'A*2E', 'd': '4454.1740', 'l': 'E', 'e': 'N', 'j': '300508'}
{'b': '140054.00', 'h': '000.0', 'c': 'A', 'a': '$GPRMC', 'f': '09325.0143', 'k': '001.1', 'g': 'W',
'l': '128.7', 'm': 'A*29', 'd': '4454.1740', 'l': 'E', 'e': 'N', 'j': '300508'}
{'b': '140055.00', 'h': '000.0', 'c': 'A', 'a': '$GPRMC', 'f': '09325.0143', 'k': '001.1', 'g': 'W',
'l': '128.7', 'm': 'A*28', 'd': '4454.1740', 'l': 'E', 'e': 'N', 'j': '300508'}
```


Geospatial text files

- Single point vector data files can be shared in *.csv* format, and then often having long and lat, or x and y, columns.
- Geospatial vector data is sometimes also shared in text files, especially various variants of *.xml* such as *.kml*, *.gpx* and others. Where such files tend to grow big they are often zipped up, e.g., *.kml* then becomes *.kmz*
- Python offers modules/classes for handling general *.xml*, but specific functions also exist for *.kml,* et cetera.
- For kml support, consider a packages like ogr and fastkml

Binary files

You can write and read files in binary mode using 'wb' and 'rb'. But this is out of scope of this lecture

If you use binary mode when you read (or write) a file, things won't be much different. You are still able to read a number of bytes (basically much the same as characters), and perform other operations associated with text files

Geospatial vector and raster data is shared in binary files also. All image data is principally in binary files; a common vector data format that is binary is the (Esri) shapefile. A shapefile is actually not one file, but a package of four or five files.

We will later discuss how to work with these file types.

Working with shapefiles

Various Python packages to read/write shapefiles: ogr, fiona, pyshp

Example code snippet:

```
from osgeo import ogr
shfile = ogr.Open("rivers.shp")
shape = shfile.GetLayer(0)                    # first feature layer of the shapefile
feature = shape.GetFeature(0)                 # first feature in that layer
…
```

Summary

- Files and directories
- Opening text files: reading, writing and append text
- Reading online text
- Reading CSV files