15-110 Principles of Computing – F19

Lecture 23:
Files I/O 2

Teacher:
Gianni A. Di Caro

Files: custom data structures for holding permanent data

- **File**: a *sequence* of data held in a *storage medium,* that can be either *volatile or not*
  - User data are written on the file according to a *custom organization* that reflects user's needs
  - **Data structure:** a collection of data elements organized in some way (e.g., list, dictionary, set)

A file is a custom, *permanent* data structure to hold data

Data storage and files

Data storage: storing data in a named location (a 'known' place, a ***file***) that can be accessed (open) later on for read / write / update operations, and can put aside (closed) when done, and it can also be removed if stored data are not anymore needed

Create a named file

✓ **Open** a *new* file

✓ **Open** an *existing* file

✓ **Read** *existing* data

✓ **Write** *new* data

✓ **Modify / write** *existing* data

Temporary shutdown an existing file

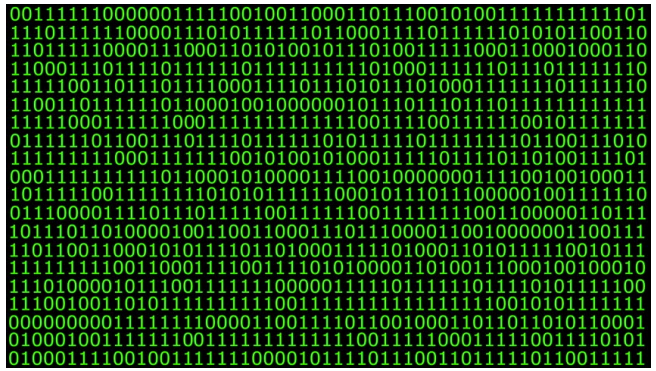
✓ **Close** *existing* file

Delete an existing file from the system

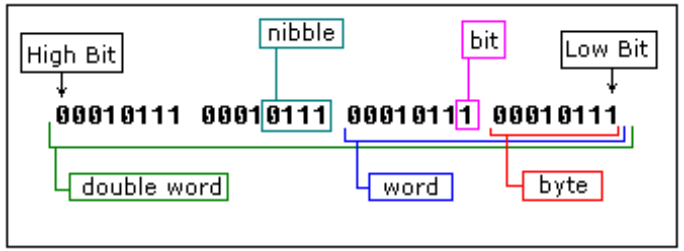
✓ **Remove** an *existing* file

Files are sequences of bytes

- A file on a computer storage system is a long sequence of 0 and 1 (bits) representing the binary encoding of the file contents
- Each sub-sequence of 8 bits identifies a byte, which is the main unit for writing / reading information
- → All files are sequence of bytes



- Each byte or group of bytes is **interpreted** depending on the OS, on the file type, on the chosen encoding:
  - E.g., a byte with the bit pattern 00101001 (97) can encode the ASCII character ‘a’
  - E.g., 2 consecutive bytes with bit pattern 00000111 11100011 can represent the integer number 2019



Text and Binary Files

Files can be broadly classified as:

- **Binary**
  - *Images*: jpg, png, gif, bmp, tiff, psd, ...
  - *Videos*: mp4, mkv, avi, mov, mpg, vob, ...
  - *Audio:* mp3, aac, wav, flac, mka, wma, ...
  - *Documents*: pdf, doc, xls, ppt, docx, odt, ...
  - *Archive*: zip, rar, 7z, tar, iso, ...
  - *Database*: mdb, accde, frm, sqlite, ...
  - *Executable*: exe, dll, so, class, ...
- **Text**
  - *Web tools*: html, xml, css, svg, json, ...
  - *Source code*: c, cpp, h, cs, js, py, java, php, sh, ...
  - *Documents*: txt, tex, markdown, asciidoc, rtf, ps, ...
  - *Configuration*: ini, cfg, rc, reg, ...
  - *Tabular data*: csv, tsv, ...

Text files: records and fields

- **Text:**
  - Human-readable: mostly composed of *printable* characters
  - Can be read / write using any text editor / viewer program
  - Organized in **multiple records** separated by **newline characters**
  - Each record is a piece of information, possibly structured in multiple **fields**

- Six records
- Four fields (at most):
  First name, Last name, ID, Sex

```
John Smith    642876 M
Adam Smith    787294 M
Ann White.    889220  F
Joan Black     627291  F
Mary Brown   78979
```

- Three records
- Variable number of fields per record

```
Old pond
Frog jumps in
Sound of water
```

- Four records
- Nine fields (at most), specified in the first record

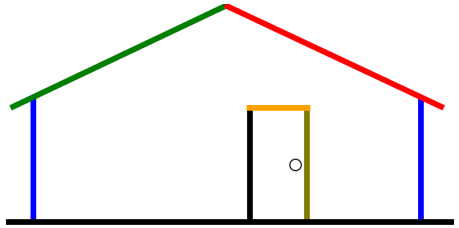
```
STATION,STATION_NAME,ELEVATION,LATITUDE,LONGITUDE,DATE,HPCP,Measurement Flag,Quality Flag
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100101 00:00,99999,],
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100101 01:00,0,g,
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100102 06:00,1, ,
```

Text files: records and fields

- Four records
- Variable number of fields per record

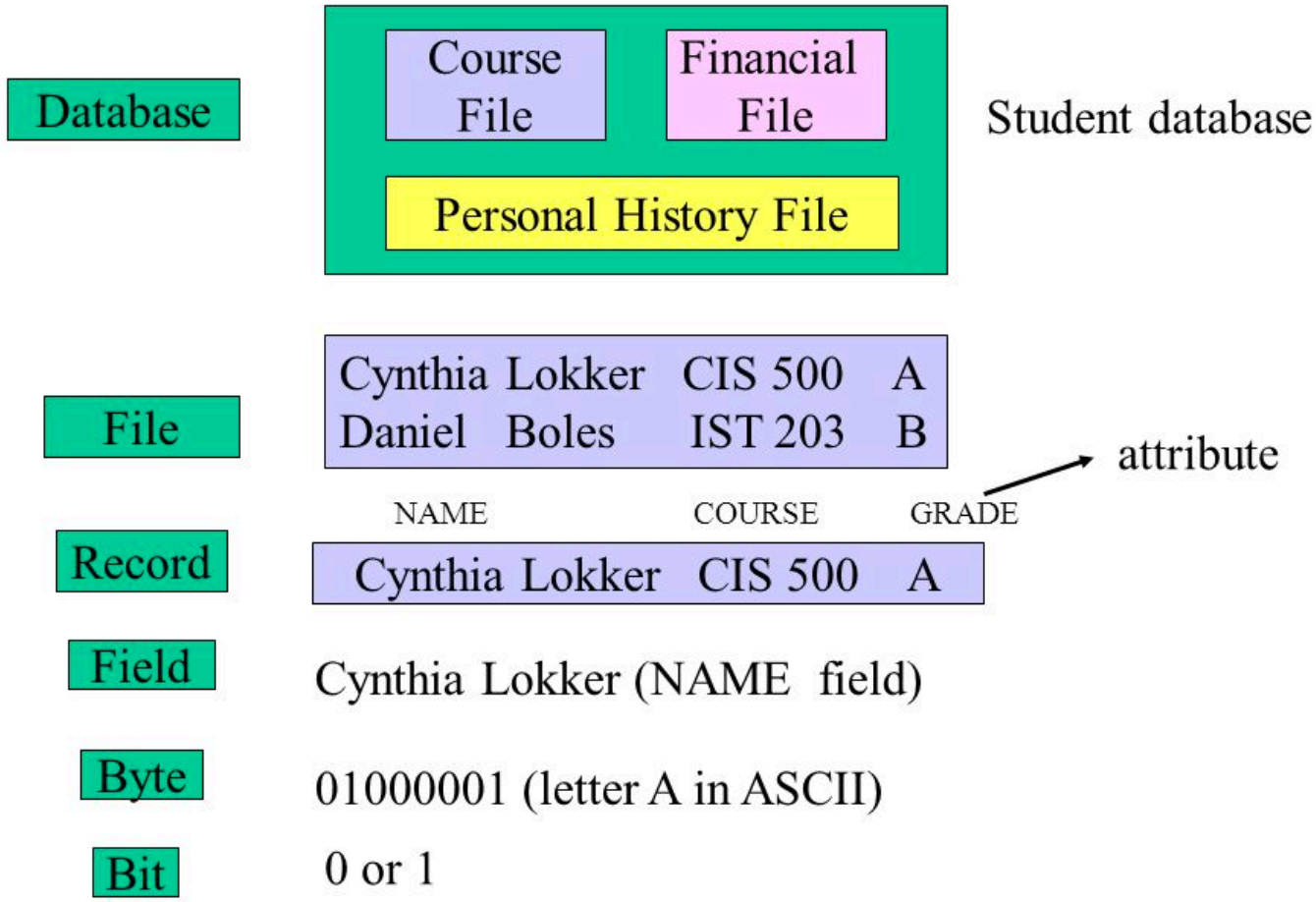
```
IOConsoleUsers: time(0) 0->0, lin 1, llk 0,
IOConsoleUsers: gIOScreenLockState 1, hs 2, bs 0, now 0, sm 0x0
loginwindow is not entitled for IOHIDLibUserClient keyboard accessIOConsoleUsers: time(0) 0->0, lin 1, llk 0,
IOConsoleUsers: gIOScreenLockState 1, hs 2, bs 0, now 0, sm 0x0
```

```
<svg xmlns="http://www.w3.org/2000/svg">
<!-- door -->
<line x1="220" y1="100" x2="220" y2="200" stroke-width="5" stroke="black" />
<line x1="270" y1="100" x2="270" y2="200" stroke-width="5" stroke="olive" />
<line x1="217" y1="100" x2="273" y2="100" stroke-width="5" stroke="orange" />
<circle cx="260" cy="150" r="5" stroke-width="1" stroke="black" fill="none"/>
<!-- walls -->
<line x1="30" y1="92" x2="30" y2="200" stroke-width="5" stroke="blue" />
<line x1="370" y1="92" x2="370" y2="200" stroke-width="5" stroke="blue" />
<!-- ground -->
<line x1="3" y1="200" x2="400" y2="200" stroke-width="5" stroke="black" />
<!-- roof line -->
<line x1="10" y1="100" x2="200" y2="10" stroke-width="5" stroke="green" />
<line x1="198" y1="10" x2="390" y2="100" stroke-width="5" stroke="red" />
</svg>
```



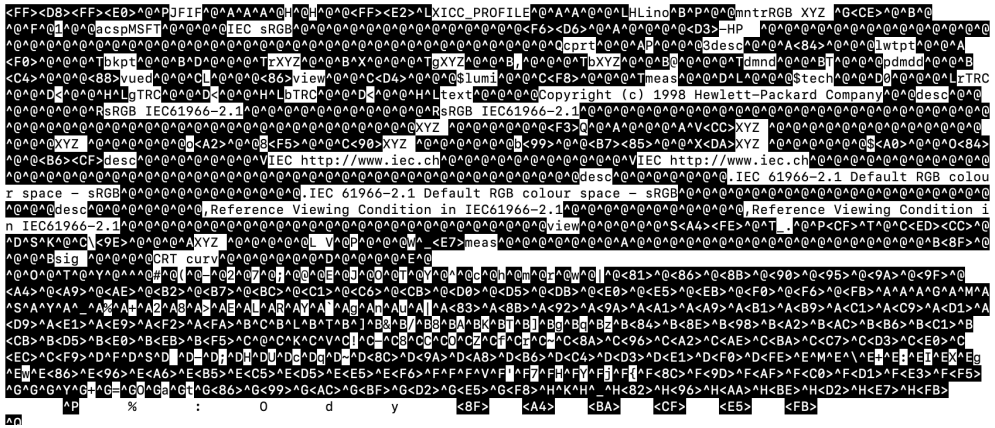
- Fifthteen records
- Variable number of fields per record
- Presence of markers for field interpretation

Text files: Data hierarchy



Binary Files

- **Binary:**
  - Non human-readable: mostly composed of *non-printable characters*
  - No general structure, it needs a specific program that is aware of the *file format* in order to read / write it consistently
  - The program knows how to interpret the individual bytes in the files
  - File extensions precisely help to identify the program that can deal with a specific binary file format (e.g., MS Word can't read a zip file)



What we see when trying to read a jpeg file with a text editor!

Open a file (and let's focus on text files): `open( )` function

- ✓ While each OS has its own way to deal with file (**file system** of an OS) , Pyhton achieves OS-independence by accessing a file through a **file handle** that holds the reference to the file in the system

- ➢ **Open** a file to make I/O on it, using the function:

```
file_handle = open(file_name, <modes>)
```

`TextIOWrapper` object type (**stream**)

`str` object type

`str` object type

```
f = open('data.txt', 'rt')
f = open('data.txt', 'r')
f = open('data.txt', 'w')
f = open('data.txt', 'r+')
f = open('data.txt', 'w+')
```

Open a file (and let's focus on text files): `open( )` function

`modes:`

1. Read/Write
2. Text/Binary

`'r'`	open an existing file for reading (default)
`'w'`	open for writing, *truncating* file to 0 bytes if it exists, or creating a new file otherwise
`'x'`	create a new file and open it for writing
`'a'`	open for writing, *appending* to end of file if it exists, or creating a new file otherwise
`'+'`	open a file for updating (both reading and writing)
`'b'`	binary mode
`'t'`	text mode (default)

Default: `'rt'`

Read from a file: `read()` method

- **Read** data from a file open with the `r` or `r+` mode flag (or, also, `w+`, `a+`)

```
string_with_data = file_handle.read(number_of_bytes)
```

- The method **read()** reads *at most* `number_of_bytes` bytes from the file, from current position in file
- One byte ~ One character (depends on the encoding)
- The entire file is read when called without the `number_of_bytes` parameter, i.e., `read()`
- If the read hits End-Of-File (EOF) before reading the required `number_of_bytes` bytes, the function returns and reads only the available bytes
- Read data are returned as a string in the variable `string_with_data`
- If the function is called at the EOF, read() returns an empty string ('')

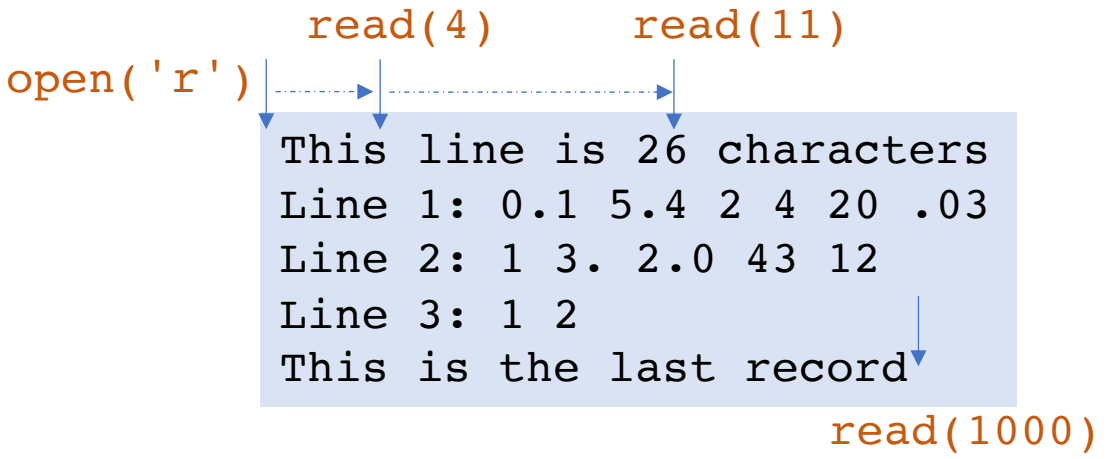
Read from a file: `read()` method

```
f = open('data.txt', 'r')
data = f.read(4)   #data holds: 'This'
data = f.read(11)  #data holds: ' line is 26'
data = f.read(1000)   #data holds: from ' char' to the end
data = f.read()   #data holds: ' '
```

File must be in the same folder of the calling program!

```
new_handler = open('data.txt', 'r')
data = f.read()   # data holds the entire file
```

data.txt



The *reading head* of each file handler *moves* forward from the current/last position

File read and use of r/w/a/+ mode flags in open( )

```
f = open('data.txt', 'w')
data = f.read(4)   # Error! File is open only for writing!
```

```
f = open('data.txt', 'a')
data = f.read()   # Error! File is open only for writing (appending at the end)!
```

```
f = open('data.txt', 'rw')      # Error! Exactly only one read/write/append mode is allowed
```

```
f = open('data.txt', 'r+')
data = f.read()   # Ok! The file is in principle also available for writing
```

```
f = open('data.txt', 'w+')
data = f.read()   # Ok BUT the file is truncated to 0 bytes, data contains nothing!
```

```
f = open('data.txt', 'a+')
data = f.read()   # Ok BUT the file is truncated to 0 bytes, data contains nothing!
```

Read a file in text and binary mode

```
f = open('data.txt', 'rt')
data = f.read()    #data holds the entire file, formatted
print(data)
```

```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```


```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```


```
f = open('data.txt', 'rb')
data = f.read()    #data holds the entire file, unformatted (for printing), all special characters are there
print(data)
```

```
b'This line is 26 characters\r\nLine 1: 0.1 5.4 2 4 20 .03\r\nLine 2: 1 3. 2.0 43 12 \r\nLine 3: 1 2 \r\nThis is the last record'
```

Read a file in text and binary mode

```
f = open('cmu_logo.png', 'rt')
data = f.read()
# Error! Reading in text mode doesn't allow to properly decode some special characters.
```

cmu_logo.png
(78 KB)

```
f = open('cmu_logo.png', 'rb')
data = f.read()   # Ok! Data holds the entire binary representation of the png image
```

```
from IPython.display import display, Image
display(Image(data))   # Display the image data, function Image() knows how to deal with binary .png format
```