15-110 Principles of Computing – S19

Lecture 15:
Files I/O 1

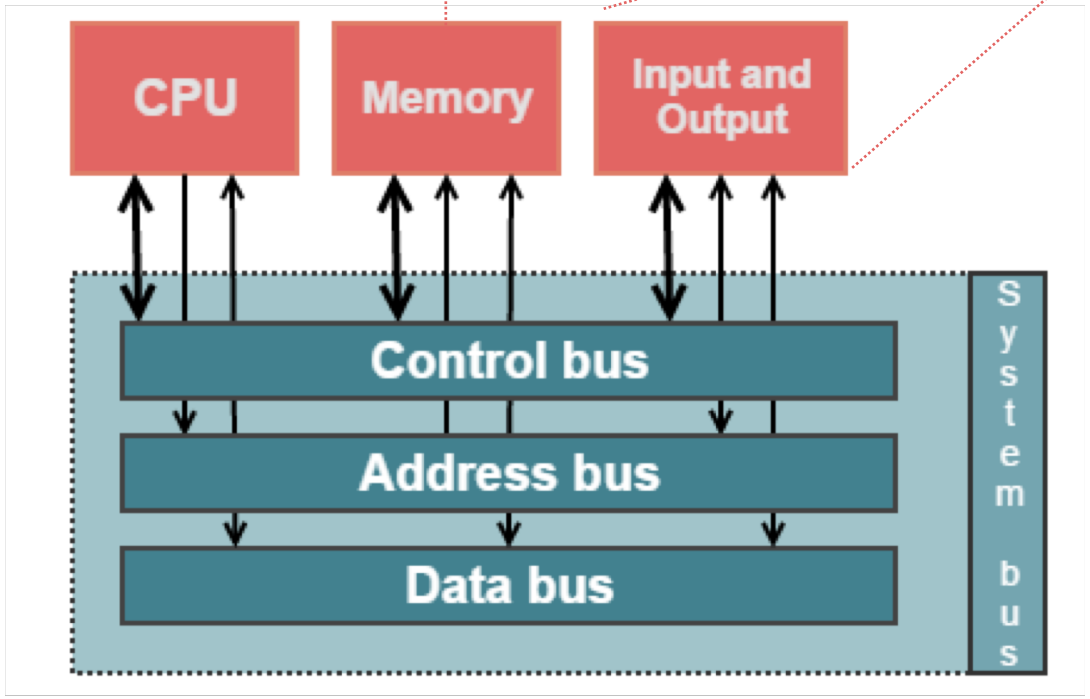
Teacher:
Gianni A. Di Caro

جامعة كارنيجي ميلون في قطر
Carnegie Mellon University Qatar

Files and computer peripherals

- **File**: a *sequence* of data that is **stored** in some **physical support** either permanently or temporarily

Internal memory
RAM: volatile
ROM: permanent



System bus architecture

Peripheral devices: parts of a computer system that are not parts of the core computer architecture, they are *optional,* can serve to *expand* the capabilities of the system

I/O storage peripherals:

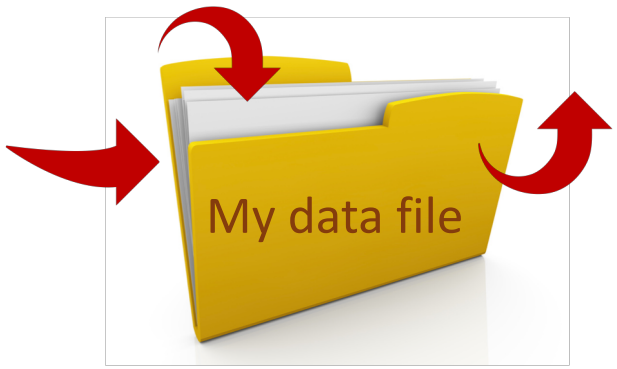
- ✓ Magnetic storage media
- ✓ Optical storage media
- ✓ Solid state storage media

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

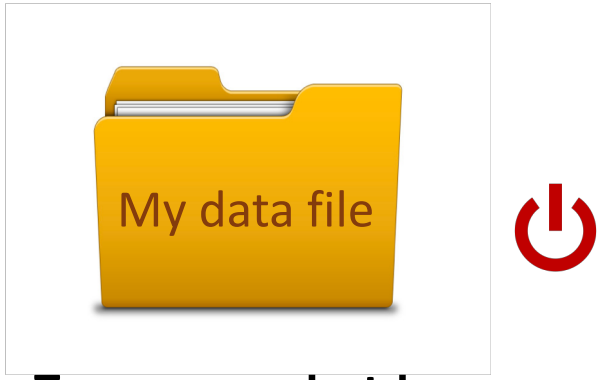


✓ **Open** an *existing* file

✓ **Read** *existing* data

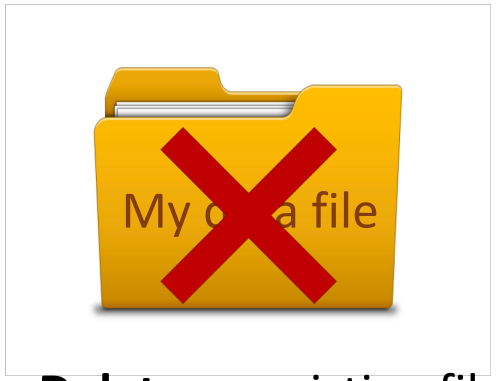
✓ **Write** *new* data

✓ **Modify / write** *existing* data



Temporary shutdown an existing file

✓ **Close** *existing* file



Delete an existing file from the system

✓ **Remove** an *existing* file

Modalities for data storage

Data can be stored:

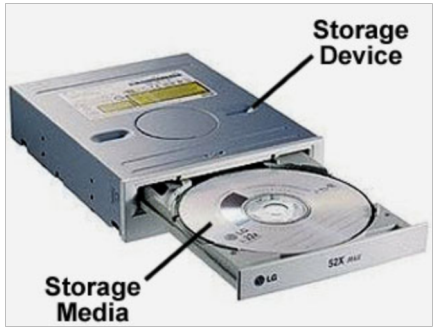
- **Temporarily** → **RAM**
  - Volatile: data is erased when computer is switched off
  - Use to store data currently in use by running programs
  - Very fast I/O operations

- **Permanently** → Backing storage device
  - Non volatile: data stays stored after powering off the computer / device
  - Use to store long-term data
  - Slower I/O operations compared to RAM

Examples: USB flash drive, solid state drive, hard drive, DVD drive, ROM

Backing storage devices

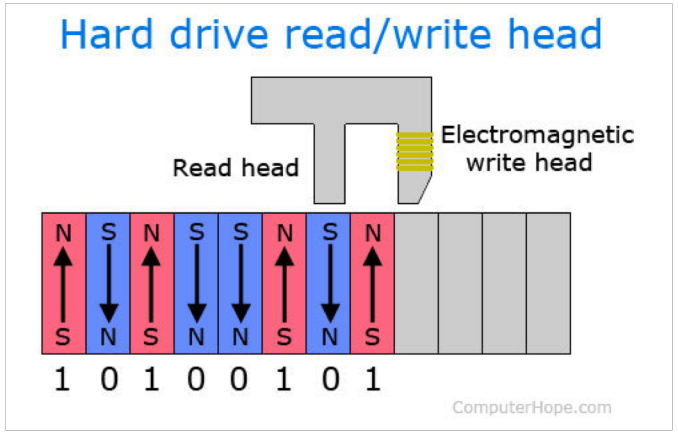
- A backing storage device consists of:
  - ✓ **Storage media:** the parts / devices that physically hold the data
  - ✓ **Storage device:** the machine / device that reads/writes the data from/to the storage media



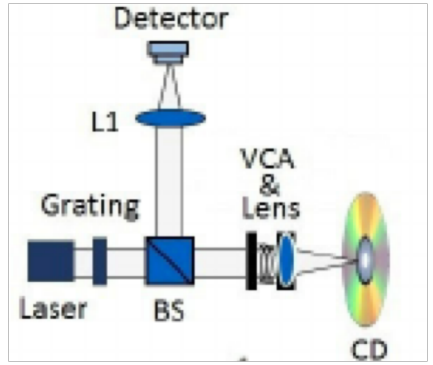
- A backing storage device can be internal or external to the computer system

Common storage media

- **Magnetic media** (Hard disk drives, Tapes)
  - Data are written by magnetic polarization. The electromagnetic write head polarizes tiny sections of the ferromagnetic hard drive so they face up or down ("North" or "South") to represent the binary digits 1 or 0 that encode information. The read head can detect the magnetic polarization and then associate 0 or 1 to each section. R/W head *moves*



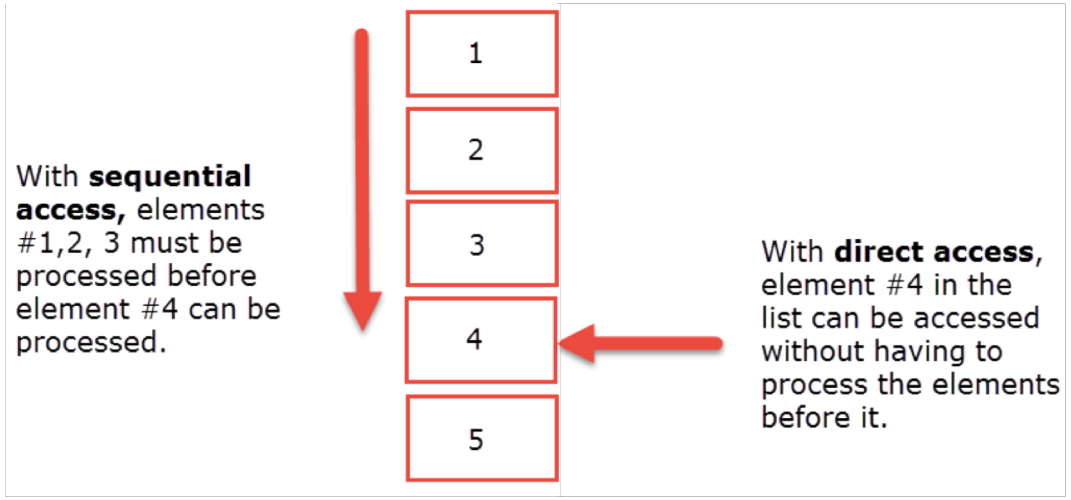
- **Optical media** (CD, DVD, Blue-ray)
  - Data are written by by *burning tiny dots* on the surface of an optically sensitive medium (e.g., a *rotating disc*) by using a high powered laser. The presence or not of a dot means either 1 or 0. Data is read by shining a low powered laser light over the dots. The presence or not of reflection light is associated to the presence or not of a burned dot.



- **Solid state media** (SSD / Flash memory: SS Hard drives, Memory sticks / Pen drives, Memory cards)
  - Data are stored electronically, using properties of silicon microchips. *No mechanically moving parts*

Accessing the data

- **Sequential access** (Magnetic data tapes)
  - Data is accessed by starting from the beginning and then moving sequentially, forward and backward, over the medium
  - Slow, very slow, but reliable, cheap, and still in used for backing up very large amount of data



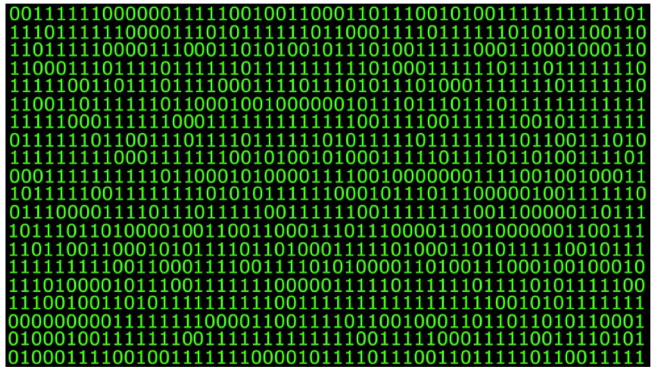
- **Direct access** (All ”modern” media)
  - Information about location of data (e.g., where a file is in the medium) is maintained in a dedicate space on the medium, such as when requested, data can be accessed directly, by “jumping” to the right place rather than moving to the place sequentially from the beginning, as in the case of sequential access
  - Fast, much faster than sequential, especially when there are no moving parts

Files: custom data structures for holding permanent data

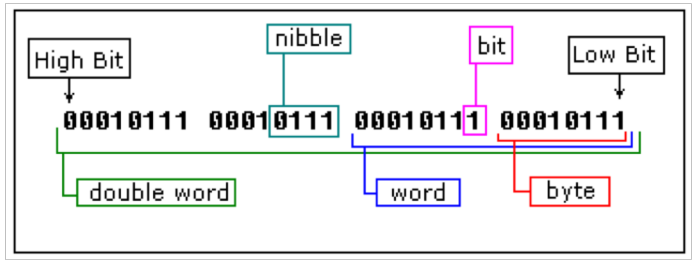
- **File**: a *sequence* of data held in a *storage medium,* that can be either *volatile or not*
  - User data are written on the file according to a *custom organization* that reflects user's needs
  - **Data structure:** a collection of data elements organized in some way (e.g., list, dictionary, set)
  - → A file is a custom, permanent data structure to hold data
  - It can be used to hold and represent virtually anything ….
    - Image, in different formats, such as jpeg, png, svg, gif …
    - Data of interest: Genome maps, financial data, climate data, traffic logs, medical data….
    - Log about a running program, such as a Web server, or your own python program …
    - Data collected by sensors during experiments
    - Text of a novel, a poem, a song, …
    - Specification of a web page, in HTML, CSS, JavaScript, …
    - Notes about … anything!
    - ….

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**

- Five 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

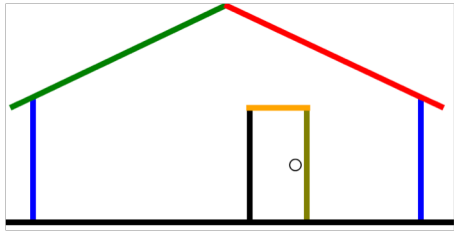
```
STATION,STATION_NAME,ELEVATION,LATITUDE,LONGITUDE,DATE,HPCP,Measurement Flag,Quality Flag
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100101 00:00,99999,],
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100101 01:00,0,g,
COOP:310301,ASHEVILLE NC US,682.1,35.5954,-82.5568,20100102 06:00,1,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

Text files: records and fields

- Four records
- Variable number of fields per record

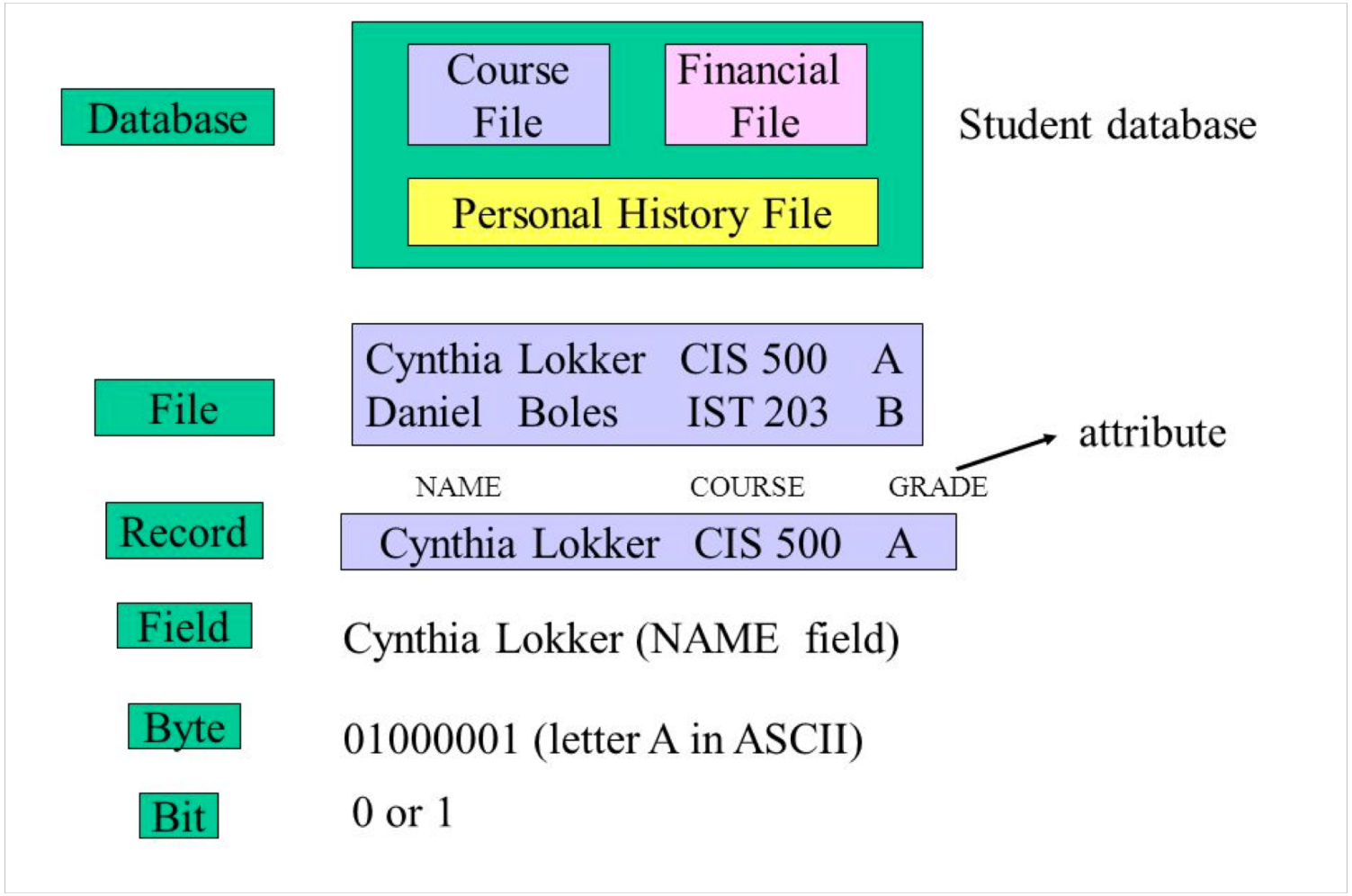
```
IOConsoleUsers: time(0) 0->0, lin 1, llk 0,
IOConsoleUsers: gIOScreenLockState 1, hs 2, bs 0, now 0, sm 0x0
loginwindow is not entitled for IOHIDLibUserClient keyboard accessIOConsoleUsers: time(0) 0->0, lin 1, llk 0,
IOConsoleUsers: gIOScreenLockState 1, hs 2, bs 0, now 0, sm 0x0
```

```
<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>
```



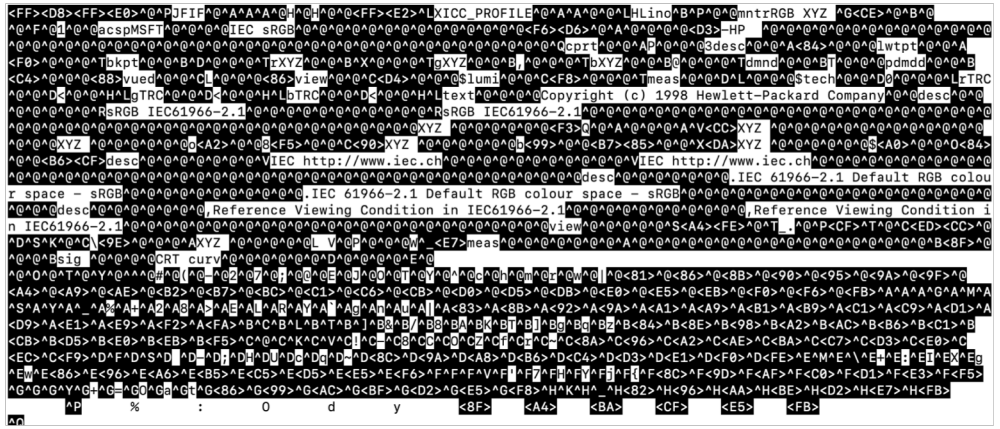
- Fifthteen records
- Variable number of fields per record
- Presence of markers for field interpretation

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+')
```

`modes:`

1. Read/Write
2. Text/Binary

Default: `'rt'`

`'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)

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 ('')

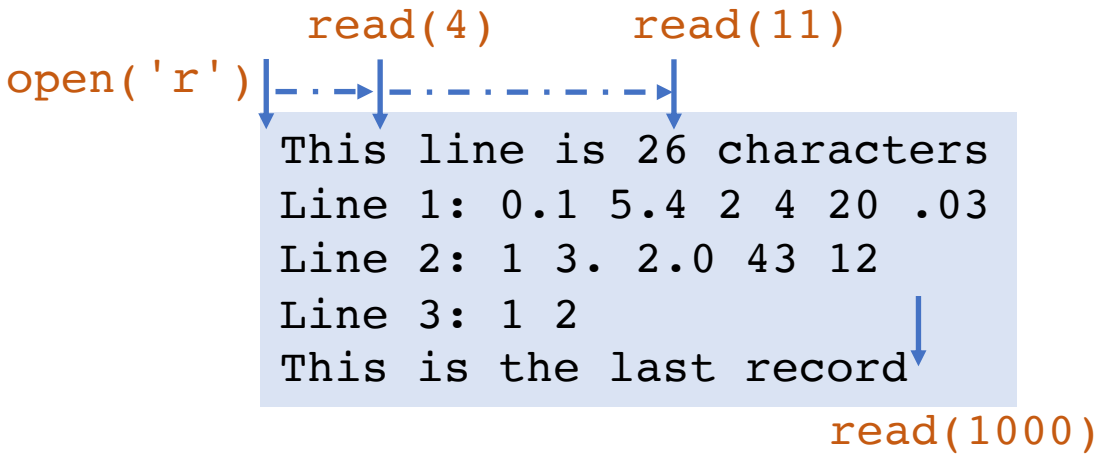
Read from a file: `read()` method

```
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
```

Moving the reading head: `seek()` and `tell()` methods

- Get the current position (in bytes) in the file from the beginning (position 0):

```
position = file_handle.tell()
```

```
f = open('data.txt', 'r')
data = f.read(11)
pos = f.tell()          # pos has value 11
data = f.read(19)
pos = f.tell()          # pos has now value 31
```

- Go to the given position (in bytes) in the file from the beginning (position 0):

```
position = file_handle.seek(pos, <from_where>)
```

```
f = open('data.txt', 'r')
pos = f.seek(30)          # pos has value 30
data = f.read(10)
pos = f.tell()          # pos has now value 40
```

Only for binary files:

- `from_where`'s default value is 0, meaning from the beginning
- `from_where` = 1 means relative to current positions
- `from_where` = 2 means relative to end

Read a file record-by-record using a for loop

- A text file is organized in **records / lines** separated by newlines: it is possible to iterate over all records/lines, that are string types
- Individual **fields** inside a record need to be extracted based on the knowledge of the structure of the 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', 'r')
for record in f:
    print(record, end = '')    # the end option removes the default newline from the record string
```

```
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', 'r')
for record in f:
    print(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
```

Read individual records / lines in a file: `readline()` method

- **Individual records / lines** can be read by invoking the `readline()` method
- The function returns the read string (that includes a newline \n)
- If the function is called at the EOF it returns an empty string (' ')

```
f = open('data.txt', 'r')
bytes_so_far = 0
record = f.readline() # 'This line is 26 characters\n'
bytes_so_far += len(record)
record = f.readline() # 'Line 1: 0.1 5.4 2 4 20 .03\n'
bytes_so_far += len(record)
record = f.readline() # 'Line 2: 1 3. 2.0 43 12 \n'
bytes_so_far += len(record)
f.seek(bytes_so_far + 5)
record = f.readline() # ': 1 2\n'
record = f.readline() # 'This is the last record\n'
```

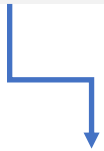
```
This line is 26 characters
Line 1: 0.1 5.4 2 4 20 .03
Line 2: 1 3. 2.0 43 12
Line 3: 1 2
This is the last record
```

Read all remaining records / lines in a file: `readlines()` method

- **All records / lines** in the file from the current position can be read using the `readlines()` method
- The method **returns a list of strings**, where each string is a consecutive line/ record of the file

```
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', 'r')
record = f.readline() # 'This line is 26 characters\n'
record = f.readline() # 'Line 1: 0.1 5.4 2 4 20 .03\n'
record = f.readlines()
```



```
record  is the list: [  'Line 2: 1 3. 2.0 43 12\n',
                        'Line 3: 1 2\n',
                        'This is the last record' ]
```

Write in a file: `write()` method

- To write something into a file, the open function must be invoked with one of the mode flags: `w, a, x, r+, w+, a+`
- Opening with `w` *erases* file's content if file exists, writing starts at the (new) beginning
- Opening with `a` lets writing start at the end of the file (appending)
- Opening with `r+` lets writing start at the beginning of the file (overwriting)
- If a file doesn't exist, `w`, `a`, `x` will create it
- The `+` versions allow both writing and reading

➢ **Write** data to a file open with a writing flag:

```
written_bytes = file_handle.write(string_to_write)
```

```
f = open('data.txt', 'a')
nbytes = f.write('New line: 0 3 5.5')
nbytes = f.write('Another new line: 1 2 3')
```

Write in a text file: newline characters and mode flags

```
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
```

There was no \n (newline) at the end of the last record

```
f = open('data.txt', 'a')
nbytes = f.write('New line: 0 3 5.5')
nbytes = f.write('Another new line: 1 2 3')
```

17 bytes

23 bytes

There is no \n (newline) at the end of the first newly appended 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 recordNew line: 0 3 5.5Another new line: 1 2 3
```

Write in a text file: newline characters and mode flags

```
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', 'w')
nbytes = f.write('New line: 0 3 5.5\n')
nbytes = f.write('Another new line: 1 2 3\n')
```

18 bytes

24 bytes

The w mode flag causes the *erase* of the existing file

There is a \n (newline) at the end of the two newly appended records

```
New line: 0 3 5.5
Another new line: 1 2 3
```

Write in a text file: newline characters and mode flags

```
f = open('data.txt', 'x')
```

`data.txt` exists in the file system,
the open returns with an error!

```
f = open('new_data.txt', 'x')
nbytes = f.write('New line: 0 3 5.5\n')
nbytes = f.write('Another new line: 1 2 3\n')
```

```
New line: 0 3 5.5
Another new line: 1 2 3
```

`new_data.txt` is now a new file in the file system

Inspecting the access mode of a file: `readable()`, `writable()`

- Check weather a file is open with **read mode flag** or not:

```
read_mode = file_handle.readable()
```

`True` is returned when file is readable, `False` otherwise

- Check weather a file is open with **write mode flag** or not:

```
write_mode = file_handle.writable()
```

`True` is returned when file is writable, `False` otherwise

```
f = open('numbers.txt', 'r')
if f.readable():
    data = f.readlines()
    print(data)
if f.writable():
    f.write('Add another record')
```

Closing a file after use: `close()` method

- **Close** a file when no further operations are needed / allowed:

```
file_handle.close()
```

- Closing a file frees up used file resources (and let the file accessible for deleting/renaming by the OS)
- If a close() isn't explicitly called, python's garbage collector does eventually the job of closing the file
- Explicitly closing the file prevents the program to perform any (unwanted) further operations on the file
- close() returns `None`

```
f = open('numbers.txt', 'w')
f.write('This file contains important data\n')
f.write('0 1 2 3 4\n')
f.write('4 3 2 1\n')
f.close()
```