COMP1730/COMP6730
Programming for Scientists

I/O and files

Outline

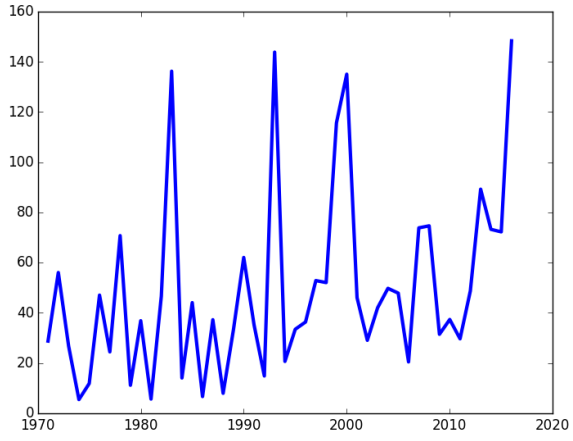
* Input and output
* Files and directories
* Reading and writing text files

Input and output

I/O: Input and Output

* A (common) way for a programs to interact with the world.
  - Reading data (keyboard, files, network).
  - Writing data (screen, files, network).
* Scientific computing often means processing or generating large volumes of data.

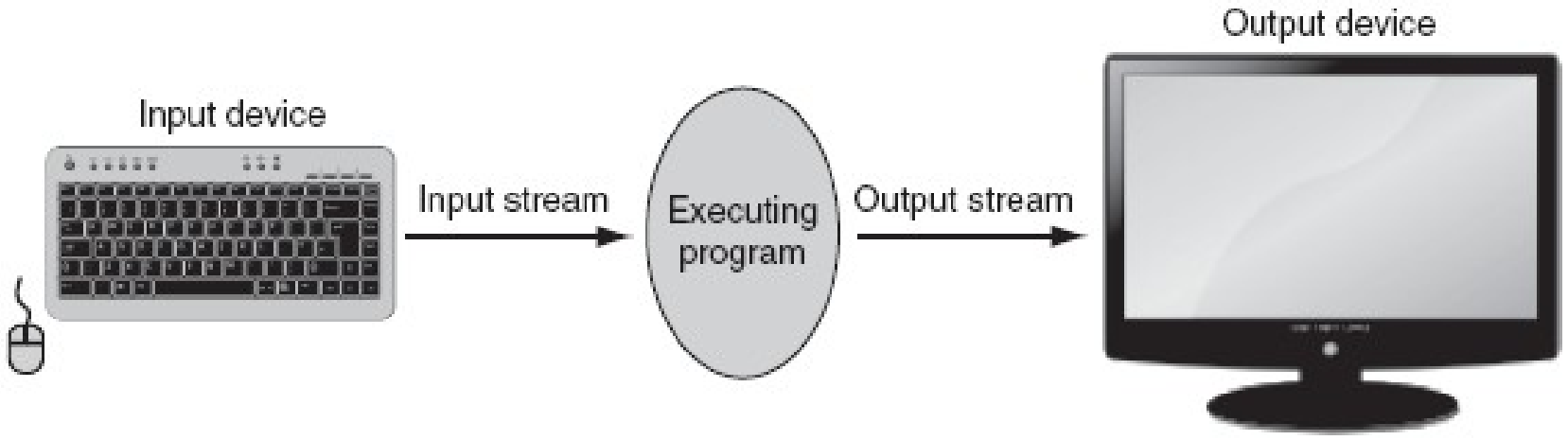
```
2016,  07,  01,   2.0,  1,  Y
2016,  07,  02,   0.0,  1,  Y
2016,  07,  03,   0.0,  1,  Y
2016,  07,  04,   0.0,   ,  Y
2016,  07,  05,   4.4,  1,  Y
2016,  07,  06,  15.4,  1,  Y
2016,  07,  07,   1.0,  1,  Y
2016,  07,  08,   0.0,  1,  Y
2016,  07,  09,   4.2,  1,  Y
2016,  07,  10,   0.0,  1,  Y
2016,  07,  11,  10.4,  1,  Y
```



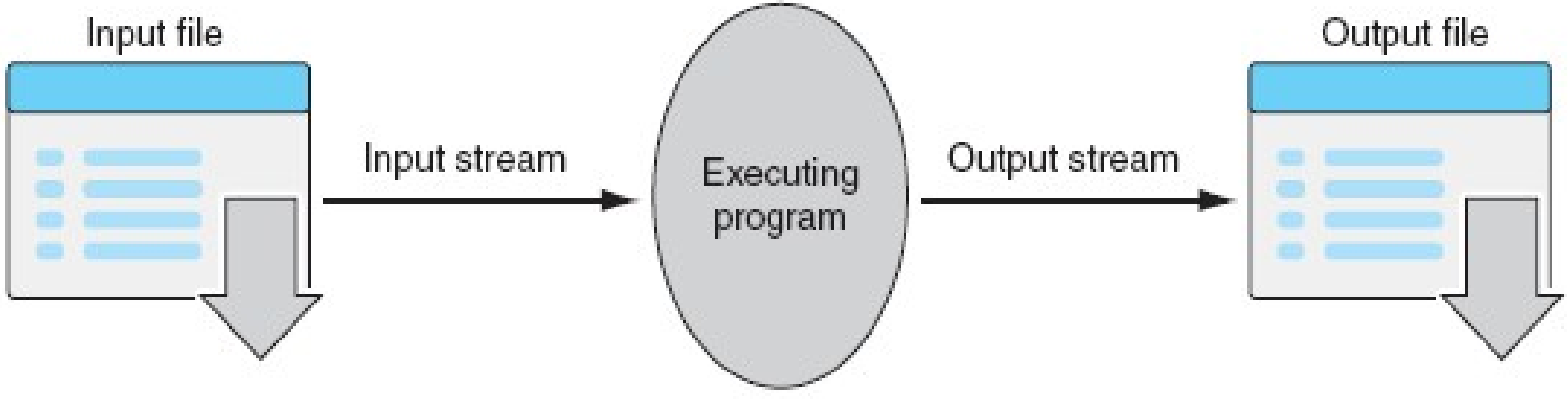
Terminal I/O

* `print(...)` generates output to the terminal (typically, screen).
* `input(...)` prints a prompt and reads input from the terminal (typically, keyboard).
  - `input` always returns a string.

```
input_str = input("Enter a number: ")
input_int = int(input_str)
...
```



a) Standard input and output



b) File input and output

FIGURE 5.1 Input-output streams.

Image from Punch & Enbody

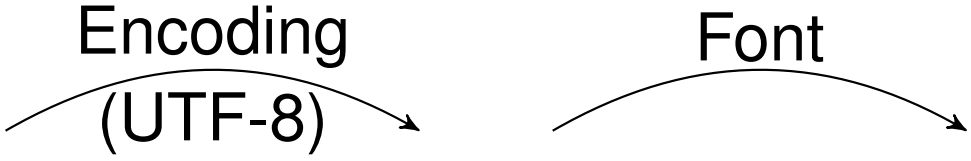
Files and directories

What is a file?

* A *file* is a collection of data on secondary storage (hard drive, USB key, network file server).
* A program can *open* a file to read/write data.
* Data in a file is a sequence of *bytes* (integer $0 \leq b \leq 255$).
  - The program reading a file must *interpret* the data (as text, image, sound, etc).
  - python & the operating system (OS) provide support for interpreting data as text.

Text encoding (recap)

* Every character has a number.
* *Unicode* defines numbers ("*code points*") for >120,000 characters (in a space for >1 million).

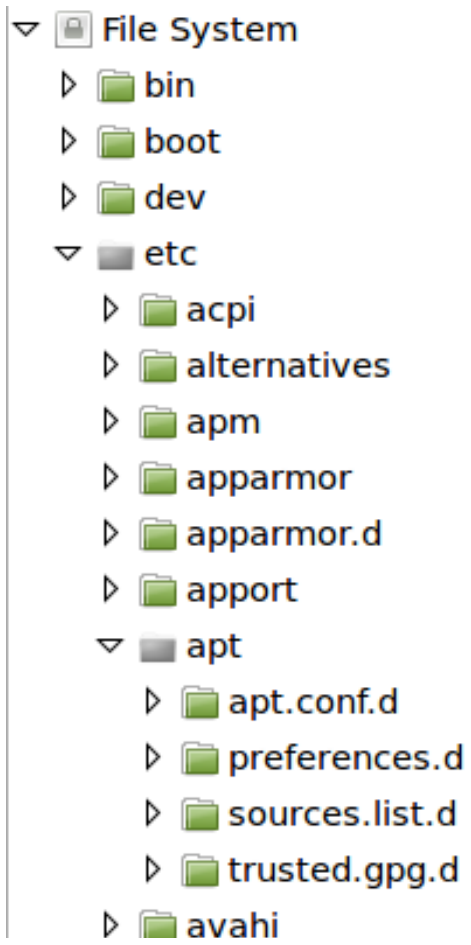


Byte(s)	Code point	Glyph
0100 0101 (69)	69	EEE$\mathcal{E}$
1110 0010 (226) 1000 0010 (130) 1010 1100 (172)	8364	€€€€

* A *text file* contains (encodings of) printable characters (including spaces, newlines, etc).
  - (python) program source code, HTML files, etc.
* A *binary file* contains arbitrary data, which may not correspond to printable characters.
  - images, audio/video, word documents.

Directory structure

* Files on secondary storage are organised into *directories* (a.k.a. *folders*).
* This is an abstraction provided by the operating system.
  - It will appear differently on different operating systems.
* The directory structure is typically tree-like.



File path

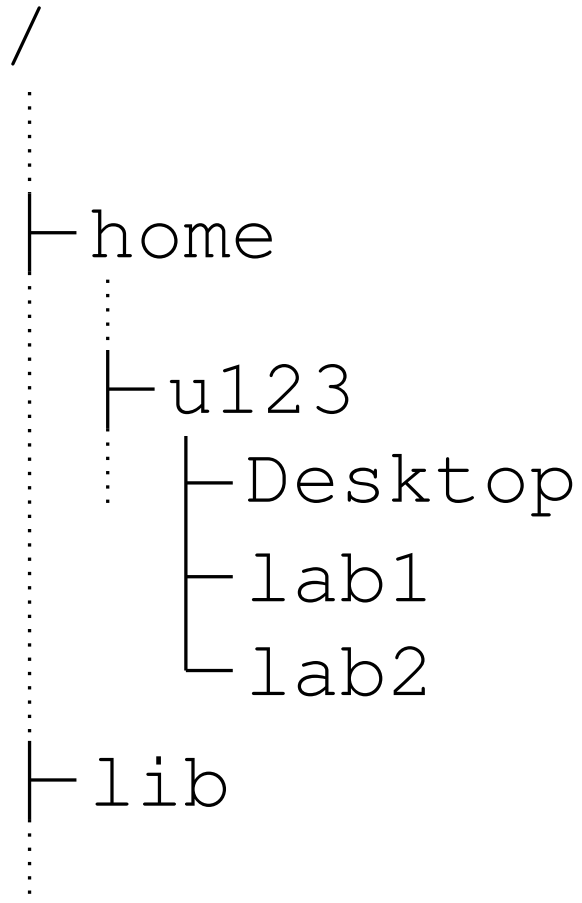
* A *path* is string that identifies the location of a file in the directory structure.
* Consists of directory names with a *separator* between each; the last name in the path is the name of the file.
* Two kinds of paths:
  - *Absolute*
  - *Relative* to the current working directory (cwd)

* When running a python file (script mode), the *current working directory* (cwd) is the directory where that file is.
* If the python interpreter was started in interactive mode (without running a file), the cwd is the directory that it was started from.
* The `os` module has functions to get (and change) the current working directory.

```
>>> import os
>>> os.getcwd()
'/home/patrik/teaching/python'
```

Example: Posix (Linux, OSX)

* Single directory tree.
  - Removable media and network file systems appear at certain places in the tree.
* The separator is `'/'`
* An absolute path starts with a `'/'`
* `'..'` means the directory above.
* File and directory names are *case sensitive*.



If the cwd is `/home/u123/lab1` then

`prob1.py` refers to `/home/u123/lab1/prob1.py`

`../lab2/prob1.py` refers to `/home/u123/lab2/prob1.py`

`../../../lib/libbz2.so` refers to `/lib/libbz2.so`

`/home/u123/Lab1/prob1.py` does not exist.

Example: Windows

* One directory tree for each “drive”; each drive is identified by a letter (`"A"` to `"Z"`)
* The separator is `'\'`
  - Must be written `'\\'` in python string literals.
* Absolute path starts with drive letter and `':'`
* `'..'` means the directory above.
* File and directory names are *not* case sensitive.

```
"C:\\Users\\patrik\\test.py"
"..\\lab1\\exercise1.py"
```

Reading and writing text files

File objects

* When we open a file, python creates a *file object* (or “stream” object).
  - The file object is our interface to the file: all reading, writing, etc, is done through methods of this object.
  - The type of file object (and what we can do with it) depends on the *access mode* specified when the file was opened.
  - For example, *text mode* vs. *binary mode*, *read-only*, *write-only*, *read-write* mode, etc.

Opening a file

* `open(file_path, access_mode)` opens a file and returns the file object.

```
my_file = open("notes.txt", "r")
first_line = my_file.readline()
second_line = my_file.readline()
my_file.close()
```

* Close the file when done!
* After calling `file_obj.close()`, we can do no more read/write calls on `file_obj`.

Access modes

* `access_mode` is a string, made up of *flags*.

		if the file exists...	if it does not exist...
`r`	read only		Error
`w`	write only	Erases file content	Creates a new (empty) file
`a`	write only	Appends new content at end of file	Creates a new (empty) file
`r+`	read/write	Reads/overwrites from beginning of file	Error
`w+`	read/write	Erases file content	Creates a new (empty) file
`a+`	read/write	Reads/overwrites starting at end of file	Creates a new (empty) file
`b`	Open as binary file (default is text)		

Caution

* Be careful with write modes. Erased or overwritten files cannot be recovered.
* Can we check if an existing file will be overwritten?

  Yes!
  - `os.path.exists(`*`file_path`*`)` returns `True` or `False`.
  - Catching exceptions (more later in the course).

Reading text files

* *file_obj*`.readline()` reads the next line of text and returns it as a string, *including* the newline character (`'\n'`).
* *file_obj*`.read(`*size*`)` reads at most *size* characters and returns them as a string.
  - If *size* `< 0`, reads to end of file.
* If already at end-of-file, `readline` and `read` return an empty string.
* *file_obj*`.readlines()` reads all remaining lines of text returning them as a list of strings.

File position

* A file is sequence of bytes.
  - But the file object is *not* a sequence type!
* The file object keeps track of where in the file to read (or write) next.
  - The next read operation (or iteration) starts from the current position.
* When a file is opened for reading (mode `'r'`), the starting position is 0 (beginning of the file).
* File position is *not* the line number.

* Suppose `"notes.txt"` contains:

```
First line
Second line
last line
```

* Then

```
>>> fo = open("notes.txt", "r")
>>> fo.read(4)
'Firs'
>>> fo.readline()
't line\n'
>>> fo.readlines()
['Second line\n', last line\n']
```

Iterating through a file

* pyton's text file objects are *iterable*.
* Iterating yields one line at time.

```
my_file = open("notes.txt", "r")
line_num = 1
for line in my_file:
      print(line_num, ':', line)
      line_num = line_num + 1
my_file.close()
```

Programming problem

* Read a python source code file, and
  - print each line;
  - prefix each line of *code* with a line number;
  - for numbering, count only lines containing code (not empty lines, or lines with only comments).

Writing text files

* Access mode `'w'` (or `'a'`) opens a file for writing (text).
* *`file_obj`*`.write(`*`string`*`)` writes the string to the file.
  - Note: `write` does not add a newline to the end of the string.
* `print(..., file=`*`file_obj`*`)` prints to the specified file instead of the terminal.

Buffering

* File objects typically have an I/O buffer.
  - Writing to the file object adds data to the buffer; when full, all data in it is written to the file ("flushing" the buffer).
* Closing the file flushes the buffer.
  - If the program stops without closing an output file, the file may end up incomplete.
* *Always close the file when done!*