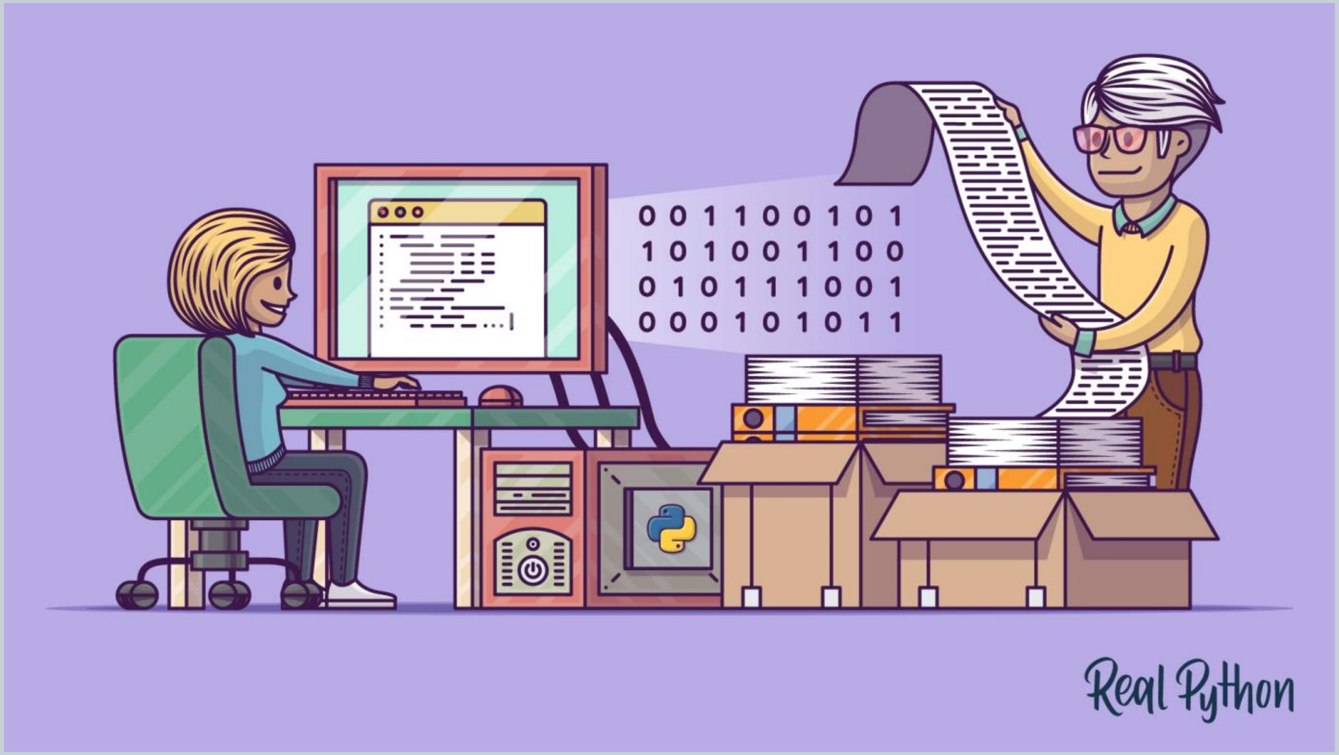
File Input/Output (I/O)

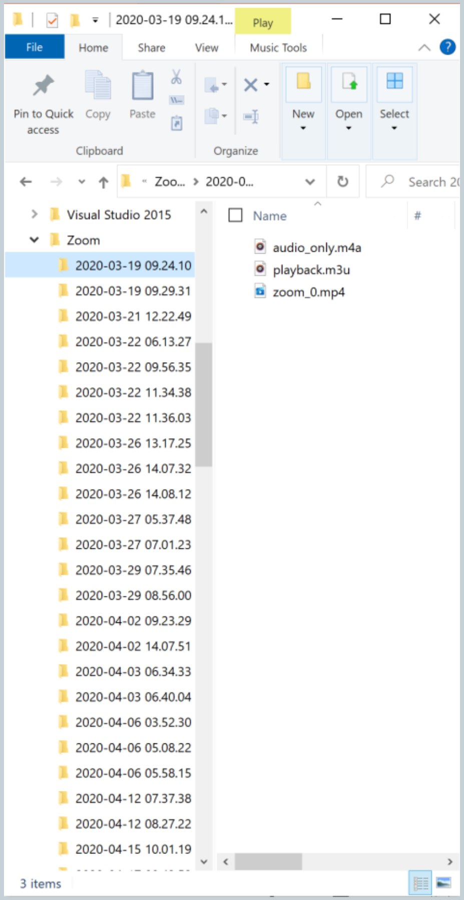


Outline

- What is a File
  - File Types
- Opening a File
- Access Modes
- Closing a File
- Reading Data from a File
- Writing Data into a File
  - Buffering
- File Pointer Positions
- Python File Methods

What is a File?

- Data stored in contiguous bytes
  - Usually on persistent media
    - Disk drive, cloud drive, USB drive, etc.
  - Examples:
    - A Python program
    - An electronic image
    - A Word document
    - A sound file

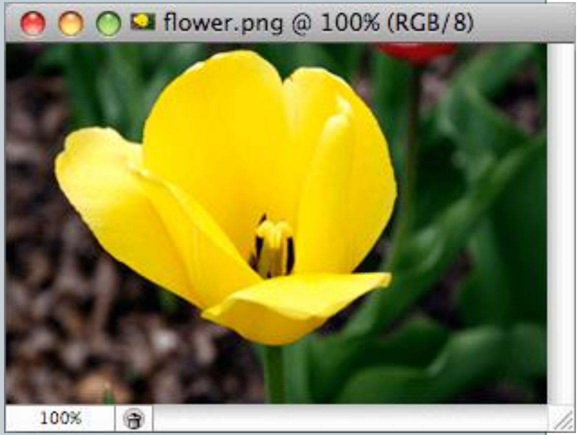


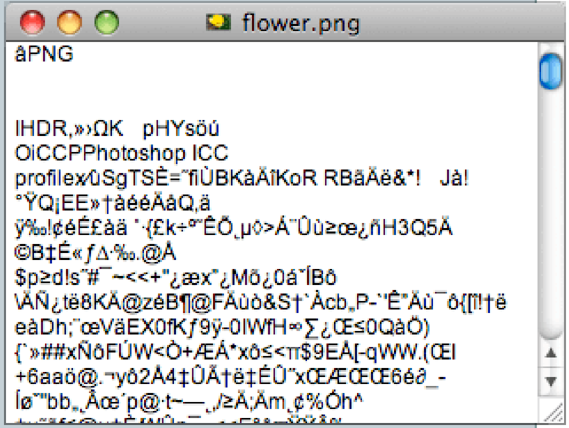
- Visual Studio 2015
- Zoom
  - 2020-03-19 09.24.10
  - 2020-03-19 09.29.31
  - 2020-03-21 12.22.49
  - 2020-03-22 06.13.27
  - 2020-03-22 09.56.35
  - 2020-03-22 11.34.38
  - 2020-03-22 11.36.03
  - 2020-03-26 13.17.25
  - 2020-03-26 14.07.32
  - 2020-03-26 14.08.12
  - 2020-03-27 05.37.48
  - 2020-03-27 07.01.23
  - 2020-03-29 07.35.46
  - 2020-03-29 08.56.00
  - 2020-04-02 09.23.29
  - 2020-04-02 14.07.51
  - 2020-04-03 06.34.33
  - 2020-04-03 06.40.04
  - 2020-04-06 03.52.30
  - 2020-04-06 05.08.22
  - 2020-04-06 05.58.15
  - 2020-04-12 07.37.38
  - 2020-04-12 08.27.22
  - 2020-04-15 10.01.19

audio_only.m4a
playback.m3u
zoom_0.mp4

File Types: Binary Versus Text Files

- ***All*** data and programs are just zeros and ones
  - each digit can have one of two values, hence *binary*
  - *bit* is one binary digit
  - *byte* is a group of eight bits
- ***Text files***: the bits represent printable characters
  - one byte per character for ASCII, the most common code
  - for example, Python program files are text files
  - so is any file created with a "text editor"
- ***Binary files***: the bits represent other types of encoded information, such as executable instructions or numeric data
  - easily read by the computer but not humans
  - they are *not* "printable" files
    - actually, you *can* print them, but they will be unintelligible
    - "printable" means "easily readable by humans when printed"





```
âPNG



IHDR,»›ΩK   pHYsöú
OiCCPPhotoshop ICC
profilexÚSgTSÈ=˜fiÙBKàÄÏKoR RBãÄë&*!   Jà!
°ŸQ¡EE»†àéêÄâQ,ä
ÿ‰!¢éÉ£àä ˙·{£k÷º˝ÊÕ¸µ◊>Á¨Ûù≥œ¿ñH3Q5Ä
©B‡É«ƒ∆·‰.@Å
$p≥d!s˝#¯~<<+"¿æx"¿Mõ¿0â˜ÏBô
\ÄÑ¿të8KÄ@zéB¶@FÄùò&S†`Àcb„P-`'Ê"Äù¯ô{[fi!†ë
eàDh;˘œVäEX0fKƒ9ÿ-0IWfH∞∑¿Œ≤0QàÖ)
{`»##xÑôFÚW<Ò+ÆÁ*xô≤<π$9EÅ[-qWW.(ŒI
+6aaö@.¬yô2Å4‡ÛÄ†ë‡ÉÛ˝xŒÆŒŒ6é∂_-
Íø˜"bb„Âœ´p@·t~—¸/≥Ä;Äm¸¢%Óh^
```

Python: Text Versus Binary Files

- Text files are more readable by humans
- Binary files are more efficient
  - computers read and write binary files more easily than text
- Reading and writing binary files is normally done by a program
- Text files are used only to communicate with humans

<u>Text Files</u>

- Source files
- Occasionally input files
- Occasionally output files

<u>Binary Files</u>

- Executable files (created by compiling source files)
- Usually input files
- Usually output files

Text Files vs. Binary Files

- Number: 127 (decimal)
  - Text file
    - Three bytes: “1”, “2”, “7”
    - ASCII (decimal): 49, 50, 55
    - ASCII (octal): 61, 62, 67
    - ASCII (binary): 00110001, 00110010, 00110111
  - Binary file:
    - One byte (`byte`): 01111111
    - Two bytes (`short`): 00000000 01111111
    - Four bytes (`int`): 00000000 00000000 00000000 01111111

Opening a File

- Python **built-in function open()**.
  - **Syntax:**
    - open(<filename>, <access_mode>)
    - **Parameters**
      - **1. filename:** (string) **name** of the **file** we want to **open**.
        - If the file **resides** in the **same directory** as your **program**, you can simply **specify** the **file name**.
        - If the file is **not present** in the **same directory**, you need to **specify** the **full path** of the **file**.
      - **2. access_mode:**
        - Access mode specifies what you want to do with the **file**.
        - **Default** mode is **read mode ‘r’**.

Opening a File

- Python open() function returns a file object
  - also called “**handle**”.
- Python treats the file as an **object** and we use this **file object** in our program to **access** the **contents** of the file.
- Example:
  - We have an already existing file in our system – “**myfile.txt**”.
  - Let’s open this file in the **Python shell**.
  - **Code:**

```
f = open("myfile.txt", "r")           # file in the same directory
f = open("E:/folder/file.txt")        # specify full path if not in the
                                      #     same directory
```

Access Modes

- Second argument to **open()** is the **access_mode**.
  - **Specifies** the **mode** in which you want to **open** the **file**.
  - Modes:
    - **“r”** for **reading**
    - **“w”** for **writing**
    - **“a”** for **appending**
- **Optional**
  - **default value** of “**r**”, i.e., the **read mode** if we pass **no value** to it.
- By default, **open() function** opens a file in **text mode.**
  - Can specify **binary mode** by adding **“b”** to any of the modes
    - **“wb” – write to binary data**
    - **“rb+” – read and write to binary data**

Access Modes

Mode	Function
r	Open a file for reading. This is the default mode.
w	Open a file for writing. If the file already exists, overwrite its contents. Create a new file if the file does not exist.
a	Open a file for appending. Preserve the file’s contents, add new data to the end of the file.
r+	Open a file for reading and writing.
w+	Allows to write as well as read from the file.
a+	Allows appending as well as reading from the file.

Closing a File

- Built-in Python function – **close()**
  - **No data lost**, **no resources left** still **tied** to the **file**.
  - **Code:**

    ```
    f = open("myfile.txt", "r")
    f.close()
    ```

  - Many programmers often forget to **close a file** after they're done **processing** it.
    - This may lead to **unwanted data loss** and **data corruption**.
  - Closing a file also helps in **freeing up** all the **resources** tied to your program for working with the file.

  - Always a good practice to **close** your **file** once you're done working with it.

Reading Data from a File

- The fun part: **operating** on the **file**.
- Python provides us with various functions to **read** from a **file**.
- One way to **read individual lines** from a file without using any function is by using the **for loop**.
  - **Code:**

```
f = open("10x10_full.txt")
for line in f:
    print(line)
f.close()
```

- Loops over the lines in **10x10_full.txt** and prints them one by one.

Reading Data from a File: read()

- Reads **size** number of bytes from the file.
  - If **size** is **not passed**, the **entire file** is **read**.
  - Code:

```
f = open(“mobydick.txt”)
f.read(5)                    # first call
f.read(22)                   # second call
f.close()
```

- The first call to **read()** reads the first 5 bytes as 5 was passed to the **size argument**.
- The second call reads next 22 bytes.

Reading Data from a File: readline()

- Reads size number of bytes from the line.
  - If we pass **no argument value**, it **reads** the **entire line**.
- Look at the example code in the Python shell:
- Notice how the first call to **readline()** returns 1st line, the second call returns 2nd line and so on.

Reading Data from a File: readlines()

- Reads all the lines from a **file** and **returns** a **list** of the **lines**, **separating** them from one another with **commas**.

Writing Data into a File

- **Writing** into a file is just as easy as **reading** from it.
  - All we need to do is **open a file** in **write mode**.

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

- Then the built-in function for writing into the file object will take care of the rest.

Writing Data into a File: write()

- Takes a **string** as an argument and writes it into the file.
- Returns the **number of characters** written into the file.
  - **Code:**

```
f.write("I am writing new text into the file\n")
```

Writing Data into a File: writelines()

- Takes a **list of strings** as an argument and writes those strings into the file.
  - Be sure to append a **“\n”** at the **end of a string** to make it act as a **line**.
  - The function **does not append** any **line endings** to the elements of the list of strings.
- Once you close the file, all changes get committed to it.

Gotcha: Overwriting a File

- Opening an output file creates an empty file
  - creates a new file if it does not already exist
  - opening an output file that already exists eliminates the old file and creates a new, empty one
    - data in the original file is lost

```
f = open("myFile.txt", "w")
f.write("This will obliterate the previous contents.\n")
f.close()
```

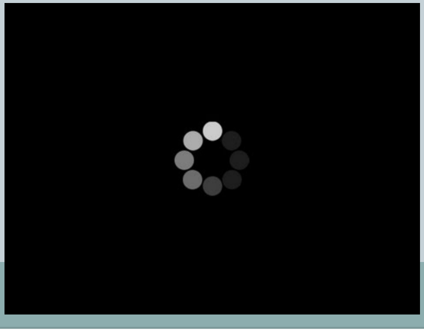
Appending to a File

- Opening an output file with the append option does not overwrite existing contents
  - creates a new file if it does not already exist
  - opening an output file that already exists allows you to add to the end
    - data in the original file is not lost

```
f = open("myFile.txt", "a")
f.write("This will add the contents at the end.\n")
f.close()
```

Buffering

- Not buffered: each byte is read/written from/to disk as soon as possible
  - “little” delay for each byte
  - A disk operation per byte - higher overhead
- Buffered: reading/writing in “chunks”
  - Some delay for some bytes
    - Assume 16-byte buffers
    - Reading: access the first 4 bytes, need to wait for all 16 bytes are read from disk to memory
    - Writing: save the first 4 bytes, need to wait for all 16 bytes before writing from memory to disk
  - One disk operation per buffer of bytes---lower overhead



File Pointer Positions

- Two methods work with the file pointer.
  - **1. tell()**
    - Returns the current position of the file pointer.
    - The number of bytes from the beginning of the file.
  - **2. seek(offset, whence)**
    - Lets you control the position of the file pointer.
    - Takes two parameters:
    - offset – number of positions(bytes) to move forward.
    - whence – Optional and can only be used in binary files. Position from where you want to move forward. It can take three values:
      - 0: move forward from the **start** of the file.
      - 1: seek relative to the **current** position.
      - 2: move backwards from the **end** of the file.

File Exceptions

- Reading and writing files is still a dangerous thing to do
  - The file may not exist
  - The file may be corrupted
  - You might read beyond the end of the file
- We haven't always done this but anytime you work with files you should use a try…except construct to catch any thrown exceptions
  - You put all dangerous activity in the "try" part:
    - opening the file, reading data, writing to it
  - And then any action to take in the except part, should things go wrong

Python File Methods

Method	What it does
close()	Closes the file.
flush()	Flushes the internal buffer.
fileno()	Returns the file descriptor of the file.
next()	Returns the next line from the file.
read(size)	Reads size number of bytes from the file. Reads the entire file if you don’t pass any argument value.
readline(size)	Reads one line from a file.
readlines()	Reads the entire file and returns a list of the lines.
seek(offset, whence)	Lets us control the position of the file pointer.
tell()	Returns the current position of the file pointer.
truncate(size)	It truncates the file to the specified size.
writable()	Returns True if we can write into the file.
write(string)	Writes string into the file.
writelines(list_of_strings)	Writes each element of the list_of_strings into the file.

Summary

- What is a File
  - File Types
- Opening a File
- Access Modes
- Closing a File
- Reading Data from a File
- Writing Data into a File
  - Buffering
- File Pointer Positions
- Python File Methods