File Operations

CS111 Computer Programming

Department of Computer Science
Wellesley College

Files are *persistent* data storage

Concepts in this slide: Persistent vs. volatile memory. The bit as the unit of information.

`titanicdata.txt`

titanic.txt

```
1 Mr Anthony Abbing (victim); age=42.0; class=3rd Class; job=Blacksmith
2 Mr Ernest Owen Abbott (victim); age=21.0; class=Victualling; job=Lounge Pantry
3 Mr Eugene Joseph Abbott (victim); age=14.0; class=3rd Class; job=Scholar
4 Mrs Rhoda Mary 'Rosa' Abbott (survivor); age=39.0; class=3rd Class
5 Mr Rossmore Edward Abbott (victim); age=16.0; class=3rd Class; job=Jeweller
6 Miss Karen Marie Abelseth (survivor); age=16.0; class=3rd Class
7 Mr Olaus Jrgensen Abelseth (survivor); age=25.0; class=3rd Class; job=Farmer
8 Mrs  Abelson (survivor); age=28.0; class=2nd Class
```

Persistent = data that is not dependent on a running program (exists outside it).

Measuring information

The unit of information is the bit. Since one character can be represented in 8 bits (or one byte), byte has become the unit for measuring information stored in a file.

1 B (byte) = 8 bits

1 KB = 1,000 B

1 MB = 1,000,000 B

1 GB = 1,000,000,000 B

1 TB = 1,000,000,000,000 B

A computer has two kinds of storage: volatile memory (i.e., the RAM) and persistent memory (i.e, the hard disk). The RAM memory is faster, but more expensive, so current computers have 8-16 GB of it, while a hard disk is slower, but cheaper, so current computers have up to 1-2 TB.

File Extensions

By **convention**, file extensions (e.g., .pdf) indicate to people the content of a file. Computer programs use other means to detect file content. Here is a list of files we will work with today and in the future:

.txt – plain text file, readable with a text editor. Lines separated by '\n'.

.csv – comma separated values. Simple text, but can be read by spreadsheet applications as well, e.g., Microsoft Excel or Google Sheets.

.json – **J**ava**S**cript **o**bject **n**otation: data structured in lists, dictionaries, or a combination of thereof, to avoid text processing. Can be viewed as text, and loaded directly into Python with the module `json`.

.html – **h**yper**t**ext **m**arkup **l**anguage. A text file, but interpretable by a browser to display content as specified by markup tags.

.css – cascading style sheets. A text file containing rules in a special language, used to style web pages. Also interpretable by a browser.

Working with files in Python

Concepts in this slide: The built-in function **open** to create file objects.

Until now, our data have been stored in variables. However, the universal way of storing data is in a file, which is persistent storage and can be used across applications.

In Python, **open** creates and returns a file object

```
In [ ]: myFile = open('thesis.txt', 'r')
In [ ]: print(myFile)
<_io.TextIOWrapper name = 'thesis.txt' mode='r'
encodings='US-ASCII'>
In [ ]: type(myFile)
Out[ ]: _io.TextIOWrapper
```

io.TextIOWrapper is a type of file object that interprets the files as a stream of text.

A file can be opened for reading (**'r'**), writing (**'w'**), or appending (**'a'**). We **cannot** write or append in a file opened for reading.

The most important methods for a file object are: **read**, **readlines**, **readline**, and **write**, that we'll cover today.

Preferred syntax: `with … as`

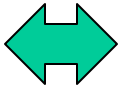
Concepts in this slide: How to open files with the notation `with … as`

It's easy to forget to close a file. This usually isn't too bad when **reading** a file, but can be disastrous when **writing** a file (the contents may not actually be written until the file is closed!)

Python's `with … as` notation for files implicitly closes a file, even if an error occurs within the file operations.

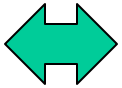
`'r'` – read mode; `'w'` – write mode

```
f = open(filename, 'r')
… file operations involving f …
f.close()
```



```
with open(filename, 'r') as f:
    … file operations involving f …
    # f implicitly closed
    # when with is done.
```

```
f = open(filename, 'w')
… file operations involving f …
f.close()
```



```
with open(filename, 'w') as f:
    … file operations involving f …
    # f implicitly closed
    # when with is done.
```

Reading all text with `read`

cities.txt

```
Wilmington
Philadelphia
Boston
Charlotte
```

```
# reads all lines at once as string
with open('cities.txt', 'r') as inputFile:
    allText = inputFile.read()

In []: allText
Out[]: 'Wilmington\nPhiladelphia\nBoston\nCharlotte'

In []: allText.split()
Out[]: ['Wilmington', 'Philadelphia', 'Boston', 'Charlotte']

In []: moreText = inputFile.read()
```

ValueError: I/O operation on closed file

To notice:

The method **read** returns all the content of the file as a single string. Notice how the returned string contains the newline characters. These are chars that allow the two other methods **readline** and **readlines** to know how to recognize lines. In this case too, if we try to access **inputFile** outside the **with … as** block, we'll get an error about operations with a closed file.

Reading all lines with `readlines`

```
# reads all lines at once as list of strings
with open('cities.txt', 'r') as inputFile:
    allLines = inputFile.readlines()
```

```
In []: allLines
Out[]: ['Wilmington\n', 'Philadelphia\n',
        'Boston\n', 'Charlotte\n']
```

cities.txt

```
Wilmington
Philadelphia
Boston
Charlotte
```

To notice:

The method **`readlines`** returns a list of all lines in a file. Each line is a string terminated by the newline character `'\n'`. These newline characters are visible in the Out[] cell, but not if we print each line. They are also not visible in the text file as well (see green box).

Important: Use **`readlines`** only when a file is not too large. This is because it will read all content and store it in the RAM memory (which, as you read in slide 2, is limited).

Reading one line with `readline`

```
# reads one line at a time
with open('cities.txt', 'r') as inputFile:
    line1 = inputFile.readline()
    line2 = inputFile.readline()

In [ ]: line1
Out[ ]: 'Wilmington\n'

In [ ]: line2
Out[ ]: 'Philadelphia\n'

In [ ]: line3 = inputFile.readline()
ValueError: I/O operation on closed file
```

cities.txt

```
Wilmington
Philadelphia
Boston
Charlotte
```

To notice:

Interpret the first line as saying: open the file 'cities.txt' for reading and assign the variable **inputFile** to it, so that we can access its content.

The indented statement is calling the method **readline** on the file object **inputFile** to read only the first line and save it in the variable **oneline**. Afterwards, the file is closed, which we can notice if we try to use **readline** again on the **inputFile**.

Reading with a For loop

Concepts in this slide: Within a for loop, there is no need to explicitly call the three read methods.

```
def linesFromFile(filename):
    '''Returns a list of all lines in the given file. In
    each line, the terminating newline has been removed.
    '''
    with open(filename, 'r') as inputFile: # open the file
        strippedLines = []
        for line in inputFile:
            strippedLines.append(line.strip())
        return strippedLines
```

No explicit method with the file object. That is, no read, readlines, or readline.

To notice:

Within a **for** loop to read the content of a file, we don't need to call explicitly any of the three methods that we saw. A file object is an iterator, it knows how to iterate over its elements, which are the lines denoted by the newline character.

This is our preferred method for reading from a file.

The string method **strip** removes the newline character and all white space around the line.

Writing Files

Concepts in this slide: Writing to a file that is opened for writing.

A file can be created (or opened) for writing by providing the argument `'w'`, signifying **write mode**.

When writing files, the syntax **`with … as`** is very important, because forgetting to close a file has consequences.

```
with open('memories.txt', 'w') as memfileW:
    memfileW.write('get coffee\n')
    memfileW.write('do CS111 homework\n')
    memfileW.write('vote in midterm elections!\n')
```

At this point, the file named memories.txt is stored persistently in the file system with the following contents:

```
get coffee
do CS111 homework
vote in midterm elections!
```

To notice:

- The second argument `'w'` is what opens the file in writing mode.
- The strings to write in the file contain the newline character `'\n'` as their last character to denote that a new line should start in the file.
- The file memFileW is closed automatically.

Appending to files

How do we add lines to an existing file? We can't open the file in **write mode** (with a `'w'`), because that erases all previous contents and starts with an empty file. Instead, we open the file in **append mode** (with an `'a'`). Any subsequent writes are made after the existing contents:

```
with open('memories.txt', 'a') as memfileA:
    memfileA.write('win Nobel prize\n')
    memfileA.write('eat big sundae\n')
```

Now the file memories.txt has the contents:

```
get coffee
do CS111 homework
vote in midterm elections!
win Nobel prize
eat big sundae
```

If a file does not already exist, opening it in **append mode** creates an empty file.

Exception Handling with `try/except`

Concepts in this slide: New Python keywords: `try` and `except` to catch exceptions.

Misspelled filename:

```
memories = linesFromFile('memory.txt')
```

IOError: No such file or directory: 'memory.txt'

An exception is an error detected during execution of a program.

When part of a program raises an exception (e.g., code for reading a file), it is often better to catch and handle the exception rather than have the program terminate abruptly with an error message (e.g., if a file doesn't exist).

In Python, exceptions can be caught and handled with `try/except` statements.

```
try:
    # Lines of code that may raise an exception
except ErrorType:
    # Lines to execute when exception
    # with type ErrorType is raised
```

Exception Handling Examples

```
try:
    memories = linesFromFile('memory.txt')
except IOError:
    memories = []  # Use empty list in this case
```

To remember

We can use try/except whenever we anticipate errors coming from sources that are related to human input (humans often make mistakes).

```
a = 0
try:
    x = 8/a
    print(x)
except ZeroDivisionError:
    print('Do not divide by zero.')
```

```
while True:
    try:
        i = int(raw_input('Please enter an integer: '))
        print('Good, you entered', i)
        break # Python keyword to exit a loop
    except ValueError:
        print('Not a valid integer. Try again... ')
```

Test your knowledge

1. We mentioned several file endings in slide 3. What are some others that you know? Have you tried to open those files with applications different from the ones which created them (or that you usually open them)? What have you seen?
2. When developing Python programs, why would we need to read files? Why would we need to write files?
3. What would be some good uses for the file methods **`readline`**, **`readlines`**, and **`read`**?
4. What do the three letters **`'r'`**, **`'w'`**, **`'a'`** mean? Do you think one of them can be omitted when opening a file?
5. What would happen if we try to write into a file open for reading?
6. How does the method **`readlines`** recognize lines, so that it can return a list of all lines?
7. **Challenge**: suppose we have a file open for reading, **`inputFile`**. Can you write one line of code to print the total number of words in the file?