CS 106AX Programming Methodologies in JavaScript and Python Autumn 2026

Lecture 15: File Reading in Python

Written by Ben Yan, from materials by Jerry Cain

This is Part II of the *Lecture 15* notes, relevant to when we transitioned in-class from discussing lists in Python to reading data from files. Lists and files are more closely linked than you might at first think ☺.

Lists and Reading Data from Files

Applications that work with lists often need to work with lists that are too large to enter by hand, or change too often to hardcode. **In many cases, it makes sense to store the list values instead in a data file, separating the data from the code**—which has the added benefit that the program can ingest and adapt to different files, e.g., *fantasy.txt*, *romance.txt*, *sci-fi.txt*, without needing to rewrite the core logic.

A *file* is a named capsule of data that's kept in persistent storage (e.g., a hard disk on your computer, or a USB drive or networked filesystem like Google Drive). This means that it survives after your program is done running, unlike the values trapped in variables. Files can store information of many different types and colorful formats. The most common type that we'll explore in 106AX is the ***text or .txt file*—which is composed of readable character data of the variety you'd find in a Python string**. For example,

olivia-rodrigo-deja-vu.txt

```
Car rides to Malibu
Strawberry ice cream
One spoon for two
And trading jackets
```

poem-red-wheelbarrow.txt

```
so much depends
upon

a red
wheelbarrow
```

taylor-swift-style.txt

```
Midnight
You come and pick me
up, no headlights
Long Drive
Could end in burning
flames or paradise
```

Reading Text Files in Python

The standard template for reading a text file unfurls with the following code to ***open* the file, and associate it with a Python variable used as its *file handle***. In practice, this variable is named **`f`** or **`file`**.

```
with open(filename) as variable:
    Code to read the file using variable as the handle
```

The **`with`** statement, which you've likely not been exposed to before, ensures that resources associated with the file are released when Python reaches the end of the **`with`** body. As such, **all code that requires access to the file's contents should be indented and bundled within the `with` block**.

Python offers many ways and cuisines to digest data from a text file, and the Michelin starred ones are:

- Reading the entire file into a single gargantuan string using the **`read`** method.
- Reading lines from a file using the **`readline`** method (one line at a time), or **`readlines`**.
- Using the file handle variable as an iterable, e.g., **`for line in file`**
- Using the **`read`** method together with **`splitlines`** to extract the list of lines from the file.

Reading Entire File as One String

In many aspects, **the easiest way to read a file is to use the (aptly named) `read` method**, which cannibalizes the entire file as a string, with embedded newline characters (`"\n"`) to mark where each line ends. The next line begins right after the previous line's `"\n"` end marker. For example, if `seuss.txt` was the name of the file displayed on the right, then calling `read`, i.e.,

```
One fish
two fish
red fish
blue fish.
```

seuss.txt



```
with open("seuss.txt") as file:
    text = file.read()
```

would ingest the entire file into a string `text` that looks like this:

O	n	e		f	i	s	h	\n	t	w	o		f	i	s	h	\n	r	e	d		f	i	s	h	\n	b	l	u	e		f	i	s	h	.	\n

One major downside to this approach is that loading an entire file into a single string, all at once, can drain a large reservoir of memory if the file itself is huge. If it's too large, the program can even crash!

Reading One Line at a Time

"Slow and steady wins the race" – The Tortoise and the Hare Fables

One safer alternative for reading from a `file` is to use the `readline` method, which reads just the next line (along with its `"\n"` end marker). I personally like to visualize the `file` handle variable as a finger 👉 pointing to some line in the text, and a call like `line = file.readline()` has a twofold effect.

- **First, we read the line we're pointing to**, and render the characters as a string ending in `"\n"`.
- **Second, the finger is bumped down to point to the next line.** That way, subsequent calls to `readline` each read different lines of the file—and cumulatively, they'll read the entire file ☺.

👉
```
Car rides to Malibu
Strawberry ice cream
One spoon for two
And trading jackets
```
`line = file.readline()`
 # line is `"Car rides to Malibu\n"`

👉
```
Car rides to Malibu
Strawberry ice cream
One spoon for two
And trading jackets
```
`line = file.readline()`
 # line is `"Strawberry ice cream\n"`

👉
```
Car rides to Malibu
Strawberry ice cream
One spoon for two
And trading jackets
```
`line = file.readline()`
 # line is `"One spoon for two\n"`

In practice, the calls to `readline` are often wrapped in a `while` loop carousel, which keeps spinning as long as we still have lines to read. We call `.strip` to mercilessly dispose of each newline character.

```
with open(filename) as file:
    while True:
        line = file.readline()
        if line == "": break   # empty line (not even \n), so no more lines left ☹

        line = line.strip()
        Code that works with the line string
```

Reading the Lines into a List

Another strategy is to **read the lines into a Python list**, using the (you guessed it) **`readlines`** method. I love how much Python is like English sometimes. Note that each line in the returned list will have a newline character **`"\n"`** at the end. You can iterate over this list of strings—one per file line—as follows.

```
with open(filename) as file:
      lines = file.readlines()
      for i in range(len(lines)):
            line = lines[i]
            Code that works with the line string, and possibly uses the index i
```

For example, if our *filename* was **`"seuss.txt"`** (the file on the prior page), the **`lines`** list would be

```
["One fish\n", "two fish\n", "red fish\n", "blue fish.\n"]
```

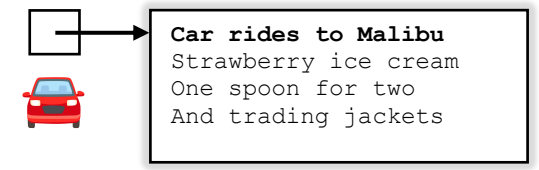
Should you wish to trim the newline markers, you can replace **`line = lines[i]`** in the loop above with **`line = lines[i].strip()`** instead. That way, the first iteration has **`line = "One fish"`**, the second has **`line = "two fish"`**, and so on. Since the file has 4 lines, the loop runs 4 times ☺.

Using the File Handle as an Iterable (Sequence of Lines)

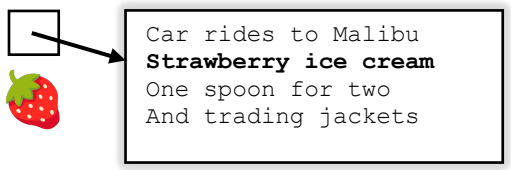
In most cases, **the optimal strategy is to use the `file` handle variable as an iterable / sequence**, and loop over the strings which are its line entries directly. Here's a skeletal template for how to do this.

```
with open(filename) as file:
      for line in file:
            line = line.strip()  # removes the newline character at the end
            Code to process this line string.
```

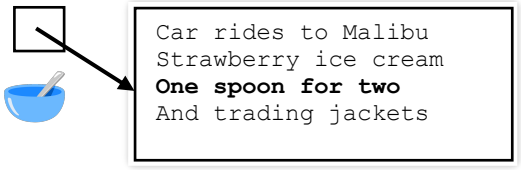
The **`line`** variable is of **string** type, and is assigned the characters of the current line in the iteration. Note that unless we call **`.strip()`** inside the loop, the newline character will be retained, sheesh.



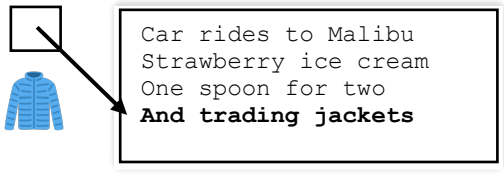
`line → "Car rides to Malibu\n"`



`line → "Strawberry ice cream\n"`



`line → "One spoon for two\n"`



`line → "And trading jacket\n"`

Like before, one can envision the **`file`** iterable as a pointer to one line of the file, starting at the first line. Iterating through **`file`** is tantamount to moving the pointer so that it slides down line by line.

Reading with the Split Lines Method

Lastly, so long as you're working with files of modest size (e.g., a Shakespeare sonnet or even a novella is fine; the entire archive of Wikipedia, not so much), **one way to read a file as a list of lines is to marry the `read` method from the *file* class with the `splitlines` method from the *string* class**, as follows.

```
with open(filename) as file:
    lines = f.read().splitlines()
    Code to process the lines of the file.
```

The **`read`** method swallows the entire file into one string, and **`splitlines`** takes that big, crunchy string and, like a hungry af Stanford frosh at TAP after a crazy night out, absolutely devours and digests it into a list of chewed up string fragments. That is, a list of file lines that were originally divided by **`"\n"`**.

For example, if our file were lyrics from the song ***Expectations*** (with its internal contents and blood-red guts spilled out below on the left), the resulting list **`lines = file.read.splitlines()`** would be

olivia-rodrigo-expecations.txt

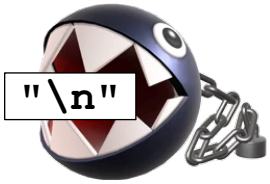
```
I met him at a party
I think he was on drugs
He wasn't smart or funny
I convinced myself he was
```

`file.read()`

I		m	e	t		h	i	m		a	t		a	
p	a	r	t	y	\n	I		t	h	i	n	k		h
e		w	a	s		o	n		d	r	u	g	s	\n
H	e		w	a	s	n	'	t		s	m	a	r	t
	o	r		f	u	n	n	y	\n	I		c	o	n
v	i	n	c	e	d		m	y	s	e	l	f		
	h	e		w	a	s	\n							

```
lines = ["I met him at a party", "I think he was on drugs",
         "He wasn't smart of funny", "I convinced myself he was"]
```

You can iterate through the **`lines`** list as you would with any Python list, i.e. **`for line in lines:`**



You'll notice the strings in **`lines`** don't have that pesky **`\n`**. **The advantage of this approach is that `splitlines()` strips the newline characters form the end of each line**, a process that is sometimes hilariously referred to as ***chomping***. Picture the Chain Chomp from *Super Mario* the next time you use **`splitlines`** ☺.

How did that feel? Wait … you seem pretty sad for a Python programmer so in love with this course…

✍ Optional: File Writing

You've just learned about **`read`** and file-reading methods in Python. Perhaps you won't be astonished to learn there's a **`write`** method as well, and **`writelines`**, etc., which does the exact opposite—taking string data imprisoned in variables, and releasing it out to a text file. Here's some code that does this ☺.

```
with open("seuss.txt", "w") as file:
    file.write("One fish\n")
    file.write("two fish\n")
    file.write("red fish\n")
    file.write("blue fish.\n")
```

seuss.txt

```
One fish
two fish
red fish
blue fish
```

Note that **`write`** doesn't automatically include a newline (**`\n`**); you need to embed one yourself! Also, we specify **`"w"`** in the **`open()`** call to indicate we're writing to and possibly creating a file, not reading.

Finding the Longest Line in a File

As a delightful, literature-inspired example, **we'll write a function `findLongestLine` that returns the longest line in a file**. For example, given a file haunted by the raw poetry of Shakespeare's ***Sonnet 2***,

```
When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
Thy youth's proud livery so gazed on now,
Will be a tattered weed of small worth held:
Then being asked, where all thy beauty lies,
Where all the treasure of thy lusty days;
To say within thine own deep sunken eyes,
Were an all-eating shame, and thriftless praise.
How much more praise deserved thy beauty's use,
If thou couldst answer 'This fair child of mine
Shall sum my count, and make my old excuse'
Proving his beauty by succession thine.
  This were to be new made when thou art old,
  And see thy blood warm when thou feel'st it cold
```



Is Shakespeare really dead???



then our function should return the longest line of the sonnet, which incidentally is the last one, i.e.

```
"  And see thy blood warm when thou feel'st it cold\n"
```

Note this line's indentation is included—it's standard for the last two lines of a sonnet, known as the *heroic couplet*, to be indented—as well as the newline `"\n"`. However, the inclusion of the latter does not affect which line is considered the ***longest***, since all lines in the file include one `"\n"` at the end.

Here's my implementation of **`findLongestLine`**, which uses the **`file`** handle as an iterator, and returns the **`\n`** as part of the line. As we iterate over the lines, we keep track of the **`longest`** line we've seen so far—a current "Guinness World Record" of sorts—updating it only when we see a longer string.

```
def findLongestLine(filename):
    longest = ""  # "Guinness World Record" of longest line encountered so far

    with open(filename) as file:
        for line in file:
            if len(line) > len(longest):  # previous record broken!
                longest = line    # setting new record

    return longest
```



Exception Handling and Try / Except

Opening a file in Python is a lot like opening an Instagram reel that your close friend sends you with the most devilish grin on their face 😈. You really don't know what to expect. You probably should not open it. Maybe the file's sus, it'll crash your phone, or it no longer exists. Python handles these situations—

and many other errors or jump scares that can occur during a program's execution—using a mechanism called ***exception handling***, which has become standard parlance in modern programming languages.

Exceptions belong to classes, for instance, file errors are classified as an **`IOError`** exception, applying operations on the wrong data type are classified as **`TypeError`**, attempting to access a key that doesn't exist in a dictionary is a **`KeyError`**, et cetera. **Raising an exception terminates the program's execution *unless* your program is safeguarded with a `try` statement and code to safely handle that exception** ☺.

In its most skeletal, brutish, and barebones form, the syntax for the Python **`try`** statement is

`try:`
    Code in which exceptions / errors may occur.
`except` ***type*`:`**
    Code that runs only when an exception / error of this type occurs. ⚠

where ***type* is the class name of exception being handled**, e.g., **`IOError`**, **`KeyError`**, **`TypeError`**, et cetera. If you want a generic catch-all for all built-in exceptions, you can write **`Exception`** as the ***type***, i.e. **`except Exception`**. The range of statements in which the exception is caught includes not only the lines of code directly imprisoned in the **`try`** body, but also in any functions these lines call. If the exception occurs in say a helper function, we ascend through any nested stack frames and function calls until we're back to the line of code in the **`try`** body that kicked off the chain of events in the first place.

As a possible use case for file reading, consider the function **`getExistingFile`, which repeatedly prompts the user until they provide the name of an existing file,** i.e. one that can be opened for input.

```
def getExistingFile(prompt = "> Input file: ")
      while True:
            filename = input(prompt)
            try:  # code that would normally crash if the file can't be opened, i.e. does not exist!
                  with open(filename):
                        print("File opened!")
                        return filename

            except IOError:  # go here if exception occurs in the try block
                  print("Can't open that file. Try again!")
```

If the **`open`** call succeeds, the body of the **`with`** statement simply returns the filename, indicating its validity. However, if an **`IOError`** occurs, the **`except`** clause prints an error message, and we backtrack to the top of the **`while`** loop to try again. Below is a sample run of this program in my ***movies*** folder ☺.

Terminal: GetExistingFile

```
> Input file: brand-new-day-script.txt
Can't open that file. Try again!
> Input file: wildwood-script.txt
Can't open that file. Try again!
> Input file: the-odyssey-script.txt
File opened!
```

Finder: Movies Folder

