

02-16-2022

CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

9: Reading files



David Kauchak
he/him/his
Lectures



Alexandra Papoutsaki
she/her/hers
Lectures



Zilong Ye
he/him/his
Labs

Welcome to lecture 9, where we take working with data to a new level.

Lecture 9: Reading files

- ▶ Files
- ▶ Strings
- ▶ More Files

Our main focus today will be files, but we will also encounter strings again.



<https://businesscloud.ca/wp-content/uploads/2016/03/hard-drive-vs-ram.jpg>

What is a file?

- ▶ A chunk of data stored on the hard disk/drive.
- ▶ Data stored in the hard drive persist even if we turn off our computer.
- ▶ When a program is running, all the data is generating and processing is moved by the CPU into the main memory, e.g., RAM.
- ▶ The main memory is faster, but doesn't persist when the power goes off.

What do I mean by files? A file is a chunk of data stored in the hard drive/disk. Data stored in the hard drive persist even if we turn off our computer. That's why when we save our pictures, homework, etc, we can be sure that when we turn our computer back on we don't lose our work. When a program is running, all the data it needs to process or that it generates, are moved by the CPU (the brain of our computer) into the main memory. You might have heard of RAM or random access memory, a type of main memory. The main memory is much faster than hard drives (and more expensive!) But it doesn't persist when the power goes off.

Opening files

- ▶ To read a file in Python, we first need to open it.
 - ▶ If we just want to hard-code the name and the name of the file is "some_file_name" then:
 - ▶ `file = open("some_file_name", "r")`
 - ▶ or if the name of the file is in a variable, then:
 - ▶ `name_of_file = "some_file_name"`
 - ▶ `file = open(name_of_file, "r")`
- ▶ `open` is a function that takes two parameters, both strings:
 - ▶ the first parameter is a string that identifies the name of the file.
 - ▶ Python assumes that the file is in the same directory as your .py program, unless you tell it to look elsewhere.
 - ▶ the second parameter is another string telling Python what you want to do with the file:
 - ▶ `r` stands for "read", that is, we're going to read some data from the file.
- ▶ `open` returns a file object that we can use later on for reading purposes
 - ▶ above, we've saved that in a variable called `file`, but I could have called it anything else.

Let's see how we would read files in Python. To accomplish that, we first need to learn how we are to open them for reading. There is a function called `open` that takes two parameters, the name of the file (or a variable storing it) we want to read, a mode of how you want to open the file. For example, `open(name_of_file, "r")`, will open the file whose name is stored in the `name_of_file` variable. The `"r"` mode indicates that we open the file to read it. Please keep in mind that the file is expected to be stored in the same directory with our .py python file.

Reading a file line by line

▶ Look at function `line_count` in [file-basics.py](#)

▶ This is a common pattern for reading from files:

▶ 1. Open the file

▶ `file = open(filename, "r")`

▶ 2. Iterate through the file a line at a time

```
for line in file:
```

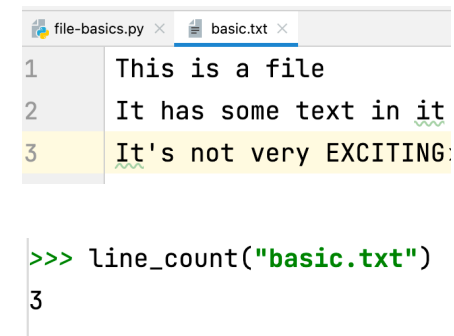
...

▶ What you want to do as you read the file is the ...

▶ 3. Close the file

▶ `file.close()`

▶ In this case, we're just incrementing the counter, `line_count`, each time through the loop. The result is a count of the number of lines in the file.



```
file-basics.py × basic.txt ×
1 This is a file
2 It has some text in it
3 It's not very EXCITING:

>>> line_count("basic.txt")
3
```

Let's look at the function `line_count` in `file-basics.py`. We will soon identify a common pattern for reading from files. We open them, and then we have a for loop that goes through each of the lines in the opened file and does something. We should also make sure that we close the file when we are done with reading it. In this function, we keep a counter for how many lines it has.

Printing the contents of a file line by line

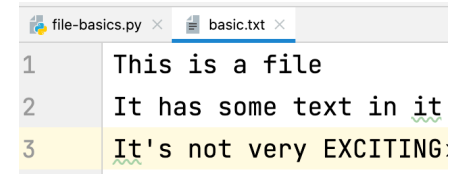
- ▶ Look at function `print_file_almost` in [file-basics.py](#)
 - ▶ Again, very similar structure but we print lines of file.

```
>>> print_file_almost("basic.txt")
```

```
This is a file
```

```
It has some text in it
```

```
It's not very EXCITING
```



```
file-basics.py × basic.txt ×  
1 This is a file  
2 It has some text in it  
3 It's not very EXCITING
```

- ▶ Anything funny about this?
 - ▶ There are extra blank lines between the output!

The `print_file_almost` attempts to print the contents of the file. When we call this function, we notice something funny: there are extra blank lines between the output.

Debugging print_file_almost

- ▶ To try and understand this, let's add some debugging statements, specifically, `print(len(line))` in the for loop and run again:

```
>>> print_file_almost("basic.txt")
```

```
This is a file
```

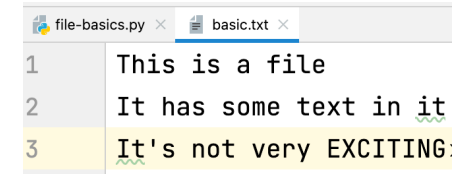
```
15
```

```
It has some text in it
```

```
23
```

```
It's not very EXCITING
```

```
22
```



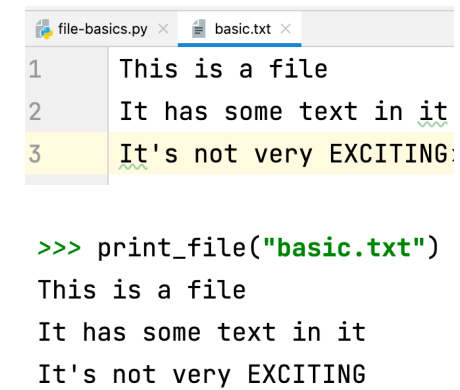
```
file-basics.py × basic.txt ×  
1 This is a file  
2 It has some text in it  
3 It's not very EXCITING
```

- ▶ If you count the characters, there's one extra!

Why does this happen? Let's try to debug our function by adding a print statement that shows the length of line. When we count the characters, we notice that there's an extra one!

Printing the contents of a file line by line - correct version

- ▶ Look at function `print_file` in [file-basics.py](#)
 - ▶ The problem before was that when we read a line, we also read the end of line character.
 - ▶ What's really in the [basic.txt](#) file is:
 - ▶ This is a file\nIt has some text in it\nIt's not very EXCITING
 - ▶ We use the `strip()` method.
 - ▶ Returns a copy of the string without leading and trailing whitespaces.



```
file-basics.py x basic.txt x
1 This is a file
2 It has some text in it
3 It's not very EXCITING:

>>> print_file("basic.txt")
This is a file
It has some text in it
It's not very EXCITING
```

That extra character is a special, end of line, character. The proper way of going about it is to use the `strip()` method which returns a copy of the string after it has removed any leading and trailing whitespaces as can be seen in `file-basics.py`.

Lecture 9: Reading files

- ▶ Files
- ▶ Strings
- ▶ More Files

It's intermission time. Let's make a pause to revisit strings.

Splitting a string into a list of substrings

- ▶ `string.split(sep)`

- ▶ Returns a list of the substrings in the string, using `sep` as the delimiter string.

- ▶ If no delimiter is provided, string is split according to any whitespace character (spaces, tabs, end of line characters).

```
>>> "this is a sentence with words".split()
['this', 'is', 'a', 'sentence', 'with', 'words']
>>> s = "this is a sentence with words"
>>> s.split()
['this', 'is', 'a', 'sentence', 'with', 'words']
>>> s.split("s")
['thi', ' i', ' a ', 'entence with word', '']
```

Specifically, let's see two useful methods. The first one is called `split` and it optionally takes as a parameter a string that acts as a delimiter. What we get back is a list of the substrings in the original string split by the delimiter. If we don't provide one, the default is any whitespace character (space, tab, EOL).

Checking whether a string is an uppercase string

- ▶ `string.isupper()`

- ▶ Returns True if the string is an uppercase string, False otherwise.

```
>>> "banana".isupper()
False
>>> "Banana".isupper()
False
>>> "BANANA".isupper()
True
```

The second method is `isupper` and returns True if the string has all characters in uppercase and false otherwise.

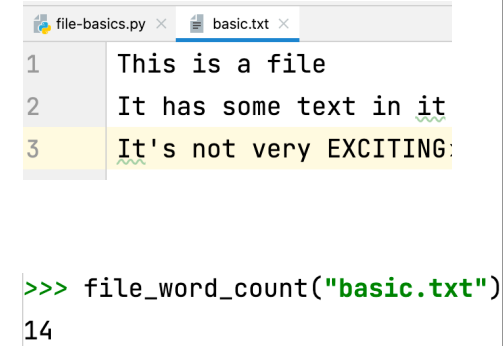
Lecture 9: Reading files

- ▶ Files
- ▶ Strings
- ▶ **More Files**

OK, we covered everything we needed about strings. Let's get back to files.

Counting number of words in a file

- ▶ Look at function `file_word_count` in [file-basics.py](#)
 - ▶ Again, very similar structure but we count number of words by splitting each line we read using the `split` method.
 - ▶ Instead of adding 1 to the counter each time through the loop, we add `len(words)`.



The screenshot shows a code editor with two tabs: 'file-basics.py' and 'basic.txt'. The 'basic.txt' tab is active and displays three lines of text: '1 This is a file', '2 It has some text in it', and '3 It's not very EXCITING:'. The third line is highlighted in yellow. Below the editor, a terminal window shows the command `>>> file_word_count("basic.txt")` and the output `14`.

We will continue going over the functions in `file-basics.py`. If we look at the function `file_word_count`, we will notice that it has a similar structure and that it now uses the `split` method we just learned about. What it does, is that it counts the number of words in the file.

Counting number of capitalized words in a list

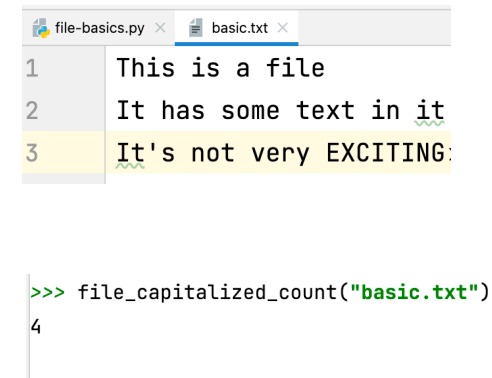
- ▶ Look at function `capitalized_word_count` in [file-basics.py](#)
 - ▶ Given a list of words, it iterates, one word at a time, and checks if the word is capitalized using the `isupper` method of the first character.
 - ▶ If so, increments the count.

```
>>> capitalized_word_count(["Hello", "HELLO", "hello"])  
2
```

The function `capitalized_word_count` receives a list of words, goes over them, and counts how many of them are capitalized using the `isupper` method.

Counting number of capitalized words in a file

- ▶ Look at function `file_capitalized_count` in [file-basics.py](#)
- ▶ Given a file, it checks each of its lines and counts how many capitalized words exist in total.



```
file-basics.py × basic.txt ×
1 This is a file
2 It has some text in it
3 It's not very EXCITING:

>>> file_capitalized_count("basic.txt")
4
```

We can use this function to build a method, `file_capitalized_count` that will count all capitalized words in a file.

[word-stats.py](#)

- ▶ Look at `file_stats` function
 - ▶ It iterates over each item in the file and keeps track of:
 - ▶ longest string found,
 - ▶ shortest string found,
 - ▶ total length of the strings iterated over, and
 - ▶ the total number of strings/items.
 - ▶ How does it keep track of the longest?
 - ▶ It starts with the empty string (`""`), compares every word to the longest so far.
 - ▶ If longer, updates longest.
 - ▶ What does `if shortest == "" or len(word) < len(shortest)` do? Why don't we have it for the longest condition?
 - ▶ For longest, we started with the shortest possible string, so any string will be longer.
 - ▶ Hard to start with the longest possible string :)
 - ▶ Instead we add a special case for the first time through the loop.
 - ▶ We could have initialized `shortest` to be a really long string, but this is a more robust solution

Now let's look at the `word-stats` file and specifically its `file_stats` function which keeps track of the longest and shortest string, the total length of the strings iterated over, and the total number of strings. How does it keep track of the longest string? We start with an empty string and compare every word to the longest so far; if longer, we update it. In contrast, for the shortest, we would not know what the longest possible string would be. That's why we will start with an empty string and we will use an `or` operation for all other instances.

Practice time

- ▶ Write a function called `read_numbers` that takes a file of numbers (one per line) and generates a list consisting of the numbers in that file.
- ▶ Don't forget to use the `int` function to turn strings into numbers.

```
>>> def read_numbers(filename):
...     file = open(filename, "r")
...
...     numbers = []
...
...     for number in file:
...         numbers.append(int(number))
...
...     file.close()
...
...     return numbers
...
>>> read_numbers("numbers.txt")
[93, 27, 44, 32, 50, 60, 31, 37, 43, 73, 14, 72, 26, 73, 6, 60, 12, 40, 68, 79, 49, 71, 10, 63, 9, 59, 2
```

Let's practice by writing a function called `read_numbers` that takes a file of numbers (one per line) and generates a list consisting of the numbers in that file. Don't forget to use the `int` function to turn strings into numbers.

What if we want to find the most frequent value in the data?

- ▶ Assume you have read a file of numbers and you got this list:
 - ▶ [1, 2, 3, 2, 3, 2, 1, 1, 5, 4, 4, 5]
 - ▶ How would you do it on paper? How did you do it?
 - ▶ Kept a tally of the number.
 - ▶ Each time you saw a new number, added it to your list with a count of 1.
 - ▶ If it was something you'd seen already, increase tally by 1.
- ▶ Key idea: keeping track of two things:
 - ▶ a key, which is the thing you're looking up, and
 - ▶ a value, which is associated with each key.

What if I gave you a file that you read using your `read_numbers` function and you got back [1, 2, 3, 2, 3, 2, 1, 1, 5, 4, 4, 5]. How would you find the most frequent value in the data? Go on, do it on paper! I assume you kept a tally of the number. If you see it for the first time, you write down 1. If you have encountered it before you increase the counter for that number by 1. This key idea of keeping track of two things can be captured in a pair of key and values. Keys are things we want to look up and values are associated with them. On Monday, we will continue with looking at a new data structure, dictionaries, that will store such pairs!

Resources

- ▶ Textbook: [Chapter 11 \(up to 11.4\)](#)
- ▶ [file-basics.py](#)
- ▶ [basic.txt](#)
- ▶ [word-stats.txt](#)
- ▶ [numbers.txt](#)

Homework

- ▶ Assignment 4 (ongoing)