**LEARN PYTHON: FILES**

**Reading a File**

Computers use file systems to store and retrieve data. Each [file](https://www.codecademy.com/resources/docs/python/files?page_ref=catalog) is an individual container of related information. If you’ve ever saved a document, downloaded a song, or even sent an email you’ve created a file on some computer somewhere. Even **script.py**, the Python program you’re editing in the learning environment, is a file.

So, how do we interact with files using Python? We’re going to learn how to read and write different kinds of files using code. Let’s say we had a file called **real\_cool\_document.txt** with these contents:

**real\_cool\_document.txt**

Wowsers!

We could read that file like this:

**script.py**

with open('real\_cool\_document.txt') as cool\_doc:  
  cool\_contents = cool\_doc.read()  
print(cool\_contents)

This opens a file object called cool\_doc and creates a new indented block where you can read the contents of the opened file. We then read the contents of the file cool\_doc using cool\_doc.read() and save the resulting string into the variable cool\_contents. Then we print cool\_contents, which outputs the statement Wowsers!.

**Instructions**

**1.**

Use with to open the file **welcome.txt**. Save the file object as text\_file.

Checkpoint 2 Passed

Hint

Use this syntax to open a file:

with open('filename.txt') as file\_object:  
  # indented block here

**2.**

Read the contents of text\_file and save the results in text\_data.

Checkpoint 3 Passed

Hint

Use this syntax on a file object to read the contents of the file into a variable:

file\_string = file\_object.read()

**3.**

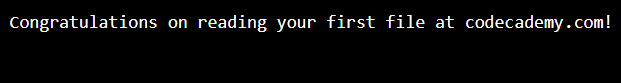
Print out text\_data.

**script.py**

with open('welcome.txt') as text\_file:

  text\_data = text\_file.read()

print(text\_data)

****

**Iterating Through Lines**

When we read a file, we might want to grab the whole document in a single string, like .read() would return. But what if we wanted to store each line in a variable? We can use the .readlines() function to read a text file line by line instead of having the whole thing. Suppose we have a file:

**keats\_sonnet.txt**

To one who has been long in city pent,  
’Tis very sweet to look into the fair  
And open face of heaven,—to breathe a prayer  
Full in the smile of the blue firmament.

**script.py**

with open('keats\_sonnet.txt') as keats\_sonnet:  
  for line in keats\_sonnet.readlines():  
    print(line)

The above script creates a temporary file object called keats\_sonnet that points to the file **keats\_sonnet.txt**. It then iterates over each line in the document and prints the entire file out.

**Instructions**

**1.**

Using a with statement, create a file object pointing to the file **how\_many\_lines.txt**. Store that file object in the variable lines\_doc.

Checkpoint 2 Passed

Hint

Remember to open a file using with syntax:

with open('filename.txt') as file\_object:  
  # indented block here

**2.**

Iterate through each of the lines in lines\_doc.readlines() using a for loop.

Inside the for loop print out each line of **how\_many\_lines.txt**.

Checkpoint 3 Passed

Hint

You can use the following syntax to print out each line of a file:

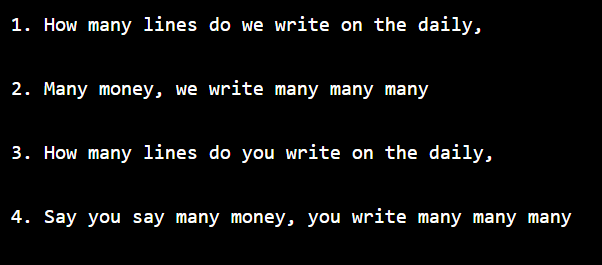
for line in file\_object.readlines():  
  print(line)

**script.py**

with open('how\_many\_lines.txt') as lines\_doc:

  for line in lines\_doc.readlines():

    print(line)

****

**Reading a Line**

Sometimes you don’t want to iterate through a whole file. For that, there’s a different file method, [.readline()](https://www.codecademy.com/resources/docs/python/files/readline?page_ref=catalog), which will only read a single line at a time. If the entire document is read line by line in this way subsequent calls to .readline() will not throw an error but will start returning an empty string (""). Suppose we had this file:

**millay\_sonnet.txt**

I shall forget you presently, my dear,  
So make the most of this, your little day,  
Your little month, your little half a year,  
Ere I forget, or die, or move away,

**script.py**

with open('millay\_sonnet.txt') as sonnet\_doc:  
  first\_line = sonnet\_doc.readline()  
  second\_line = sonnet\_doc.readline()  
  print(second\_line)

This script also creates a file object called sonnet\_doc that points to the file **millay\_sonnet.txt**. It then reads in the first line using sonnet\_doc.readline() and saves that to the variable first\_line. It then saves the second line (So make the most of this, your little day,) into the variable second\_line and then prints it out.

**Instructions**

**1.**

Using a with statement, create a file object pointing to the file **just\_the\_first.txt**. Store that file object in the variable first\_line\_doc.

Checkpoint 2 Passed

Hint

Remember to open a file using with syntax:

with open('filename.txt') as file\_object:  
  # indented block here

**2.**

Save the first line of **just\_the\_first.txt** into the variable first\_line.

Checkpoint 3 Passed

Hint

Use the file object method .readline() to store a line into the variable line.

**3.**

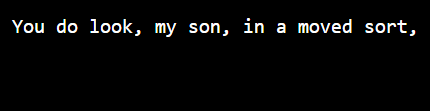
Print out the variable first\_line.

**script.py**

with open('just\_the\_first.txt') as first\_line\_doc:

  first\_line = first\_line\_doc.readline()

  print(first\_line)

****

**Writing a File**

Reading a file is all well and good, but what if we want to create a file of our own? With Python we can do just that. It turns out that our [open()](https://www.codecademy.com/resources/docs/python/built-in-functions/open?page_ref=catalog) function that we’re using to open a file to read needs another argument to open a file to write to.

**script.py**

with open('generated\_file.txt', 'w') as gen\_file:  
  gen\_file.write("What an incredible file!")

Here we pass the argument 'w' to open() in order to indicate to open the file in write-mode. The default argument is 'r' and passing 'r' to open() opens the file in read-mode as we’ve been doing.

This code creates a new file in the same folder as *script.py* and gives it the text What an incredible file!. It’s important to note that if there is already a file called *generated\_file.txt* it will completely overwrite that file, erasing whatever its contents were before.

**Instructions**

**1.**

Create a file object for the file **bad\_bands.txt** using the open() function with the w argument. Assign this object to the temporary variable bad\_bands\_doc.

Checkpoint 2 Passed

Hint

Remember to use the with statement to open a file and pass the 'w' argument to open it in write mode:

with open('file\_to\_write.txt', 'w') as file\_object:  
  # indent

**2.**

Use the bad\_bands\_doc.write() method to add the name of a musical group you dislike to the document bad\_bands.

**script.py**

with open('bad\_bands.txt', 'w') as bad\_bands\_doc:

  bad\_bands\_doc.write("Poison")

**Appending to a File**

So maybe completely deleting and overwriting existing files is something that bothers you. Isn’t there a way to just add a line to a file without completely deleting it? Of course there is! Instead of opening the file using the argument 'w' for write-mode, we open it with 'a' for append-mode. If we have a generated file with the following contents:

**generated\_file.txt**

This was a popular file...

Then we can add another line to that file with the following code:

**script.py**

with open('generated\_file.txt', 'a') as gen\_file:  
  gen\_file.write("\n... and it still is")

In the code above we open a file object in the temporary variable gen\_file. This variable points to the file *generated\_file.txt* and, since it’s open in append-mode, adds the string \n... and it still is to the file. The newline character \n moves to the next line before adding the rest of the string. If you were to open the file after running the script it would look like this:

**generated\_file.txt**

This was a popular file...  
... and it still is

Notice that opening the file in append-mode, with 'a' as an argument to open(), means that using the file object’s [.write()](https://www.codecademy.com/resources/docs/python/files/write?page_ref=catalog) method *appends* whatever is passed to the end of the file. If we were to run **script.py** again, this would be what **generated\_file.txt** looks like:

**generated\_file.txt**

This was a popular file...  
... and it still is  
... and it still is

Notice that we’ve appended "\n... and it still is" to the file a second time! This is because in **script.py** we opened **generated\_file.txt** in append-mode.

**Instructions**

**1.**

We’ve got a file, **cool\_dogs.txt**, filled with all the cool dogs we know. Somehow while compiling this list we forgot about one very cool dog. Let’s fix that problem by adding him to our **cool\_dogs.txt**.

Open up our file **cool\_dogs.txt** in append-mode and assign it to the file object cool\_dogs\_file.

Checkpoint 2 Passed

**2.**

Inside your with block, add “Air Buddy\n” to **cool\_dogs.txt**. Air Buddy is a Golden Retriever that plays basketball, which more than qualifies him for this list. The \n character moves to the next line after appending the string.

Checkpoint 3 Passed

Hint

Use cool\_dogs\_file.write() to add a string to the end of **cool\_dogs.txt**

**script.py**

with open('cool\_dogs.txt', 'a') as cool\_dogs\_file:

  cool\_dogs\_file.write("Air Buddy\n")

**What's With "with"?**

We’ve been opening these files with this with block so far, but it seems a little weird that we can only use our file variable in the indented block. Why is that? The with keyword invokes something called a *context manager* for the file that we’re calling open() on. This context manager takes care of opening the file when we call open() and then closing the file after we leave the indented block.

Why is closing the file so complicated? Well, most other aspects of our code deal with things that Python itself controls. All the variables you create: integers, lists, dictionaries — these are all Python objects, and Python knows how to clean them up when it’s done with them. Since your files exist *outside* your Python script, we need to tell Python when we’re done with them so that it can close the connection to that file. Leaving a file connection open unnecessarily can affect performance or impact other programs on your computer that might be trying to access that file.

The with syntax replaces older ways to access files where you need to call [.close()](https://www.codecademy.com/resources/docs/python/files/close?page_ref=catalog) on the file object manually. We can still open up a file and append to it with the old syntax, as long as we remember to close the file connection afterwards.

fun\_cities\_file = open('fun\_cities.txt', 'a')  
  
# We can now append a line to "fun\_cities".  
fun\_cities\_file.write("Montréal")  
  
# But we need to remember to close the file  
fun\_cities\_file.close()

In the above script we added “Montréal” as a new line in our file **fun\_cities.txt**. However, since we used the older-style syntax, we had to remember to close the file afterwards. Since this is necessarily more verbose (requires at least one more line of code) without being any more expressive, using with is preferred.

**Instructions**

**1.**

In **script.py** there’s a file object that doesn’t get closed correctly. Let’s fix it by changing the syntax!

Remove this line:

close\_this\_file = open('fun\_file.txt')

And change it to use the with syntax from our previous exercises.

Remember to indent the rest of the body so that we don’t get an IndentError.

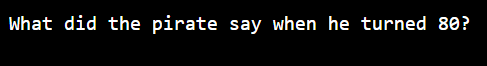
**script.py**

with open("fun\_file.txt") as close\_this\_file:

  setup = close\_this\_file.readline()

  punchline = close\_this\_file.readline()

print(setup)

****

**What Is a CSV File?**

Text files aren’t the only thing that Python can read, but they’re the only thing that we don’t need any additional parsing library to understand. [CSV](https://www.codecademy.com/resources/docs/general/csv?page_ref=catalog) files are an example of a text file that impose a structure to their data. CSV stands for Comma-Separated Values and CSV files are usually the way that data from spreadsheet software (like Microsoft Excel or Google Sheets) is exported into a portable format. A spreadsheet that looks like the following

| **Name** | **Username** | **Email** |
| --- | --- | --- |
| Roger Smith | rsmith | [wigginsryan@yahoo.com](mailto:wigginsryan@yahoo.com) |
| Michelle Beck | mlbeck | [hcosta@hotmail.com](mailto:hcosta@hotmail.com) |
| Ashley Barker | a\_bark\_x | [a\_bark\_x@turner.com](mailto:a_bark_x@turner.com) |
| Lynn Gonzales | goodmanjames | [lynniegonz@hotmail.com](mailto:lynniegonz@hotmail.com) |
| Jennifer Chase | chasej | [jchase@ramirez.com](mailto:jchase@ramirez.com) |
| Charles Hoover | choover | [choover89@yahoo.com](mailto:choover89@yahoo.com) |
| Adrian Evans | adevans | [adevans98@yahoo.com](mailto:adevans98@yahoo.com) |
| Susan Walter | susan82 | [swilliams@yahoo.com](mailto:swilliams@yahoo.com) |
| Stephanie King | stephanieking | [sking@morris-tyler.com](mailto:sking@morris-tyler.com) |
| Erika Miller | jessica32 | [ejmiller79@yahoo.com](mailto:ejmiller79@yahoo.com) |
|  |  |  |

In a CSV file that same exact data would be rendered like this:

**users.csv**

Name,Username,Email  
Roger Smith,rsmith,wigginsryan@yahoo.com  
Michelle Beck,mlbeck,hcosta@hotmail.com  
Ashley Barker,a\_bark\_x,a\_bark\_x@turner.com  
Lynn Gonzales,goodmanjames,lynniegonz@hotmail.com  
Jennifer Chase,chasej,jchase@ramirez.com  
Charles Hoover,choover,choover89@yahoo.com  
Adrian Evans,adevans,adevans98@yahoo.com  
Susan Walter,susan82,swilliams@yahoo.com  
Stephanie King,stephanieking,sking@morris-tyler.com  
Erika Miller,jessica32,ejmiller79@yahoo.com

Notice that the first row of the CSV file doesn’t actually represent any data, just the labels of the data that’s present in the rest of the file. The rest of the rows of the file are the same as the rows in the spreadsheet software, just instead of being separated into different cells they’re separated by… well I suppose it’s fair to say they’re separated by commas.

### Instructions

**1.**

 CSV files are just plain text files!

Open logger.csv using our standard with syntax, saving the file object in the temporary variable log\_csv\_file.

Checkpoint 2 Passed

**2.**

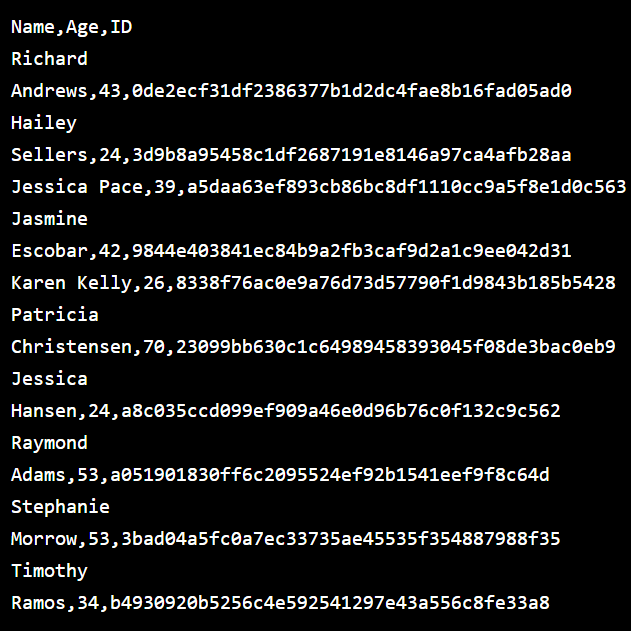
Print out the contents of logger.csv by calling .read() on the file. Notice that it is parsed as a string.

**script.py**

with open('logger.csv') as log\_csv\_file:

  var = log\_csv\_file.read()

print(var)

****

**Reading Different Types of CSV Files**

I need to level with you, I’ve been lying to you for the past two exercises. Well, kind of. We’ve been acting like CSV files are Comma-Separated Values files. It’s true that CSV stands for that, but it’s also true that other ways of separating values are valid CSV files these days.

People used to call Tab-Separated Values files TSV files, but as other separators grew in popularity everyone realized that creating a new .[a-z]sv file format for every value-separating character used is not sustainable.

So we call all files with a list of different values a CSV file and then use different *delimiters* (like a comma or tab) to indicate where the different values start and stop.

Let’s say we had an address book. Since addresses usually use commas in them, we’ll need to use a different delimiter for our information. Since none of our data has semicolons (;) in them, we can use those.

**addresses.csv**

Name;Address;Telephone  
Donna Smith;126 Orr Corner Suite 857\nEast Michael, LA 54411;906-918-6560  
Aaron Osborn;6965 Miller Station Suite 485\nNorth Michelle, KS 64364;815.039.3661x42816  
Jennifer Barnett;8749 Alicia Vista Apt. 288\nLake Victoriaberg, TN 51094;397-796-4842x451  
Joshua Bryan;20116 Stephanie Stravenue\nWhitneytown, IA 87358;(380)074-6173  
Andrea Jones;558 Melissa Keys Apt. 588\nNorth Teresahaven, WA 63411;+57(8)7795396386  
Victor Williams;725 Gloria Views Suite 628\nEast Scott, IN 38095;768.708.3411x954

Notice the \n character, this is the escape sequence for a new line. The possibility of a new line escaped by a \n character in our data is why we pass the newline='' keyword argument to the open() function.

Also notice that many of these addresses have commas in them! This is okay, we’ll still be able to read it. If we wanted to, say, print out all the addresses in this CSV file we could do the following:

import csv  
  
with open('addresses.csv', newline='') as addresses\_csv:  
  address\_reader = csv.DictReader(addresses\_csv, delimiter=';')  
  for row in address\_reader:  
    print(row['Address'])

Notice that when we call csv.DictReader we pass in the delimiter parameter, which is the string that’s used to delineate separate fields in the CSV. We then iterate through the CSV and print out each of the addresses.

**Instructions**

**1.**

Import the csv module.

Checkpoint 2 Passed

**2.**

Open up the file **books.csv** in the variable books\_csv.

Checkpoint 3 Passed

**3.**

Create a DictReader instance that uses the @ symbol as a delimiter to read books\_csv. Save the result in a variable called books\_reader.

Checkpoint 4 Passed

**4.**

Create a list called isbn\_list, iterate through books\_reader to get the ISBN number of every book in the CSV file. Use the ['ISBN'] key for the dictionary objects passed to it.

**script.py**

import csv

with open('books.csv', newline='') as books\_csv:

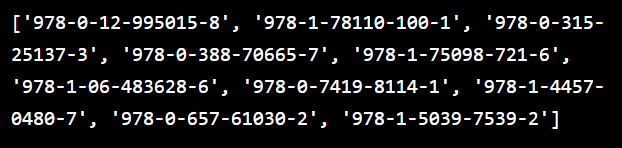
    books\_reader = csv.DictReader(books\_csv, delimiter='@')

    isbn\_list = []

    for row in books\_reader:

      isbn\_list.append(row['ISBN'])

    print(isbn\_list)

****

**Writing a CSV File**

Naturally if we have the ability to read different CSV files we might want to be able to programmatically create CSV files that save output and data that someone could load into their spreadsheet software. Let’s say we have a big list of data that we want to save into a CSV file. We could do the following:

big\_list = [{'name': 'Fredrick Stein', 'userid': 6712359021, 'is\_admin': False}, {'name': 'Wiltmore Denis', 'userid': 2525942, 'is\_admin': False}, {'name': 'Greely Plonk', 'userid': 15890235, 'is\_admin': False}, {'name': 'Dendris Stulo', 'userid': 572189563, 'is\_admin': True}]  
  
import csv  
  
with open('output.csv', 'w') as output\_csv:  
  fields = ['name', 'userid', 'is\_admin']  
  output\_writer = csv.DictWriter(output\_csv, fieldnames=fields)  
  
  output\_writer.writeheader()  
  for item in big\_list:  
    output\_writer.writerow(item)

In our code above we had a set of dictionaries with the same keys for each, a prime candidate for a CSV. We import the csv library, and then open a new CSV file in write-mode by passing the 'w' argument to the open() function.

We then define the fields we’re going to be using into a variable called fields. We then instantiate our CSV writer object and pass two arguments. The first is output\_csv, the file handler object. The second is our list of fields fields which we pass to the keyword parameter fieldnames.

Now that we’ve instantiated our CSV file writer, we can start adding lines to the file itself! First we want the headers, so we call .writeheader() on the writer object. This writes all the fields passed to fieldnames as the first row in our file. Then we iterate through our big\_list of data. Each item in big\_list is a dictionary with each field in fields as the keys. We call output\_writer.writerow() with the item dictionaries which writes each line to the CSV file.

**Instructions**

**1.**

We have a list in the workspace access\_log which is a list of dictionaries we want to write out to a CSV file.

Let’s start by importing the csv module.

Checkpoint 2 Passed

**2.**

Open up the file **logger.csv** in the temporary variable logger\_csv. Don’t forget to open the file in write-mode.

Checkpoint 3 Passed

**3.**

Create a csv.DictWriter instance called log\_writer. Pass logger\_csv as the first argument and then fields as a keyword argument to the keyword fieldnames.

Checkpoint 4 Passed

**4.**

Write the header to log\_writer using the .writeheader() method.

Checkpoint 5 Passed

**5.**

Iterate through the access\_log list and add each element to the CSV using log\_writer.writerow().

**script.py**

access\_log = [{'time': '08:39:37', 'limit': 844404, 'address': '1.227.124.181'}, {'time': '13:13:35', 'limit': 543871, 'address': '198.51.139.193'}, {'time': '19:40:45', 'limit': 3021, 'address': '172.1.254.208'}, {'time': '18:57:16', 'limit': 67031769, 'address': '172.58.247.219'}, {'time': '21:17:13', 'limit': 9083, 'address': '124.144.20.113'}, {'time': '23:34:17', 'limit': 65913, 'address': '203.236.149.220'}, {'time': '13:58:05', 'limit': 1541474, 'address': '192.52.206.76'}, {'time': '10:52:00', 'limit': 11465607, 'address': '104.47.149.93'}, {'time': '14:56:12', 'limit': 109, 'address': '192.31.185.7'}, {'time': '18:56:35', 'limit': 6207, 'address': '2.228.164.197'}]

fields = ['time', 'address', 'limit']

import csv

with open('logger.csv', 'w') as logger\_csv:

  #fields = ['name', 'userid', 'is\_admin']

  log\_writer = csv.DictWriter(logger\_csv, fieldnames=fields)

  log\_writer.writeheader()

  for item in access\_log:

    log\_writer.writerow(item)

**Reading a JSON File**

CSV isn’t the only file format that Python has a built-in library for. We can also use Python’s file tools to read and write [JSON](https://www.codecademy.com/resources/docs/python/json-module?page_ref=catalog). JSON, an abbreviation of JavaScript Object Notation, is a file format inspired by the programming language JavaScript. The name, like CSV is a bit of a misnomer — some JSON is not valid JavaScript (and plenty of JavaScript is not valid JSON).

JSON’s format is endearingly similar to Python dictionary syntax, and so JSON files might be easy to read from a Python developer standpoint. Nonetheless, Python comes with a json package that will help us parse JSON files into actual Python dictionaries. Suppose we have a JSON file like the following:

**purchase\_14781239.json**

{  
  'user': 'ellen\_greg',  
  'action': 'purchase',  
  'item\_id': '14781239',  
}

We would be able to read that in as a Python dictionary with the following code:

**json\_reader.py**

import json  
  
with open('purchase\_14781239.json') as purchase\_json:  
  purchase\_data = json.load(purchase\_json)  
  
print(purchase\_data['user'])  
# Prints 'ellen\_greg'

First we import the json package. We opened the file using our trusty open() command. Since we’re opening it in read-mode we just need to pass the file name. We save the file in the temporary variable purchase\_json.

We continue by parsing purchase\_json using json.load(), creating a Python dictionary out of the file. Saving the results into purchase\_data means we can interact with it. We print out one of the values of the JSON file by keying into the purchase\_data object.

### Instructions

**1.**

Let’s read a JSON file! Start by importing the json module.

Checkpoint 2 Passed

Hint

Import the JSON library with the following command:

import json

**2.**

Open up the file **message.json**, saving the file object to the variable message\_json.

Open the file in read-mode, without passing any additional arguments to open().

Checkpoint 3 Passed

Hint

Remember the syntax for opening a file:

with open('file.json') as file\_json:  
  pass

**3.**

Pass the JSON file object as an argument to json.load() and save the resulting Python dictionary as message.

Checkpoint 4 Passed

**4.**

Print out message['text'].

**script.py**

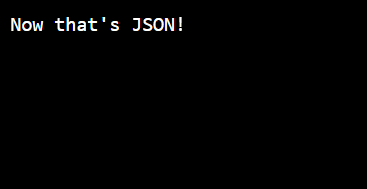
import json

with open('message.json') as message\_json:

  message = json.load(message\_json)

print(message['text'])

# Prints 'ellen\_greg'

****