Worksheet #11A Files Answer Key =)

TRUE/FALSE

1. (10 points: 1 point each) The following are true/false questions. **Write either T or F in the boxes at the bottom of page 1.** If there's any counterexample, it's false.

   (a) Using the `open()` function cannot result in errors when providing the name of a file that exists on your computer.

   (b) Closing a file using `.close()` is important because it ensures that system resources are released and any data remaining in memory buffers is written.

   (c) When reading from a file using `.readline()`, the newline character at the end of each line is included in the returned string.

   (d) Directories themselves do not have file paths, because file paths refer only to the location of files within directories.

   (e) The `.close()` method must be called on files in the same order that they were opened to avoid potential issues with file handling.

   (f) When writing to a file in Python, the new data will always overwrite the original contents of the file.

   (g) In Python, using `open()` on a directory allows you to read or write to all files within that directory.

   (h) When using the `"w"` mode to write to a file, only alphabetical or numerical characters are allowed, and attempting to write other characters will result in an error.

   (i) The `.readlines()` method will always return all lines from a file, regardless of whether the file handle has previously read from the file.

   (j) Files are used to store data persistently, while directories are used to provide a hierarchical structure for files and do not store data themselves.

a	b	c	d	e	f	g	h	i	j
F	T	T	F	F	F	F	F	F	T

Notes:
A: It can! What if the file is in the incorrect directory? If my Python program is on my Desktop, but the file I'm trying to call open() on is in my Downloads folder, then unless I provide an absolute file path it won't be recognized. So simply having the file installed, isn't error-safe.

C: Yes! This is true. Because the newline character is just like any other character. This is why in Project 3, you kiddos needed to use .strip() before calling .split() on the first line of the file (list of city names). Because the file had multiple lines, all but the last had an invisible \n at the end.

D: Windows file path to a directory: C:\parent_directory\subdirectory
Mac file path to a directory: /parent_directory/subdirectory/

File paths simply point us to a specific location in our computer – it's possible that a path points to a folder, so directories can have their own file paths.

E: False! I can do

```
file1 = open("benfordData.txt", "r")
file2 = open("flights.txt", "r")
file2.close()
file1.close()
```

In that order. Despite opening file1 first, I close it last. But this doesn't cause an error.

F: False! If we do file1 = open("benfordData.txt", "a") then we are simply appending to benfordData.txt. So this depends on the mode. Mode "a" will add onto what is already present, mode "w" will overwrite the file entirely.

G: False! We cannot call open() on a directory – this will be an error.

H: False! In homework 11, you kiddos used mode "w" but wrote to the file spaces, newline characters, tabs, and colons.

I: False! It will return the line that are yet to be seen within the file.

MULTIPLE CHOICE

2	3	4	5	6	7	8	9
B	C	B	D	E	C	C	B

2. When reading data from a file, which of the following methods is considered the most memory-efficient for processing large files?

   A. `.read()`

   B. `.readline()`

   C. `.readlines()`

   D. All three methods have similar memory efficiency.

Correct answer explanation:

(B) Yes! Reading a file line-by-line is more memory efficient. This is because at any given time, we're only storing one line of information from the file in memory. .readline() and .readlines() will read and store the entire contents of a file, which requires more space in memory.

3. What is the difference between a relative file path and an absolute file path?

   A. Relative paths always start from the directory where Python is installed, while absolute paths can start from any directory.

   B. Relative paths cannot be used for files in our current working directory, while absolute paths can be used for any file.

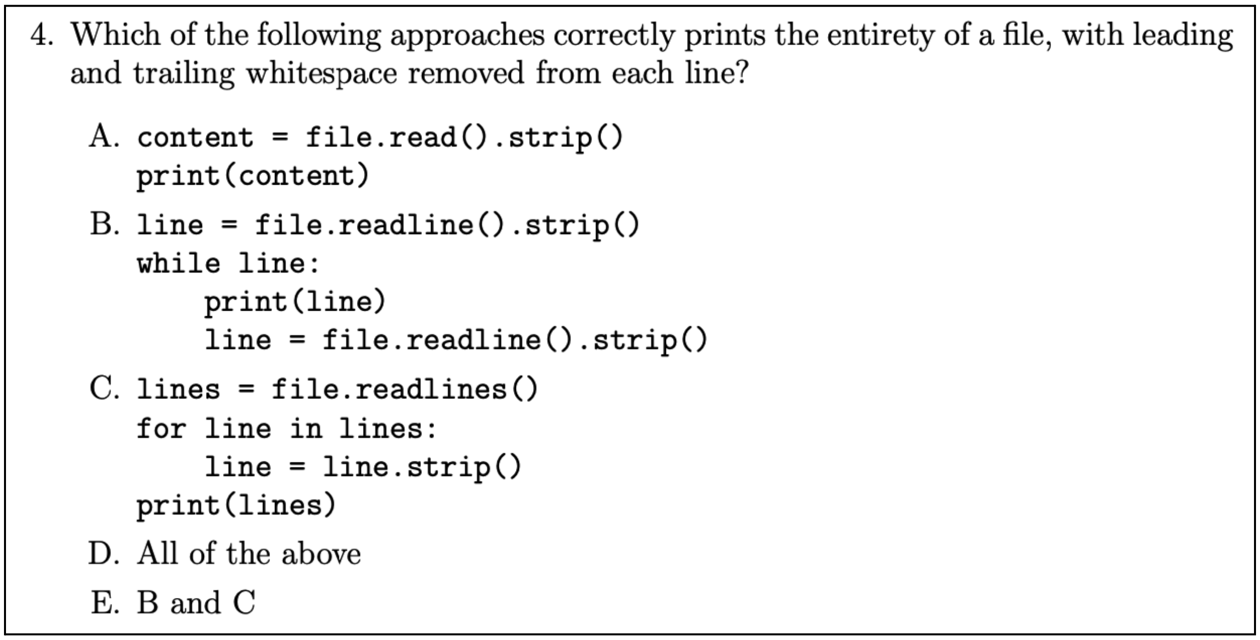
   C. Relative paths specify a file's location based on the current working directory, while absolute paths specify a file's location starting from the top-level (root) directory of the file system.

   D. Relative paths are typically used for specifying input files, whereas absolute paths are exclusively reserved for specifying output files.

Correct answer explanation:

(C) Yes! Dr. Young gave a great example on Ed. Suppose you and Britney Spears live on the street PopStar. You're house 1 and she's house 3. If someone asks you where Britney lives, you can say "she lives two houses down from me." This would be relative – this location depends on where you're at currently, because if you move somewhere else, Britney will no longer be two houses down from you.

Or , you can provide her exact address. 3 PopStars Street. This is absolute. No matter where you move, you can find Britney at that address.



4. Which of the following approaches correctly prints the entirety of a file, with leading and trailing whitespace removed from each line?

A.
```
content = file.read().strip()
print(content)
```

B.
```
line = file.readline().strip()
while line:
    print(line)
    line = file.readline().strip()
```

C.
```
lines = file.readlines()
for line in lines:
    line = line.strip()
print(lines)
```

D. All of the above

E. B and C

Correct answer explanation:

(B) Yes!! We get the first line of the file, with whitespace removed, by calling .readline().strip(). Then we do a 'while line' loop. Recall that .readline() returns a string, so here we are using a string as our while loop condition. And a string will evaluate to True when it is non-empty, and False when it is empty. So, this loop will persist until 'line' is empty. And within this loop, we print line, and then set line to be the next line in the file, by calling file.readline().strip(). At this point, the loop repeats, so we print line again, then set line to the next in the file…

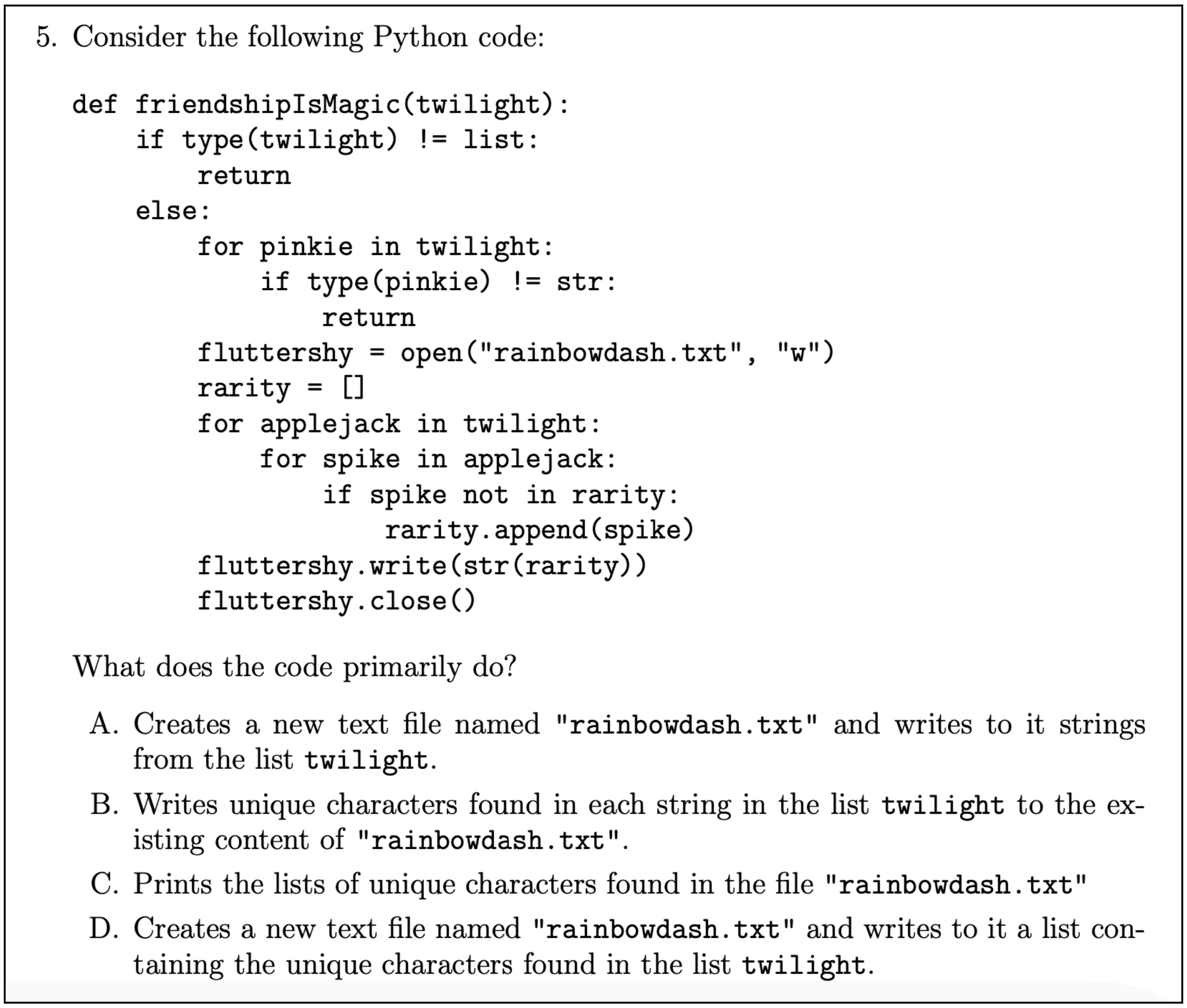
Wrong answer explanations:

(A) Close! But while .read() returns the entirety of a file's text, it's as a single string. .strip() will only remove whitespace at the start and ends of a string, so doing .read().strip() only removes the whitespace at the start and end of our file text, and /not/ at the start and end of each line within the file.

(C) CLOSE!! What's going on here is: we call .readlines() on our file pointer, storing the result in 'lines.' So, our variable 'lines' is a list of lines from the file. When we do 'for line in lines:' as our for loop, though, note that with this kind of for loop, 'line' is a local variable. That is, any modification to 'line' won't affect the source. So, we set line = line.strip(), but that isn't actually modifying the list. So, when we print(lines) after, the items in our list can still have whitespace.

(D) because A and C are incorrect, D cannot be correct.

(E) because C is incorrect, E cannot be correct.



5. Consider the following Python code:

```
def friendshipIsMagic(twilight):
    if type(twilight) != list:
        return
    else:
        for pinkie in twilight:
            if type(pinkie) != str:
                return
        fluttershy = open("rainbowdash.txt", "w")
        rarity = []
        for applejack in twilight:
            for spike in applejack:
                if spike not in rarity:
                    rarity.append(spike)
        fluttershy.write(str(rarity))
        fluttershy.close()
```

What does the code primarily do?

A. Creates a new text file named `"rainbowdash.txt"` and writes to it strings from the list `twilight`.

B. Writes unique characters found in each string in the list `twilight` to the existing content of `"rainbowdash.txt"`.

C. Prints the lists of unique characters found in the file `"rainbowdash.txt"`

D. Creates a new text file named `"rainbowdash.txt"` and writes to it a list containing the unique characters found in the list `twilight`.

Correct answer explanation:

(D) We first check whether our parameter is a list. If it is, then we check whether each item in the list is a string. If so, then we create a new file (rainbowdash.txt). And we create an empty list (rarity).

We then iterate over our list twilight, referring to each string as applejack. And we iterate over applejack, meaning spike will refer to the characters in each string. We check whether spike is already in our list. If it is not, then we add it to the list.

This means that we're simply scanning through all the characters present in the strings in our list. And we're adding unique characters to our list. At the end, we write the list itself to the file.

<u>Wrong answer explanations:</u>
(A) Close! But because we do

```
for applejack in twilight:
        for spike in applejack:
```

Each iteration of the loop, applejack will refer to a string in twilight, and spike refers to the characters in applejack. So we're looking at individual characters. But (A) is considering only strings as a whole.

(B) CLOSE! But note that we did

```
fluttershy = open("rainbowdash.txt", "w")
```

Mode "w" means we are overwriting the contents of rainbowdash.txt completely. If we wanted to add to the existing content, we would need mode "a". Additionally, (B) says 'unique characters found in each string' but the code is writing unique characters found in the list of strings as a whole. That is to say, if our list is ["pop", "pip"], (B) suggest that we would write p, o, p, i (unique characters in each string). But our code only considers unique characters as a whole, so it will write p only once (not twice).

(C) For what it's worth, there actually isn't a single print statement within this bit of code. How can we print, if we aren't using a print()? Teehee. =) But, also – note we opened rainbowdash.txt in mode "w". This means that we cannot read from rainbowdash.txt, so it would be impossible for us to determine what characters are in the file.

6. Which of the following are differences between `print()` and `.write()` in Python?

   A. `.write()` is for writing to files, while `print()` is only for console output.
   B. `.write()` only accepts strings, while `print()` can handle various data types.
   C. `print()` automatically appends a newline character at the end, while `.write()` does not.
   D. `print()` accepts multiple parameters, while `.write()` only accepts one.
   E. All of the above.
   F. A and C

Correct answer explanation:
(E) Yes! All of A, B, C, and D are true. print() will display to the console (to the program itself), while .write() is for writing to a file. .write() also only accepts strings, but print() will implicitly convert its parameters to strings. (ex: we can do print(1))

print() also automatically appends newline characters – if we do

```
print(2)
print(5)
```

Then our output will be

```
2
5
```

But if we try doing

```
myFile.write(“Silly”)
myFile.write(“goose”)
```

Then we won’t get

```
Silly
goose
```

Instead, we’ll get

```
Sillygoose
```

And print accepts multiple parameters – we can do print(“Silly”, “goose”). But we can’t do myFile.write(“Silly”, “goose”). We need to concatenate to get one string that we write – myFile.write(“Silly” + “goose”).

7. Which of the following is false regarding files in Python?

   A. You can have a file open for reading and writing at the same time.
   B. You can call `open()` on the same file twice before calling `close()`.
   C. When reading from a file in `"r+"` mode, numerical characters will automatically be converted to integers.
   D. When you use `open()` on one file within Python, you do not need to call `close()` on the file before you use `open()` on a different file.

Correct answer explanation:
(C) Nope! When reading from a file, the data will always be interpreted as strings in Python. .read(), .readlines(), .readline() all return strings. The mode is only to set permissions, such as reading/writing.

Wrong answer choices:
(A) “r+” mode lets you read from and write to a file.
(B) Yes! This is possible. You’ll just have two separate pointers to the file. And ‘advancing’ one (by calling .read() or .readlines() or .readline()) does not advance the other.
(D) This is correct. I can call open() on two separate files, before I close either of them. I could potentially read from ten different files at once, and then close them all at the very end of my program.

8. Consider the following approach to reading a file's contents in Python:

```
wall-e = open("eve.txt", "r")
m_o = wall-e.readline()
while m_o:
    m_o = wall-e.readline()
```

Why does the above approach work?

A. `.readline()` will return `False` when it hits the end of a file, so the condition `while m_o` will evaluate to False, and the loop will terminate.

B. The file handle `wall-e` recognizes when we read all lines from the file `eve.txt` and will end the loop itself.

C. `.readline()` will return the empty string `""` when it hits the end of a file, so the condition `while m_o` will evaluate to `False`, and the loop will terminate.

D. The file system automatically appends a special character to indicate the end of a file, so the condition `while m_o` will recognize this marker and terminate the loop.

Correct answer explanation:
(C) This is correct! When .readline() hits the end of the file, it will return an empty string (because there was nothing left to read). So our condition will be equivalent to 'while "" ', which is False. Thus, the loop will terminate.

Wrong answer explanations:
(A) Close, but .readline() doesn't return False. .readline() will always return a string, either the line within the file or the empty string.
(B) Not true! The file handle simply acts as a connection to the file on our computer. It will not automatically recognize anything for us, as (B) is stating.
(D) Not true! Files can end with any character – there is no universal character that indicates the end of a file. And even if there was, it would still be a character in Python – so, 'while m_o' would evaluate to True (because the only string that evaluates to False is the empty string), so the while loop would still continue. This is very error prone!

9. After opening a file in "a" mode, what happens if you attempt to read from the file?

   A. The file is read without any issue.
   B. An error is raised.
   C. The file is read, but only the new content added in "a" mode is retrieved.
   D. Reading the file in "a" mode results in an empty string.

Correct answer explanation:
(B) Yes! If we're in "a" mode, that means we are specifically appending to the file. We can't write to the file but also read from the file in this mode. This would require either mode "w+" or "r+".

Wrong answer explanations:
(A), (C) Reading from the file in mode "a" (and also in mode "w") will cause an error, different modes grant us different permissions.
(D) This answer is kind of silly. Teehee. I don't know what else to say, other than it's wrong! =) LOL

TRACING

10. (3 points)

```
muppets = open("hungergames.txt", "r")
gonzo = muppets.readline()
kermit = []
while gonzo:
    msPiggy = gonzo.split(" ")
    fozzie = msPiggy[-1]
    kermit.append(fozzie)
    gonzo = muppets.readline()
muppets.close()
print(kermit)
```

['primrose\n', 'haymitch\n', 'rue']

This is a fun question! I love the Muppets!

OK. We create a file handler called 'muppets' by calling open("hungergames.txt", "r"). We're in read mode, specified by "r". We call .readline() and store the result (the first line of the file, which is “katniss peeta primrose\n”) in gonzo. We then create an empty list, called kermit. And then we begin a loop.

This loop, because we’re using gonzo, a string variable, as our condition, will continue as long as gonzo is non-empty. Our string was not empty, so this loop activates. msPiggy splits gonzo by spaces, and creates a list of the substrings. So, missPiggy will be

[“katniss”, “peeta”, “primrose\n”]

fozzie uses negative indexing (-1) to get the last item of this list. This would be “primrose\n”. We append this to our empty list. Then, we set gonzo to be the next line in the file, by doing gonzo = muppets.readline().

Our loop then repeats. So, the next line in the file is “finnick effie haymitch\n”. When we split and get the last item, it’ll be “haymitch\n”. We add this to the list.

And the line after this in the file is “joanna cinna rue”. There is no line after rue in our file (not even an empty line), so there is no \n at the end of rue. Thus, when we split and get the last item of this line, it’ll just be “rue”. Add it to the list.

We now close our file handler, and print our list. It’ll be [“primrose\n”, “haymitch\n”, “rue”].

11. (3 points)

```
batman = open("barney.txt", "r")
deadpool = batman.readline().strip()
blackWidow = []
superman = ""
while deadpool:
    spiderman = deadpool.split()
    scarletWitch, thanos = spiderman[0], spiderman[1]
    blackWidow += [scarletWitch, thanos]
    superman += blackWidow[-1]
    deadpool = batman.readline().strip()
batman.close()
print(superman)
```

246

OOO!! A lot going on in this question!

OK – we open barney.txt, and make an empty list, create a variable named superman (empty string). Then we get the first line of the file by doing deadpool = batman.readline().strip(). And then we start a while loop, ‘while deadpool’, which will persist as long as deadpool is non-empty.

Within this loop, we split our current line. Note the structure of each line in barney.txt, each line is

num1 num2 word1 word2 word3
(a number, a space, a number, a space, some strings)
(ex: 1 2 buckle my shoe)

So if we split each line, recall that .split(someString) will create from the string a list, where each of the items are the parts of the string separated by someString. But since we simply called .split(), then the default behavior is to split on any and all whitespace. So each iteration of the loop, spiderman will be something like

[#, #, word, word, word]
(ex: [“1”, “2”, “buckle”, “my”, “shoe])

When we hit scarletWitch, thanos = spiderman[0], spiderman[1], we’re simply doing a multi-assignment. scarletWitch will contain the item at index 0 of spiderman, and thanos will contain the item at index 1. But we know that these are both numbers (although, string representations still – remember that in Python when we read from a file, the content is returned as a string).

We add those two ‘numbers’ to our blackWidow list. And then we add the current last item of blackWidow to superman. When processing the “1 2 buckle my shoe” line, this means we add “1” and “2” to blackWidow. So, “2” is currently blackWidow’s last item. Thus, superman += blackWidow[-1] means we add “2” to superman.

Then we proceed in the file. So our loop will repeat these steps until we’ve read the whole file. That is to say, the next line, “3 4 shut the door”, we’ll add “4” to superman. Then the final line, “5 6 pick up sticks”, we’ll add “6” to superman.

We then close the file and print superman, which will be “246”.

12. (3 points)

```
ghostface = open("hungergames.txt", "r")
myers = ghostface.readlines()
for krueger in range(len(myers)):
    chucky = myers[krueger].split()
    if krueger == 0:
        for friday13 in chucky:
            print(friday13, end = " ")
    print(chucky[0], end = " ")
ghostface.close()
```

katniss peeta primrose katniss finnick joanna

OK, Halloween!!! Except that it's April. LOL

OK. We open the hungergames file, in mode "r", so we're reading and begin at the start of the file. We call .readlines(), and store the result in myers. So, myers is a list of all our lines in the file. For visual purposes, myers would be

["katniss peeta primrose\n", "finnick effie haymitch\n", "joanna cinna rue"]

We then start a loop: for krueger in range(len(myers)). So, this goes over the length of our list. Our list has length 3, so our range is 0, 1, and 2.

We create a variable called chucky: chucky = myers[krueger].split(). So, we index into our list to get a line, and then we make a list out of that line. For example: the first iteration, krueger is 0, so myers[krueger] will be myers[0], or "katniss peeta primrose\n". When we call .split(), this becomes ["katniss", "peeta", "primrose"]. That's what chucky is set to the first iteration.

We then check if krueger == 0. On the first iteration, this is true. So, our 'if' activates, and we begin another loop: for friday13 in chucky. This is simply iterating over chucky (which is ["katniss", "peeta", "primrose"]), referring to the current item as friday13. We then print friday13, so we're essentially printing this list, item at a time. Thus, katniss peeta primrose.

Then, we print chucky[0], which is the first item in the list. The first iteration, this would be katniss.

The loop then repeats. But because krueger is only 0 on the first iteration, the 'if' won't activate again. So, the second iteration, we're looking at the second line in our list ""finnick effie haymitch\n", when we repeat the above steps, we'll print finnick. On the third iteration, we're looking at the line "joanna cinna rue", repeat the above steps and we'll print joanna.

13. (3 points)

```
monstersINC = open("barney.txt", "w")
monstersINC.write("BOO!")
monstersINC.close()
sully = open("barney.txt")
mike = sully.readline()
sully.close()
print(mike)
```

BOO!

I cried during this movie!

OK – we start by creating a file handle named "monstersINC" and opening the file "barney.txt" in Python. Since we've opened it in "w" mode, it means we're going to completely overwrite any existing content in the file. If we wanted to append to the end of "barney.txt," we would use "a" mode.

Thus, the previous content in "barney.txt" is wiped out. In this code, all we do is write "BOO!" to the file. So when we reopen the file after closing it and use .readline(), we retrieve the first and only line in the file, which is "BOO!" We then close the file and print "BOO!"

For a monster-themed question, this wasn't all that scary!!

14. (3 points)

```
stevenConnie = open("hungergames.txt", "a")
stevenConnie.write("\ncrystalgems\n")
stevenConnie.close()
garnet = open("hungergames.txt", "r")
pearl = garnet.read().split()
garnet.close()
amethyst = pearl[-1]
print(amethyst)
```

crystalgems

We open the hungergames file, in mode “a”, which opens the file for reading and places the ‘pointer’ at the end of it. So, when we write “\ncrystalgems\n”, we add a newline character, which means we start a new line in our file. Then comes “crystalgems”. And \n follows this, we start another new line. We close this file handler.

And we open a new one, called garnet! This time, strictly reading “r”. Mode “r” opens the file for reading and places the pointer at the start. We read the entirety of this file by doing

pearl = garnet.read().split()

.read() returns our entire file contents, as a string. .split() will create a list of the substrings in our file separated by whitespace. Our file data is

“katniss peeta primrose\nfinnick effie haymitch\njoanna cinna rue\ncrystalgems\n”

Splitting this on whitespace, our list is

[“katniss”, “peeta”, “primrose”, “finnick”, “effie”, “haymitch”, “joanna”, “cinna”, “rue”, “crystalgems”]

We close garnet, and access the last item of our list by negative indexing (-1). And we print this item, which is “crystalgems”.

15. (3 points)

```
peridot = open("hungergames.txt", "r")
lapis = open("hungergames.txt", "r")
peridot.readline()
peridot.readline()
lapis.readline()
print(lapis.readline())
lapis.close()
peridot.close()
```

finnick annie haymitch

I wanted to confuse you kiddos by having multiple file handles point to the same file! But they act independently. Imagine two 'pointers' in the file, and that's peridot and lapis. When we do peridot.readline(), we move peridot further. And when we do lapis.readline(), we move lapis further.

We call .readline() on peridot twice, so it is now pointing at the last line in the file (that's what it would return if we called .readline() on peridot again).

We call lapis.readline() once, before printing(lapis.readline()). So, lapis will return the second line in the file, which is 'finnick annie haymitch'.

16. (3 points)

```
peterpan = open("barney.txt")
tinkerbell = peterpan.readline()
peterpan.close()
peterpan.close()
print(tinkerbell)
```

1 2 buckle my shoe

Alright – here, we open the file barney.txt and grab its first line ("1 2 buckle my shoe") and store it in tinkerbell. I wanted to scare you kiddos by attempting to call .close() twice on a file handle. But it doesn't raise an error. So, we're able to successfully print the first line in the file.

17. (3 points)

```
plankton = open("barney.txt")
karen = plankton.read()
plankton.close()
mrsPuff = [0] * 10
for larry in karen.split():
    pearl = len(larry)
    mrsPuff[pearl] += 1
print(mrsPuff)
```

[0, 6, 2, 1, 4, 0, 2, 0, 0, 0]

This question is neat!!! Wow. I feel like I say that about a lot of the questions. LOL

OK, so we're reading in the entirety of the barney text file here. We store this jumbo string in karen. We create a list – [0] * 10 will create the list [0, 0, 0, 0, 0, 0, 0, 0, 0, 0] (list of 10 zeroes)

And we do

for larry in karen.split()

Remember, .split() gets us a list, separating pieces of string upon any whitespace – so karen.split() will be a list where each item is text from barney.txt that has whitespace before/after it. So, karen.split() will just be a list of each word from our text file:

['1', '2', 'buckle', 'my', 'shoe', '3', '4', 'shut', 'the', 'door', '5', '6', 'pick', 'up', 'sticks']

So, we're essentially looping over this list, referring to the string we're currently at as larry. We then obtain the length of that string. And we increment the corresponding index in our list.

For example, len('1') is 1. So we increment index 1. Then we move onto '2'. len('2') is '1', so we increment index 1. Then we move onto 'buckle' which has length 6. So we increment index

6. Below is a table for each word of the file, its length (index we increment in our list), and our list after we make the update.

Current word	Length (and index to increment)	Resulting List
1	1	[0, **<u>1</u>**, 0, 0, 0, 0, 0, 0, 0, 0]
2	1	[0, **<u>2</u>**, 0, 0, 0, 0, 0, 0, 0, 0]
buckle	6	[0, 2, 0, 0, 0, 0, **<u>1</u>**, 0, 0, 0]
my	2	[0, 2, **<u>1</u>**, 0, 0, 0, 1, 0, 0, 0]
shoe	4	[0, 2, 1, 0, **<u>1</u>**, 0, 1, 0, 0, 0]
3	1	[0, **<u>3</u>**, 1, 0, 1, 0, 1, 0, 0, 0]
4	1	[0, **<u>4</u>**, 1, 0, 1, 0, 1, 0, 0, 0]
shut	4	[0, 4, 1, 0, **<u>2</u>**, 0, 1, 0, 0, 0]
the	3	[0, 4, 1, **<u>1</u>**, 2, 0, 1, 0, 0, 0]
door	4	[0, 4, 1, 1, **<u>3</u>**, 0, 1, 0, 0, 0]
5	1	[0, **<u>5</u>**, 1, 1, 3, 0, 1, 0, 0, 0]
6	1	[0, **<u>6</u>**, 1, 1, 1, 0, 1, 0, 0, 0]
pick	4	[0, 6, 1, 1, **<u>4</u>**, 0, 1, 0, 0, 0]
up	2	[0, 6, **<u>2</u>**, 1, 4, 0, 1, 0, 0, 0]
sticks	6	[0, 6, 2, 1, 4, 0, **<u>2</u>**, 0, 0, 0]