FILES AND COMMAND LINE ARGUMENTS

WARMUP: COUNTING MULTIPLE THINGS WITH LISTS

What is the final value of `counts` that is printed? Describe what this snippet does. **L1** (Yes, you can have emojis in Python strings.)

```
message = "🍏🍌🍒🍍🍒🍏🍍🍒🍍"
counts = [0, 0, 0]
for fruit in message:
    if fruit == "🍏":
        counts[0] += 1
    elif fruit == "🍌":
        counts[1] += 1
    elif fruit == "🍒":
        counts[2] += 1
print(counts)
```

This snippet fills in the `counts` list with the number of times each of the following appear in a string: `"🍏"`, `"🍌"`, and `"🍒"`

```
message = "🍏🍌🍒🍍🍒🍏🍍🍒🍍"
counts = [0, 0, 0]
for fruit in message:
    if fruit == "🍏":
        counts[0] += 1
    elif fruit == "🍌":
        counts[1] += 1
    elif fruit == "🍒":
        counts[2] += 1
print(counts)
```

We're counting three things, so initialize a list with three different count values, each starting at 0. (**Why 0?**)

This snippet fills in the `counts` list with the number of times each of the following appear in a string: "🍏", "🍌", and "🍒"

```
message = "🍏🍌🍒🍍🍒🍏🍍🍒🍍"
counts = [0, 0, 0]
for fruit in message:
    if fruit == "🍏":
        counts[0] += 1
    elif fruit == "🍌":
        counts[1] += 1
    elif fruit == "🍒":
        counts[2] += 1
print(counts)
```

Go character by character, where each emoji is a character...

This snippet fills in the `counts` list with the number of times each of the following appear in a string: `"🍎"`, `"🍌"`, and `"🍒"`

```
message = "🍎🍌🍒🍍🍒🍎🍍🍒🍍"
counts = [0, 0, 0]
for fruit in message:
    if fruit == "🍎":
        counts[0] += 1
    elif fruit == "🍌":
        counts[1] += 1
    elif fruit == "🍒":
        counts[2] += 1
print(counts)
```

If we see an apple, increment `counts[0]`—we're counting apples with the number saved at index `0`.

This snippet fills in the `counts` list with the number of times each of the following appear in a string: `"🍏"`, `"🍌"`, and `"🍒"`

```
message = "🍏🍌🍒🍍🍒🍏🍍🍒🍍"
counts = [0, 0, 0]
for fruit in message:
    if fruit == "🍏":
        counts[0] += 1
    elif fruit == "🍌":
        counts[1] += 1
    elif fruit == "🍒":
        counts[2] += 1
print(counts)
```

Which index does the banana counter live at? The cherry counter?

FILES

FILES

On computers we have things called **files**.
Files are where we store information that the computer can still access even after the computer turns off and on again.

We have already use files before: our programs are stored in `.py` files.

For now, we can assume that the contents of files are all characters. For text files like these, we think of files as being made of a "sequence" of lines of text.

```
Is there anybody     # first line
                     # second line
out there?           # third line
```

OPEN() AND CLOSE()

To read a file, we need to create a file "object" associated with that file. We can create a variable holding a file object with the `open()` call.

```
# opens the file "filename.txt" with "r" (Reading) enabled
example_file = open("filename.txt", "r")
```

When we are completely done with a file, we need to close it

```
example_file.close()
```

...but what do we do in between the opening and closing?

READLINE()

Once we have an open file object, we can use `readline()` to read a line from the file. TRY IT!

```
# on website as first_three_lines.py
my_file = open("beep.boop", "r")    # open beep.boop for reading
for i in range(3):
    line = my_file.readline()
    print(line)
my_file.close()
```

The next time we call `readline()` we get the next line of the file. These File objects remembers our position in the file.

PRACTICE (C1)

Assume we have a file named `beep.boop` with the layout:

```
this file has 3 lines after this
line 0
line 1
line 2
```

Write some code that can read a file like this and print out all the lines but the first. *Write your solution so that it'd work if file had any number, not just* `3`.

- You should probably use: `open(filename, "r")`, `file.readline()`, `file.close()`, `string.strip()`, `string.split()`

COMMAND LINE ARGUMENTS

COMMAND LINE ARGS

When we run a program we usually type something like

```
python my_program.py
```

We can also specify some information when we run the program through **Command Line Arguments**:

```
python greeting.py Harry
```

The stuff written after `python greeting.py` is available to the program immediately when we start it!

SYS.ARGV

We can `import sys` and use `sys.argv` to get command line args as a *list of strings*.

```
import sys
print(sys.argv)
```

Run with:

```
python args.py Ta lie Ma ssa chi
```

prints: `['args.py', 'Ta', 'lie', 'Ma', 'ssa', 'chi']`

Note: argv has EVERYTHING after `python` in the commnad. Also note that it is a list of strings, if we want other types we have to explicitly convert.

COMMAND LINE ARGS

Consider we have the following program called `greeting.py`

```
import sys

print(f"Hello, {sys.argv[1]}!")
```

If we run it with the command: `python greeting.py Harry`, then:

- `sys.argv` is the list `["greeting.py", "Harry"]`
- `sys.argv[1]` is `"Harry"`

And so the printed output is: `Hello, Harry!`

COMMAND LINE ARGS

Consider we have the following program called `greeting.py`

```
import sys
print(f"Hello, {sys.argv[1]}!")
```

What if we just did `python greeting.py`?

```
Traceback (most recent call last):
  File "/some_path/greeting_argv.py", line 3, in <module>
    print("Hello, " + sys.argv[1] + "!")
IndexError: list index out of range
```

☠️ No CLA at position `1` ☠️

INPUT

READING USER INPUT

- `input()` function for getting text interaction
  - Like `print()`, any string placed in the parentheses will be first displayed as a prompt.
  - Whatever the user types is saved for later inside of `name`

READING USER INPUT: EXAMPLE

hello_input.py

```
print("Who would you like to say hello to?")
name = input(">")
print("Hello, ", name)
```

RUNNING HELLO_INPUT.PY

First execution:

```
$ python hello_input.py
Who would you like to say hello to?
> Agustina
Hello, Agustina
```

Next execution, different name:

```
$ python hello_input.py
Who would you like to say hello to?
> Carson
Hello, Carson
```

COMMAND LINE ARGUMENTS VS. INPUT

- **Command line arguments** are passed into the program at the moment that the program starts
- `input()` is a function called while the program is running
  - stops the program until some input is provided

MAD LIBS

Mad Libs books contain short stories on each page with many key words replaced with blanks. Beneath each blank is specified a category, such as "noun", "verb", "number", "celebrity", "exclamation" or "part of the body". One player asks the other players, in turn, to contribute a word of the specified type for each blank, but without revealing the context for that word. Finally, the completed story is read aloud. The result is usually a comedic sentence.

Stern and Price's original Mad Libs book gives the following sentence as an example:[12]

"_____!" he said _____ as he jumped into his convertible _____ and drove off with his _____ wife.

exclamation *adverb* *noun* *adjective*

After completion, they demonstrate that the sentence might read:

```
 "Ouch!" he said stupidly as he jumped into his convertible cat and drove off with his
brave wife.
```

An illustrative example: `madlibs.py`

- Takes a single command line argument as the name of the Mad Lib file to read
- Uses `input()` to ask the person running the program for words to fill in.

WARMING UP TO FUNCTIONS

SOME FACTS ABOUT FUNCTIONS

- Functions have names, which we use to call them.
- Functions have inputs, which we use to:
  - control the specific way that they behave on different calls
  - process different information with them on different calls
- Functions can, but do not have to!, return values.

```
result = max(17, 34)
```

`max` is the name of a function. We can call it like `max(17, 34)`, which passes in `17` and `34` as its two inputs. The value returned by this function call is `34`, which is saved in `result`.

FUNCTION CALLS: RETURN VALUE REPLACING CALL

An **expression**, as we know, is a bit of Python that has a value.

- `3` is the expression with the value of `3`
- `3 + 7` is the expression with the value of `10`
- `max(7, 11)` is an expression with a value of `11`

When you write a function call as part of an expression, that function call gets **evaluated** in order to determine the full value of the expression.

- Function calls are evaluated inside-out: evaluate the inputs, then call the function with those evaluated inputs.
- This also applies to nested function calls.

L2 Write the values printed by these snippets with function calls.

```
# 1
print(len("hello"))

#2
print(max(3 + 4, 10 - 2))

#3
message = "pYTHon rUles".upper()
words = message.split()
print(words)

#4
print("pYTHon rUles".upper().split())

#5
print(min(7, max(len("vessir"), 12 // 5)))
```