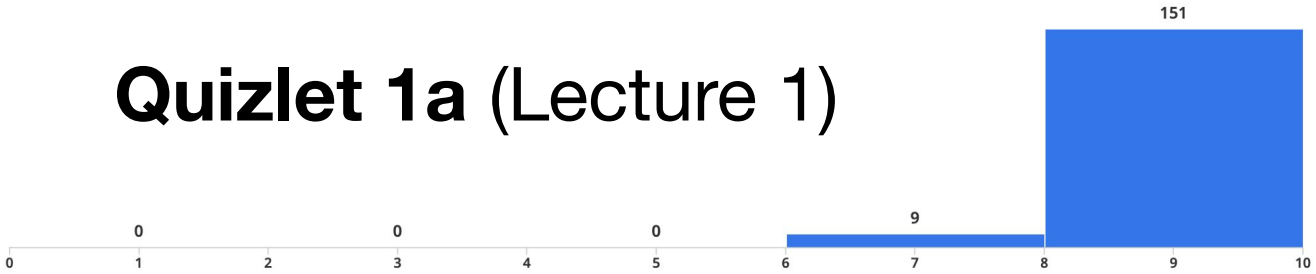
Lists and Methods

15-110 – Friday 09/20

Announcements

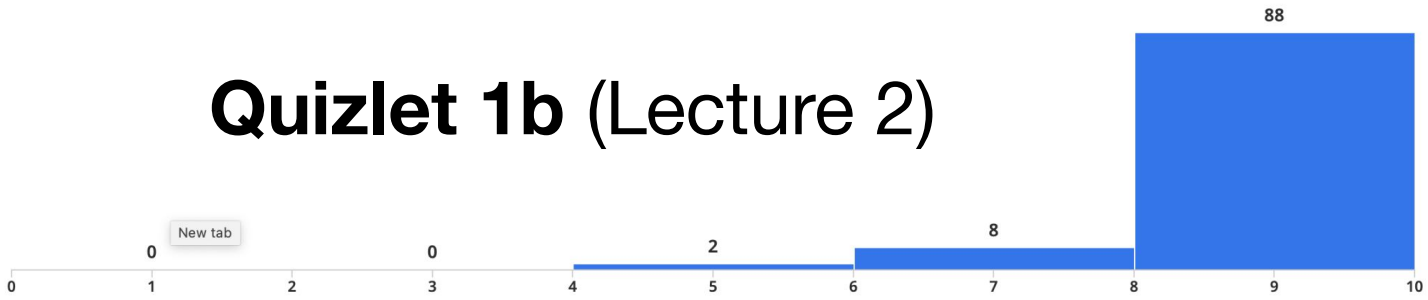
- Hw2 due Monday at noon
- Quizlet1 grades released
- Past quizlet questions and answers posted on the assessments page of the course website
- We are updated gradebook in Canvas

Quizlet 1a (Lecture 1)



Minimum	Median	Maximum	Mean	Std Dev
6.0	10.0	10.0	9.48	1.07

Quizlet 1b (Lecture 2)



Minimum	Median	Maximum	Mean	Std Dev
4.0	10.0	10.0	9.09	1.41

Reminder

As we are progressing into the semester and working with more complex topics in written and programming checks/homework, we want to remind you that you should NOT be using any Generative AI (ChatGPT, Co-pilot, etc.) for your solutions.

Using these tools to solve assignment (checks and homework, written and programming) problems is an academic integrity violation. Do not enter assignment prompts or solutions into these tools.

What to do when you are stuck instead of going to GenAI? We have multiple resources that can help you including Piazza and OH!! Further, you can always submit partial work and/or use the revision deadline. Submitting YOUR work, at any stage, is always better for your learning!

Quizlet 1

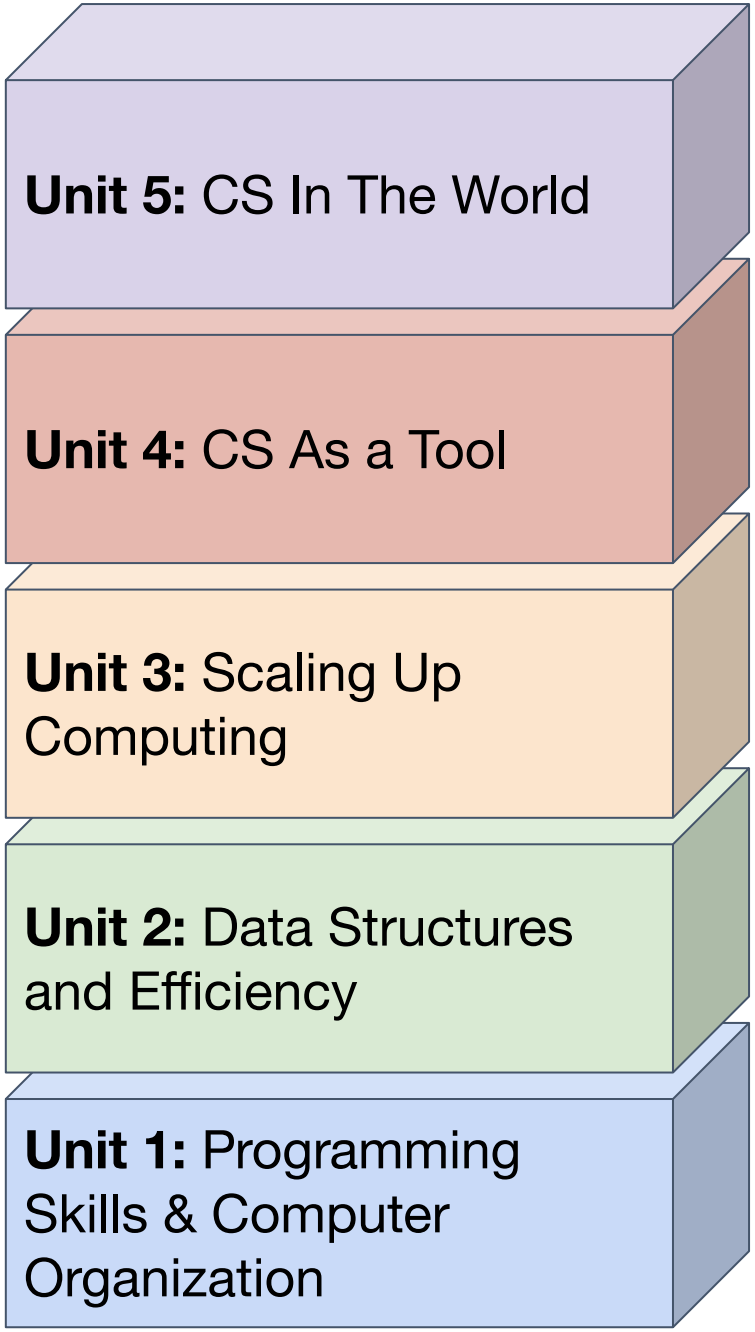
Solutions posted on the course website!

Activity: Practice Code Tracing

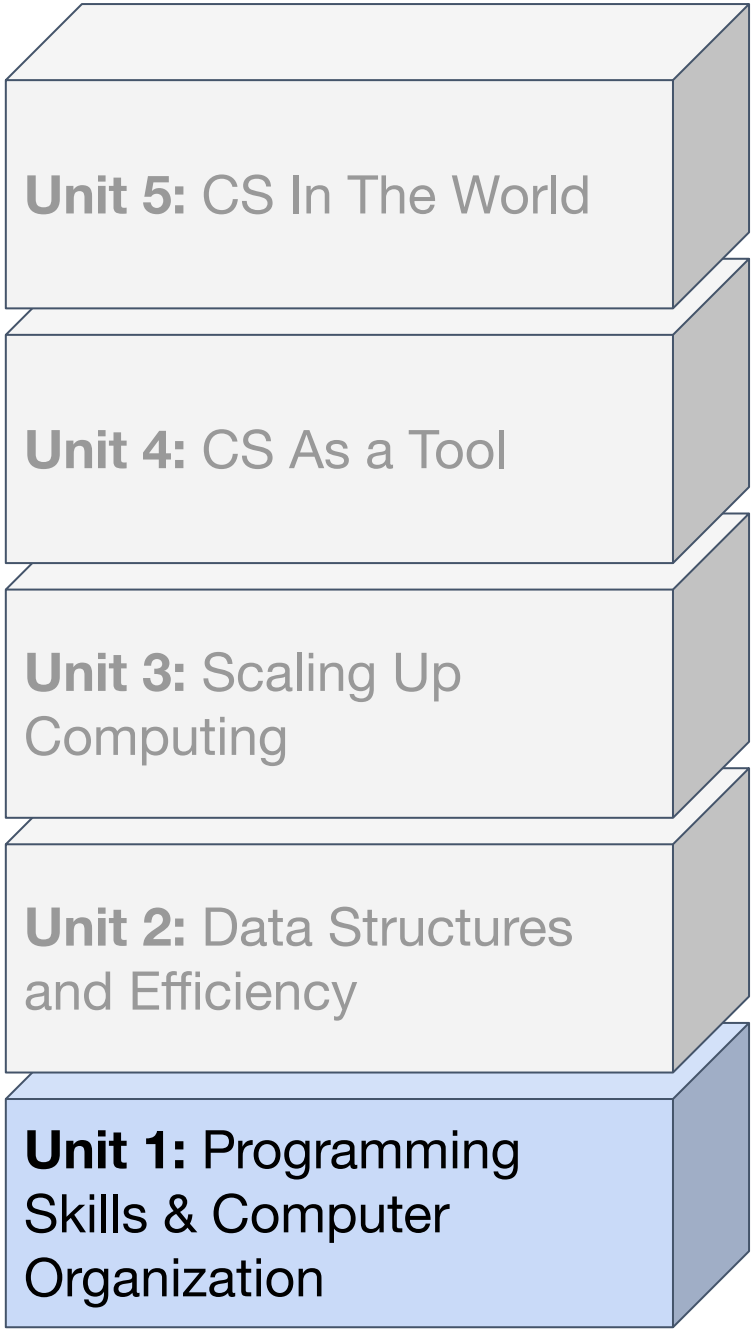
Similar example from Quizlet 1 on the course website here:

```
1  def outer(x):
2      y = x + 2
3      print("outer y:", y)
4      return inner(y) * 2
5
6  def inner(x):
7      y = x * 2
8      print("inner y:", y)
9      return y
10
11 print(outer(4))
```

We’ve finished Unit 1!

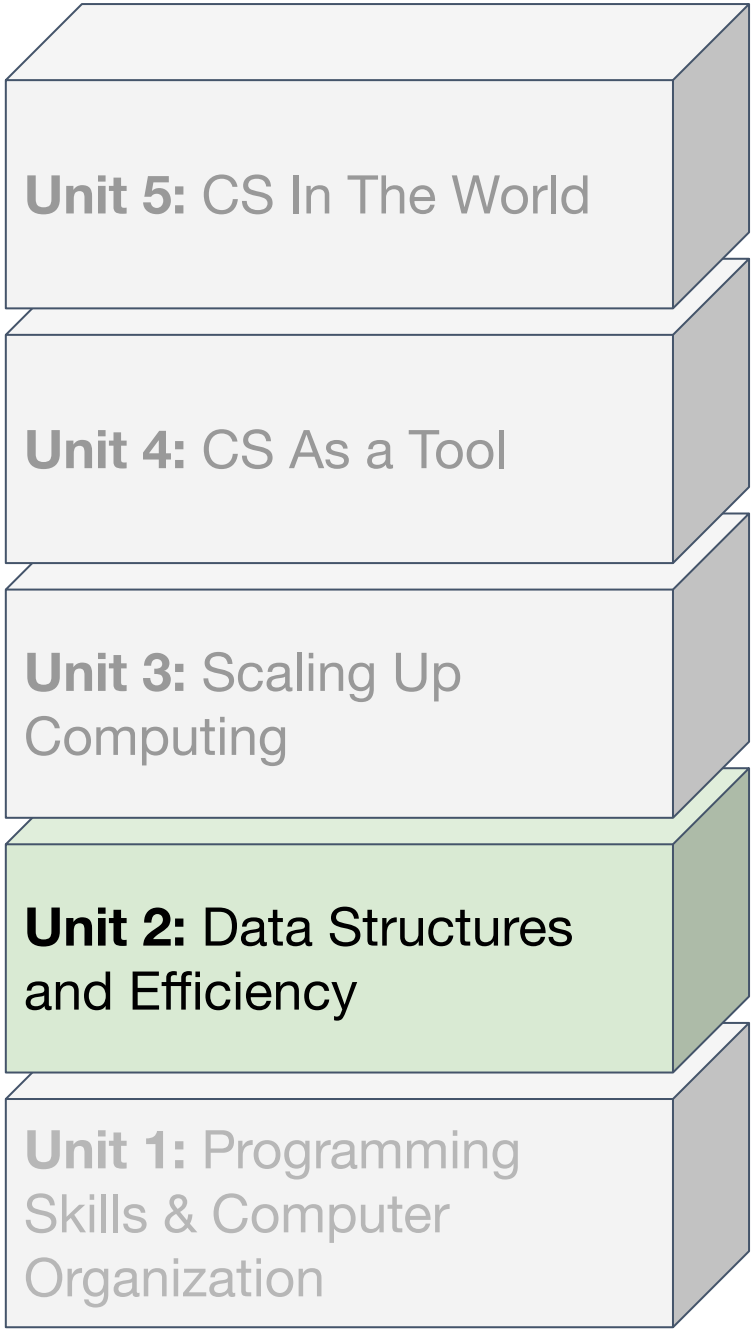


The topics we cover in this course build on each other!



Exam 1 covers Unit 1: [Topics Lists]

- Algorithms and Abstraction
- Programming Basics
- Data Representation
- Function Calls
- Function Definitions
- Booleans, Conditionals, and Errors
- Circuits and Gates
- While & For Loops
- String Indexing, Slicing, and Looping



Unit 2 Topics:

Data Structures: things we use while programming to organize multiple pieces of data in different ways.

- lists, dictionaries, trees, graphs

Efficiency: the study of how to design algorithms that run quickly, by minimizing the number of actions taken

- search algorithms, Big-O, tractability

Learning Goals

- Read and write code using **1D** and **2D lists**
- Use string/list **methods** to call functions directly on values

String Methods

A **method** is different from a built-in function, **it belongs to the object** and has the syntax `object.method()`.

We have seen an example of this with **tkinter**:

`create_rectangle` is a method **called on** canvas, which is a data structure.

```
canvas.create_rectangle(50,50,100,100)
```

Instead of `isdigit(s)` we write `s.isdigit()`.

Instead of `islower(s)` we write `s.islower()`.

You do not need to **memorize** all of these methods!

There is a whole library of built-in string and list methods that have already been written; you can find them at:

- docs.python.org/3/library/stdtypes.html#string-methods
- docs.python.org/3/tutorial/datastructures.html#more-on-lists

Use the **Python documentation** to lookup the name of functions, instead of trying to memorize all of them.

Some methods **return information** about the string.

`s.isdigit()`, `s.islower()`, and `s.isupper()` return `True` if the string is all-digits, all-lowercase, or all-uppercase, respectively.

```
s = "HELLO"
s.islower() # False
s.isupper() # True

s = "12345"
s.isdigit() # True
```

Some methods **return information** about the string.

`s.count(x)` return the number of times the subpart `x` occurs in `s`.

```
s = "heLlo"
s.count("l") # 1
```

`s.index(x)` return the index of the subpart `x` in `s`, or raise an error if it doesn't occur in the value.

```
s.index("o") # 4
s.index(“H”) # ValueError!
```

Example: Check if a string starts with a capital letter and the rest are lowercase.

```
def formalName(s):
    return s[0].isupper() and s[1:].islower()
```

Some methods **return a new string.**

`s.lower()` and `s.upper()` return a new string that is like the original, but all-lowercase or all-uppercase, respectively.

```
s = "heLlo"
a = s.lower() # a = "hello"
b = s.upper() # b = "HELLO"
c = "OK".lower()  # c = "ok"
```

Some methods **return a new string.**

`s.replace(a, b)` returns a new string where all instances of the string a have been replaced with the string `b`.

```
s = "Hello"
c = s.replace("l", "y") # c = "Heyyo"
```

`s.strip()` returns a new string with excess whitespace (spaces, tabs, newlines) at the front and back removed.

```
d = "   Hi there  ".strip() # d = "Hi there"
```

Example: Make a password-generating function

```
def makePassword(phrase):
    phrase2 = phrase.lower()
    phrase3 = phrase2.replace("a","@").replace("o", "0")
    return phrase3
```

Lists

A **list** is a data structure that holds an **ordered collection of data values**.

Example: a sign-in sheet for a class.

Lists make it possible for us to assemble and analyze a collection of data using **only one variable**.

```
signInSheet = ['Elena','Max','James','Iyla','Ayaan']
```

We use **square brackets** to create lists in Python.

```
a = [ ] # empty list
b = [ "uno", "dos", "tres" ] # list with three strings
c = [5, 8, 1, 10] # list with 4 numbers
d = [ 1, "dance", 4.5 ] # lists can have mixed types
```

We can use **operators with lists** in a similar way as we can with strings.

Like strings, lists can be **concatenated** using addition operator `+`:

```
>> [ 1, 2 ] + [ 3, 4 ]
[ 1, 2, 3, 4]
```

Like strings, lists can be **repeated** using multiplication operator `*`:

```
>> [ "a", "b" ] * 2
[ "a", "b", "a", "b" ]
```

Like strings, we can **index**, **slice**, **get the length**, and **check membership** with lists.

```
lst = [ "a", "b", "c", "d" ]
lst[1] # indexing - "b"
lst[2:] # slicing - [ "c", "d" ]
"c" in lst # membership - True
"abc" in lst # membership - False
```

There are **built-in functions** we can also use with lists.

```
len(lst) # length of a list
min(lst) # smallest element of the list
max(lst) # biggest element of the list
sum(lst) # total sum of elements in the list

random.choice(lst) # picks a random element from the list
```

Activity: Evaluate the Code

You do: what will each of the following code snippets evaluate to?

```
[ 5 ] * 3
```

```
["a", "b", "c"][1]
```

```
min([5, 1, 8, 2])
```

We can use a **for loop over the indexes** of the list to access each item.

Algorithm: Sum the elements in a list `prices`.

```
total = 0
for i in range(len(prices)):
    total = total + prices[i]
print(total)
```

Algorithm: Write a function that finds the **maximum value in a list** of strings.

```
def findMax(nums):
    biggest = nums[0] # why not 0? Negative numbers!
    for i in range(len(nums)):
        if nums[i] > biggest:
            biggest = nums[i]
    return biggest
```

Some methods **return information** about lists.

`lst.count(x)` return the number of times the element `x` occurs in `lst`.

```
lst = [10, 20, 30, 40, 50]
lst.count(20) # 1
```

`lst.index(x)` return the index of the element `x` in `lst`, or raise an error if it doesn't occur in the in `lst`.

```
lst.index(50) #4
lst.index(5) # ValueError!
```

Some methods **convert strings into lists and lists to strings**.

`s.split(c)` splits up a string into a list of strings based on the separator character, `c`.

```
e = "one,two,three".split(",")
# e = [ "one", "two", "three" ]
```

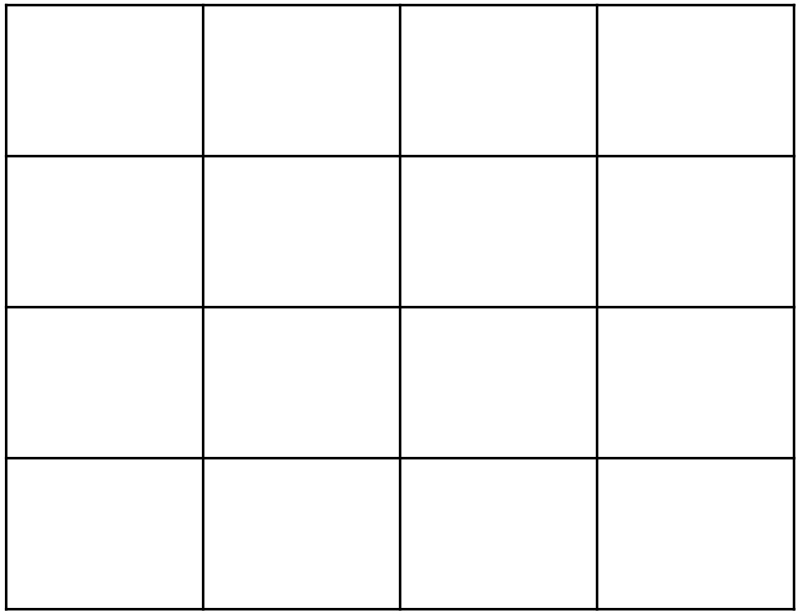
`c.join(lst)` joins a list of strings together into a single string, with the string `c` between each pair.

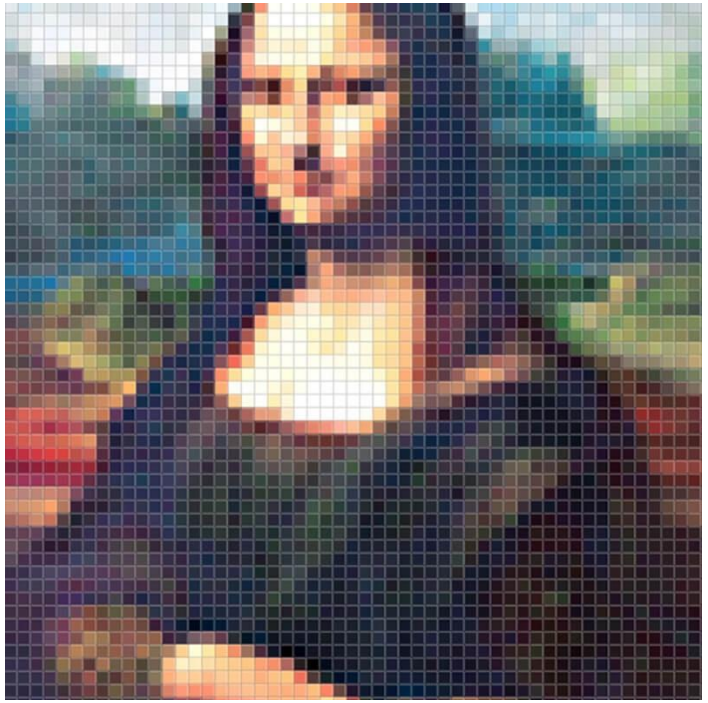
```
f = "-".join(["ab", "cd", "ef"])
# f = "ab-cd-ef"
```

2D Lists

A **2D** list is a list where the items it contains are lists.

We often need to work with data that is **two-dimensional**, such as the coordinates on a grid, values in a spreadsheet, or pixels on a screen.





2D List Example

The table below shows cities in Pennsylvania, the county they're in, and their population.

City	County	Population
Pittsburgh	Allegheny	302,407
Philadelphia	Philadelphia	1,584,981
Allentown	Lehigh	123,838
Erie	Erie	97,639
Scranton	Lackawanna	77,182

In Python, we could represent this using the 2D list to the right. Each of the five elements of the list **is itself a list**!

Population List

```
0. 0. "Pittsburgh"
   1. "Allegheny"
   2. 302407
1. 0. "Philadelphia"
   1. "Philadelphia"
   2. 1584981
2. 0. "Allentown"
   1. "Lehigh"
   2. 123838
3. 0. "Erie"
   1. "Erie"
   2. 97639
4. 0. "Scranton"
   1. "Lackawanna"
   2. 77182
```

Setting up a 2D list still uses square brackets, where each **inner list is one data value**.

```
cities = [ ["Pittsburgh", "Allegheny", 302407],
           ["Philadelphia", "Philadelphia", 1584981],
           ["Allentown", "Lehigh", 123838],
           ["Erie", "Erie", 97639],
           ["Scranton", "Lackawanna", 77182] ]
```

This is across multiple lines but treated as one line because each part ends with a comma.

The **length of a 2D list** is the number of lists in the outer list.

```
len(cities)  # 5
```

When indexing into a 2D list, the **first square brackets index into a row** and the **second index into a column**.

```
cities = [ ["Pittsburgh", "Allegheny", 302407],
           ["Philadelphia", "Philadelphia", 1584981],
           ["Allentown", "Lehigh", 123838],
           ["Erie", "Erie", 97639],
           ["Scranton", "Lackawanna", 77182] ]


cities # the entire lists
cities[2] # ["Allentown", "Lehigh", 123838]
cities[2][1] # "Lehigh"
```

When using **looping over a 2D list** with one loop, indexing into the list will produce the **inner list**.

```
def getCounty(cities, cityName):
    for i in range(len(cities)):
        entry = cities[i] # entry is a list
        if entry[0] == cityName:
            return entry[1]
    return None # city not found
```

To loop over every element of a 2D list, we need to use **nested for loops**.

```
gameBoard = [ ["X", " ", "O"], [" ", "X", " "], [" ", " ", "O"] ]
for row in range(len(gameBoard)): # each row is a list
    boardString = ""
    for col in range(len(gameBoard[row])): # each col is a string
        boardString = boardString + gameBoard[row][col]
    print(boardString) # separate rows on separate lines
```

Activity: `getTotalPopulation(cities)`

Fill in the blanks for the function `getTotalPopulation(cities)` that takes the city-information 2D list from before and finds the total population of all cities in the list.

```
def getTotalPopulation(cities):
    __________ = 0
    for row in range(__________):
        pop = __________
        total = __________
    return total
```

Hint: note that the population is in the third column. Which index corresponds to that?

[if time] Activity: `getFirstName(fullName)`

You do: write the function `getFirstName(fullName)`, which takes a string holding a full name (in the format `"Farnam Jahanian"`) and returns just the first name. You can assume the first name will either be one word or will be hyphenated (like `"Soo-Hyun Kim"`).

You'll want to use a **method** and/or an **operation** in order to isolate the first name from the rest of the string.

Learning Goals

- Read and write code using **1D** and **2D lists**
- Use string/list **methods** to call functions directly on values