#3-2: Strings and Lists II

CS SCHOLARS – PROGRAMMING

Learning Goals

Use **for loops** to loop over strings/lists by **component**

Use string/list **methods** to call functions directly on values

Read and write code using **2D lists**

Looping over Sequences

Looping Over Sequences Directly

If we don't care about where values are located in a sequence, we don't need to use a `range` in the `for` loop. We can loop over the parts of a sequence directly by providing the value instead of a `range`.

```
for <itemVariable> in <sequenceValue>:
    <itemActionBody>
```

For example, if we run the following code, it will print out each string in the list with an exclamation point after it.

```
wordlist = [ "Hello", "World" ]
for word in wordList:
    print(word + "!")
```

Example: Looping over Strings

Another example – how do we count the number of exclamation points in a string? We don't need the indexes, so we can loop over the string directly.

```
s = "Wow!! This is so! exciting!!!"
count = 0
for c in s:
    if c == "!":
        count = count + 1
print(count) # 6
```

Choosing Loops

How do you decide whether to loop over a range or loop over the value directly? Think about whether you need to know **where** the parts are located in the sequence.

For example – what if you wanted to loop over a pair of lists at the same time? Use a single for-range loop and **share** the loop control variable between the two lists.

```
def printPhonebook(names, numbers):
    for i in range(len(names)):
        print(names[i], ":", numbers[i])
```

Activity: `findMax(lst)`

Write a function `findMax(lst)` which takes a list of numbers and returns the largest number in the list.

Hint: consider what **variables** you'll need to keep track of, and what type of **loop** you should use.

Methods

Methods Are Called Differently

Most string and list built-in functions (and data structure functions in general) work differently from other built-in functions. Instead of writing:

```
isdigit(s)
```

write:

```
s.isdigit()
```

This tells Python to call the built-in string function `isdigit` **on the string** `s`. It will then return a result normally. We call this kind of function a **method**, because it belongs to a **data structure**.

This is how our Tkinter methods work too! `create_rectangle` is called **on** `canvas`, which is a data structure.

Don't Memorize- Use the API!

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

and

docs.python.org/3/tutorial/datastructures.html#more-on-lists

We're about to go over a whole lot of potentially useful methods, and it will be hard to memorize all of them. Instead, **use the Python documentation** to look for the name of a function that you know probably exists.

If you can remember which basic actions have already been written, you can always look up the name and parameters when you need them.

Some Methods Return Information

Some methods return information about the value.

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

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

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

```
s = "hello"
lst = [10, 20, 30, 40, 50]


s.isdigit() # False
s.islower() # True
"OK".isupper() # True


s.count("l") # 2
lst.count(20) # 1


s.index("o") # 4
lst.index(5) # ValueError!
```

Example: Checking a String

As an example of how to use methods, let's write a function that returns whether or not a string holds a capitalized name. The first letter of the name must be uppercase and the rest must be lowercase.

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

Activity: Evaluate the Code

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

```
"Yay".islower()
```

```
lst = [4, 8, 10, 8, 6, 4]
lst.count(4)
lst.index(4)
```

Some Methods Create New Values

Other methods return a new value based on the original.

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

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

`s.split(c)` returns a list that has split up the string based on the separator character, `c`.

```
s = "Hello"

a = s.lower() # a = "hello"
b = s.upper() # b = "HELLO"

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

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

Example: Making New Strings

We can use these new methods to make a silly password-generating function.

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

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.

Some Methods Change the Value

Finally, there are some methods that let us **change the value itself**, without reassigning the variable. We call these **destructive** methods.

`lst.append(item)` destructively adds a new item to the end of a list.

`lst.remove(item)` destructively removes the given item from a list once.

`lst.sort()` destructively sorts the list by comparing all the elements.

Note that the function calls do not return a new list; the original list is changed instead. That means we typically call the method **by itself** and do not assign the result to a variable.

```
lst = [ 1, 2, "a" ]


lst.append("b")

# lst = [ 1, 2, "a", "b" ]


lst.remove("a")

# lst = [ 1, 2, "b" ]


lst = [ 40, 2, 13, 7 ]

lst.sort()

# lst = [ 2, 7, 13, 40]
```

Example: `getFactors(n)`

Let's write a function that takes an integer and returns a list of all the factors of that integer.

```
def getFactors(n):
    factors = [ ]
    for num in range(1, n+1): # n is a possible factor
        if n % num == 0:
            factors.append(num)
    return factors
```

Example: `removeEvens(lst)`

Now let's write a function that takes a list of numbers and removes any numbers that are not even.

```
def removeEvens(lst):
    i = 0
    while i < len(lst):
        if lst[i] % 2 == 0:
            lst.remove(lst[i])
        else:
            # only update i when we don't remove!
            i = i + 1
    return lst
```

Activity: `longWordsOnly(words)`

Write a function `longWordsOnly(words)` that takes a list of words (strings) and returns a new list that only contains words that were longer than 4 characters.

For example, `longWordsOnly(["What", "a", "fabulous", "day", "it", "is", "today"])` would return `["fabulous", "today"]`.

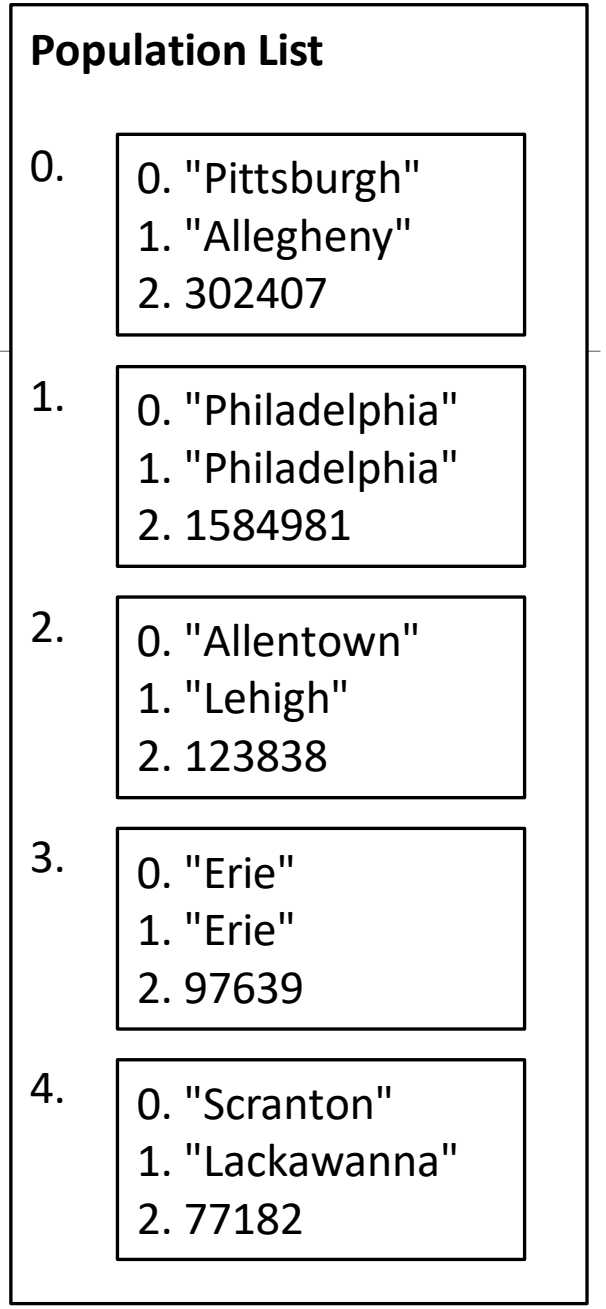
Try using the **append** method to set up the new list instead of using list concatenation!

2D Lists

2D Lists are Lists of 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. We can store this type of data in a **2D list**, which is just a list that contains other lists.

For example, the 2D list to the right holds population data, where each population entry itself contains multiple data values (city, county, and population).



Syntax of 2D Lists

Setting up a 2D list is no different than setting up a 1D list; 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.

When indexing into a 2D list, the first square brackets index into a row and the second index into a column. The length of a 2D list is the number of lists inside the outer list.

```
cities[2]    # [ "Allentown", "Lehigh", 123838 ]
cities[2][1] # "Lehigh"
len(cities)  # 5
```

Activity: Evaluate the Code

Consider the following 2D list that represents a word search board

```
a = [ [ "t", "y", "r", "t" ],
      [ "d", "a", "a", "s" ],
      [ "o", "c", "l", "y" ],
      [ "g", "e", "z", "f" ] ]
```

What is the index of the **row** the two `"a"`'s are on?

What is the index of the **column** the `"r"` is in?

What expression would let you access the letter `"c"`?

Looping Over 2D Lists

We can loop over a 2D list the same way we loop over a list. Indexing into a list once will produce an **inner list**. We'll need to index a second time to get a value.

```
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
```

Looping Over All 2D List Elements

When you loop over a 2D list and want to access *every* element, you need to use **nested for loops**. Often, the outer loop iterates over the indexes of the outer list (**rows**) and the inner loop iterates over the indexes of the inner list (**columns**).

```
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 slide 23 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?

Learning Goals

Use **for loops** to loop over strings/lists by **component**

Use string/list **methods** to call functions directly on values

Read and write code using **2D lists**