Lists and Methods

15-110 – Friday 02/11

Announcements

- Hw2 due Monday at noon
- Quiz1 grades will be released Monday

Learning Goals

- Read and write code using **1D** and **2D lists**

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

Unit 2 Overview

Unit 2: Data Structures and Efficiency

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

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

These concepts are **connected**, as we often design data structures so that specific tasks have efficient algorithms.

Unit 2 Topic Breakdown

Data Structures: lists, dictionaries, trees, graphs

Efficiency: search algorithms, Big-O, tractability

Lists

Lists are Containers for Data

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

Example: a sign-in sheet for a class.

Sign In Here
0. Elena
1. Max
2. Eduardo
3. Iyla
4. Ayaan

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

List Syntax

We use **square brackets** to set up a list in Python.

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

Core List/String Operations

Lists share most of their core operations with strings. You can **concatenate** lists together, just like strings.

```
[ 1, 2 ] + [ 3, 4 ] # concatenation – [ 1, 2, 3, 4]
"ABC" + "DEF" # concatenation - "ABCDEF"
```

And you can also **repeat** lists an integer number of times. This works for strings too!

```
[ "a", "b" ] * 2 # repetition – [ "a", "b", "a", "b" ]
"HA" * 3 # repetition - "HAHAHA"
```

We learned about **indexing** and **slicing** last time- those work on lists too.

```
lst = [ "a", "b", "c", "d" ]
lst[1] # indexing – "b"
lst[2:] # slicing – [ "c", "d" ]
```

Looping Over Lists

Looping over lists works the same way as with strings. We can use a for loop over the indexes of the list to access each item. For example, the following loop sums all the values in `prices`.

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

Example: `findMax(nums)`

Let's write a function that finds the maximum value in a list of integers.

```
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
```

We'll often use this algorithmic structure to find the biggest/best item in a structure.

Membership Checks

Recall that strings can be broken down into individual parts (**characters**). The same is true of lists, which can be broken into individual values. Data types that can be broken down into parts in an ordered fashion are called **sequences**.

We can use a special operator called `in` to see whether an individual part occurs in a sequence data type. This returns a Boolean.

```
"e" in "Hello" # True
"W" in "CRAZY" # False
4 in [ "a", "b", 1, 2 ] # False
```

Sidebar: Built-in List/String Functions

There are some new built-in functions we'll want to use with lists and/or strings.

```
len(s) # length of a string/list

ord(c) # ASCII number of a character
chr(x) # character associated with the ASCII number

min(lst) # min element of the list
max(lst) # max 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" in "easy"
```

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

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 datapoint itself contains multiple data values (city, county, and population).

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

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 in the outer list.

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

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
```

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()
```

Some Methods Create New Values

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

```
s = "Hello"


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

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

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

Some Methods Change Data Types

Finally, some methods let you convert between strings and lists as needed.

`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"
```

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
```

[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 **list methods** to change lists without variable assignment

Feedback: https://bit.ly/110-s22-feedback