Lists

15-110 – Monday 09/28

Learning Goals

- Read and write code using **1D** and **2D lists**
- Use **list methods** to change lists without variable assignment

Unit 2 Overview

Unit 2: Data Structures and Efficiency

Data Structures: things we use while programming to organize 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, sort algorithms, tractability

Lists

Lists are Containers for Data

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

Example: a sign-in sheet for a class.

Sign In Here

1. Elena
2. Max
3. Eduardo
4. Iyla
5. 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 3 strings

c = [ 1, "woof", 4.5 ] # lists can have mixed types
```

Basic List Operations

Lists share most of their basic operations with strings.

```
a = [ 1, 2 ] + [ 3, 4 ] # concatenation – [ 1, 2, 3, 4]
b = [ "a", "b" ] * 3 # repetition – [ "a", "b", "a", "b", "a", "b" ]
c = [ 1, 3 ] < [ 2, 4 ] # comparison/equality – True
d = 4 in [ "a", "b", 1, 2 ] # membership – False
```

This includes **indexing** and **slicing**.

```
items = [ "a", "b", "c", "d" ]
print(items[1])  # indexing – prints "b"
print(items[2:]) # slicing – prints [ "c", "d" ]
```

Singleton Lists Are Just Like Any Other List

`3` is an int

`[3]` is a list containing a single int

`3 * 5` is `15`

`[3] * 5` is `[3, 3, 3, 3, 3]`

What is `[3] + [5]` ?

`len(3)` gives an error

`len([3])` is `1`

List Subscripting vs. Slicing Is Not Like Strings

List Subscripting / Slicing

```
t = ["a", "b", "c", "d", "e"]
```

`t[2]` is `"c"` # result is not a list

`t[2:3]` is `["c"]` not `"c"`

`t[2:4]` is `["c", "d"]`

String Subscripting / Slicing

```
s = "abcde"
```

`s[2]` is `"c"` # result is a string

`s[2:3]` is also `"c"`

`s[2:4]` is `"cd"`

List Functions and Methods

There are a few useful built-in functions that work directly on lists.

```
len(lst) # the number of elements in lst
min(lst) # the smallest element in lst
max(lst) # the largest element in lst
sum(lst) # the sum of the elements in lst
```

There are also some list methods, which are called directly on the list, like string methods.

```
lst.count(element) # the number of times element occurs in lst
lst.index(element) # the first index of element in lst
```

Looping Over List Indices

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

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

New: Looping Over List Elements

A **for** loop can also loop over the elements of a list directly instead of looping over the indices. Often this is more convenient.

```
total = 0
for item in prices:
    total = total + item
```

This kind of for loop can also be used on strings, by looping over each character in turn.

Compare Two Looping Styles

Loop over indices

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

Loop over elements

```
total = 0
for item in prices:
    total = total + item
```

Better solution

Activity: Predict the Result

What will be printed after each of the following code snippets?

```
stuff = ["a", "b", 1, 2]

print(stuff[1]) # Q1

s = ""
for x in stuff:
    s = s + str(x)
print(s) # Q2
```

Get First Odd: Return the Number

Loop over indices

```
def getFirstOdd(nums):
  for i in range(len(nums)):
    if nums[i] % 2 == 1:
      return nums[i]
```

Loop over elements

```
def getFirstOdd(nums):
  for n in nums:
    if n % 2 == 1:
      return n
```

Better solution

Find First Odd: Return the Index

Loop over indices

```
def findFirstOdd(nums):
  for i in range(len(nums)):
    if nums[i] % 2 == 1:
      return i
```

Better solution

Loop over elements

```
def findFirstOdd(nums):
  index = 0
  for n in nums:
    if n % 2 == 1:
      return index
    index = index + 1
```

Example: `findMax(lst)`

Let's write a function that finds the maximum value in a list of integers (without using the built-in `max` function).

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

Use `s.split(c)` to Turn Strings Into Lists

We'll also use a new string method, `s.split(c)`, to split up a string into a new list based on a separator character, `c`.

```
def findName(sentence, name):
    words = sentence.split(" ")
    for w in words:
        if w == name:
            return True
    return False

findName("Ask Tom to phone Nina", "Tom")
```

List Methods

Some List Methods Change the List

Sometimes we want to modify a list directly, to add or remove elements from it. There are a set of list methods that can do this without using variable assignment at all.

```
lst = [ 1, 2, "a" ]
lst.append("b") # adds the element to the end of the list
```

Note that we do not set `lst = lst.append`; the list is changed **in place**. In fact, the `append` method returns `None`, not a list. We'll talk more about how this works next time.

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 factor in range(1, n+1):
        if n % factor == 0:
            factors.append(factor)
    return factors
```

Additional List Methods

Here are a few other useful list methods that change the list in place:

```
lst = [ 1, 2, "a" ]
lst.insert(1, "foo") # inserts the 2nd param into the 1st index
lst.remove("a") # removes the given element from the list once
lst.pop(0) # removes the element at given index from the list
```

Sidebar: Don't Change List Length in For Loops

It is a **very bad idea** to remove elements from a list while looping over it with a for loop.

This will often lead to unexpected and bad behavior, since the range is only calculated once.

```
lst = ["a", "a", "c", "d", "e"]
for i in range(len(lst)):
    if lst[i] == "a" or \
       lst[i] == "e":
        lst.pop(i)
```

Instead, use a **while loop** if you're planning to change the length of the list. The list length is reevaluated when the while condition is checked each iteration.

```
lst = ["a", "a", "c", "d", "e"]
i = 0
while i < len(lst):
    if lst[i] == "a" or \
       lst[i] == "e":
        lst.pop(i)
    else:
        i = i + 1
```

Subscripted Assignment

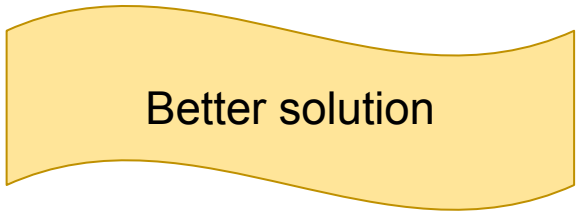
We can also directly update an element in a list based on its index:

```
fruits = ["apple", "pear", "cherry"]
print(fruits)          #  ["apple", "pear", "cherry"]

fruits[1] = "banana"

print(fruits)          # ["apple", "banana", "cherry"]
```

Double Every Number

Loop over indices

```
for i in range(len(lst)):
  lst[i] = lst[i] * 2
```



Loop over elements

```
index = 0
for num in lst:
  lst[index] = num * 2
  index = index + 1
```

2D Lists

Lists Can Contain Other Lists

```
pairs = [["H", 1], ["Li", 3], ["Na", 11], ["K", 19]]

for item in pairs:
  print(item[0] + "\t" + str(item[1]))
```

```
H       1
Li      3
Na      11
K       19
```

Lists Can Contain Other Lists

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

Lists of Lists

```
len(cities)          # 5
cities[2]            # ["Allentown", "Lehigh", 123838]
cities[2][0]         # "Allentown"
cities[0][2]         # 302407
cities[2][1]         # "Lehigh"

def getCounty(city):
    for entry in cities:
        if entry[0] == city:
            return entry[1]
```

Example: `getTotalPopulation(cityList)`

Given a list of city-info-lists, where the 3rd element in each city-info-list is the population of the city, find the total population of all the cities in the list.

```
def getTotalPopulation(cityList):
    total = 0
    for cityEntry in cityList:
        total = total + cityEntry[2]
    return total
```

Looping with 2D Lists

Sometimes you may need to loop over **both** dimensions of a 2D list, to access all the elements in the list. The outer loop goes over the list-of-lists, while the inner loop goes over the elements in each inner list.

```
gameBoard = [ ["X", " ", "O"], [" ", "X", " "], [" ", " ", "O"] ]
boardString = ""
for row in gameBoard:
    for entry in row:
        boardString = boardString + entry # entry is a string
    boardString = boardString + "\n" # separate rows
```

Looping with 2D Lists - Elements vs Indexes

You can loop over a range instead of the 2D list directly, usually if you want to access elements by their index. The list is indexed first by the variable of the outer loop (to get a list), then by the variable of the inner loop (to get an element).

```
s = ""
for row in gameBoard:
  for entry in row:
    s = s + entry
  s = s + "\n"
```

```
s = ""
for i in range(len(gameBoard)):
  for j in range(len(gameBoard[i])):
    s = s + gameBoard[i][j]
  s = s + "\n"
```

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-feedback**