Class 5 - Lists Part II

MISE Summer Programming Camp 2023

Class Project Announcement!

By the end of the week we will post instructions on piazza how the class project will work, along with a companion video you should watch

Recall that the class project is optional but it is encouraged: you'll have fun and it's a great learning opportunity

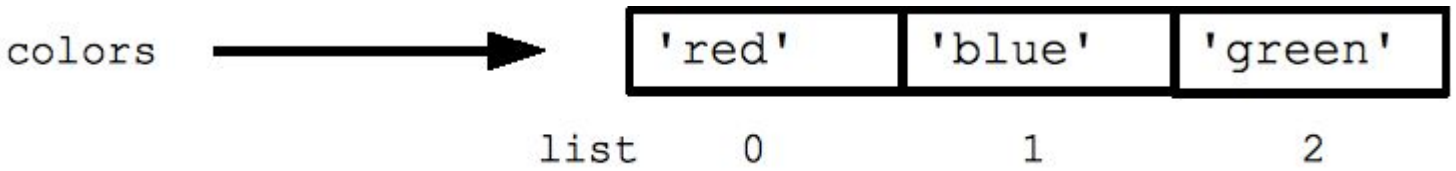
Recap of Class 4

- Nested loops
  - Loops within loops.
  - Useful for printing more complex structures (such as grids).
- Lists
  - Storing collections of data.
  - len() function.
  - Mutability - we can add and replace elements of a list.
  - Tuples - immutable.

A reminder on lists:

```
empty_list = []

colors = ["red", "blue", "green"]
numbers = [1, 2, 3, 4, 5]
```




```
colors[0] # red
colors[1] # blue
colors[2] # green
```

```
numbers[0] # 1
numbers[1] # 2
numbers[2] # 3
```

References and Lists

List References

What's the output of the following code:

```
1  a = 1
2  b = a
3  a = 2
4  print(b)
```

```
1  a = [1]
2  b = a
3  a[0] = 2
4  print(b)
```

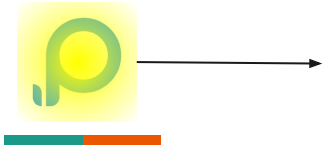
```
1  a = [1]
2  b = a
3  a = [2]
4  print(b)
```

```
colors = ["red", "blue", "green"]
b = colors
```

Copying a list

To fix the problem from the previous slide we can “copy” a list, which means creating a distinct clone of the original list.

```
1  a = [1]
2  b = list(a) # Creates a copy of the list
3  b = a.copy() # Another way of copying
4  a[0] = 2
5  print(b)
```



itempool.com/mise23/live

Pop Quiz 1:

What is the output of the following code:

```
1  a = [1, 2]
2  a.append(3)
3  b = list(a)
4  b.append(4)
5  print(a[len(a) - 1])
6  print(b[len(a)])
```

Multidimensional lists

2D Lists

Let's assume 3 friends rank their 3 favorite cheeses. How can we store that information?

```
1  cheeseBob = ['cheddar','edam','gouda']
2  cheeseMary = ['edam', 'gouda','cheddar']
3  cheeseLiz = ['gouda','cheddar','edam']
```

2D Lists

With 2D lists, we can store this information in only one variable.

```
cheeses = [ ['cheddar','edam','gouda'], ['edam', 'gouda','cheddar'], ['gouda','cheddar','edam'] ]
```

Recall that lists can contain elements of any data type, since list is a data type itself, we can form lists of lists!

Appending

Now a fourth friend arrives and he also ranks his favorite cheeses.

```
cheeseDan = ['mozzarella', 'cheddar', 'ricotta']
```

How can we add this information to our 2D list containing the favorite cheeses from Bob, Mary and Liz?

Appending

We still use the **append()** function!!

```
cheeses.append(cheeseDan)
print(cheeses)
```

```
[['cheddar', 'edam', 'gouda'], ['edam', 'gouda', 'cheddar'], ['gouda', 'cheddar', 'edam'], ['mozzarella',
    'cheddar', 'ricotta']]
```

Visualizing 2D lists as a table

We can think of 2D lists as a table!

cheddar	edam	gouda
edam	gouda	cheddar
gouda	cheddar	edam
mozzarella	cheddar	ricotta

Pop Quiz 2:

Which of the following result in the variable *l* being the list [[0, 0], [0, 3]]

A)
```
l = [[0, 0]]
l.append([0, 3])
```

B)
```
l = [[0, 0]] * 2
l[1][1] = 3
```

C)
```
l = [[0] * 2, [0] * 2]
l[1][1] = 3
```

D)
```
a = [[0, 0], [0, 0]]
l = a.copy()
a[1][1] = 3
```

E)
```
l = [0, 0] + [0, 3]
```

2D Lists - Dimensions

How can we get the number of 1D lists stored inside our 2D list (i.e. the number of **rows** of our 2D list)?

```
print(len(cheeses)) # output: 4
```

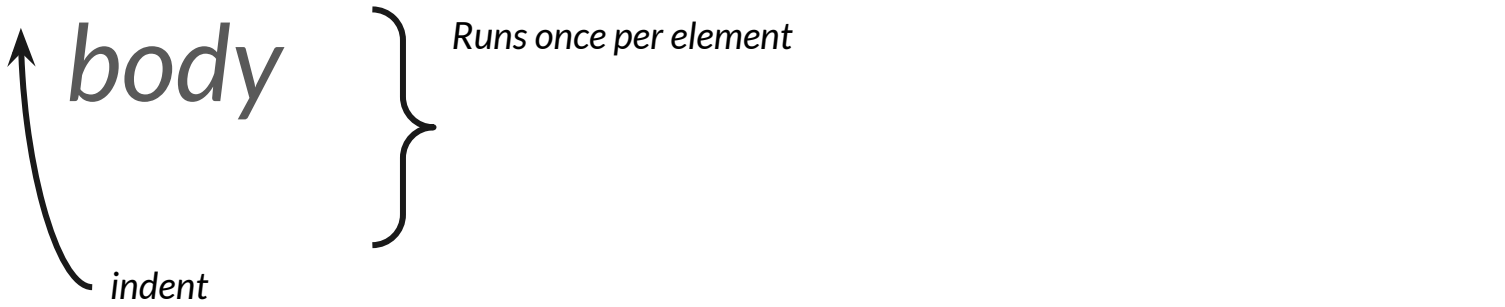
And what about the elements a given 1D list contains (i.e. the number of **columns** of our 2D list)?

```
print(len(cheeses[0])) # output: 3
```

This gives us the two dimensions of our 2D list: the number of **rows** and the number of **columns**!

Recall: anatomy of a for loop

for i **in** *list*:



Challenge: Loops on 2D Lists

How can we print our cheese 2D list with only one 1D list per line?

```
for rank in cheeses:
    print(rank)
```

```
['cheddar', 'edam', 'gouda']
['edam', 'gouda', 'cheddar']
['gouda', 'cheddar', 'edam']
['mozzarella', 'cheddar', 'ricotta', 'gouda']
```

Challenge: Nested Loops on 2D Lists

How can we print all different cheeses on our cheese 2D list?

We need to check **every value** in our 2D list in order to get a complete list of all the cheeses present in our 2D list.

DEMO TIME :D

Challenge: Nested Loops on 2D Lists

```
diffCheeses = []

for rank in cheeses:
    for cheese in rank:
        if cheese not in diffCheeses:
            diffCheeses.append(cheese)

print(diffCheeses)
```

```
['cheddar', 'edam', 'gouda', 'mozzarella', 'ricotta']
```

Demo result

Pop Quiz 3:

What is the output of the following code:

```
1  l = [[1, 2, 3], [1], [1, 2], [1, 2, 3, 4]]
2  res = 0
3  for i in range(len(l)):
4      j = 0
5      while j < len(l[i]):
6          res = res + l[i][j]
7          j += 2
8  print(res)
```

ND Lists

Lists are not limited to 1 or 2 dimensions. They can have many more dimensions: 3, 4, 20, 100, …

All methods applied to 1D or 2D lists can also be adapted to lists of higher dimensions.

An example of a 3D list would be if we know had 2 different groups of friends ranking their favorite cheeses.

```
cheeses =   [   [   ['cheddar','edam','gouda'],
                    ['edam', 'gouda','cheddar'],
                    ['gouda','cheddar','edam'],
                    ['mozzarella', 'cheddar', 'ricotta','gouda'] ],

                [   ['mozzarella, ricotta, gouda'],
                    ['ricotta, mozzarella, cheddar'] ] ]
```

```
print(len(cheeses)) # output: 2
print(len(cheeses[0])) # output: 4
print(len(cheeses[0][0])) # output: 3
```

Advanced features of lists

Feature 1: List Comprehension

new_list = [f(i) **for** i **in** *list*]

Some expression that uses i

- Lightweight and simple way of constructing lists
- List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists

Some examples:

```
numbers = [1, 2, 3, 4, 5]
colors = ["red", "blue", "green"]
```

```
doubles = [2 * i for i in numbers]
```

```
balloons = [color + " balloon" for color in colors]
```

```
concats = [str(numbers[i]) + colors[i] for i in range(min(len(colors), len(numbers)))]
```

new syntax!

```
evens = [i for i in range(10) if i % 2 == 0]
```

Recall that ranges are like lists!

Feature 2: List Slicing

Given a list, let's say l = [1,2,3,4,5]

List slicing follows the format: <list_to_slice>[start_index: end_index -1: step]

The step is mostly ignored, meaning the defaut step is one, but it can be set to any value, when the start_index is not set, the default value is the first index of the array, when the end_index is not set, the default value is the last index of the array

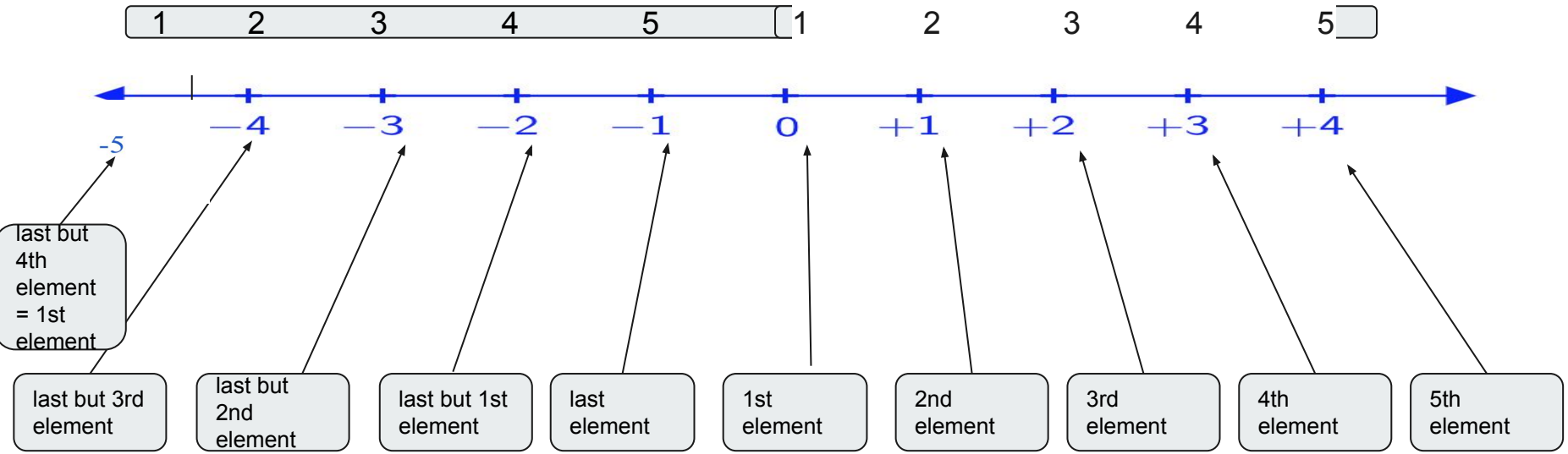
So arr[: :] will return the whole array from first index to last index, so when one does arr[: :-1], it will return the whole array from the last index to the first index, meaning arr[: :-1] will return [5,4,3,2,1] because negative slicing starts from the opposite index of the positive slice

Eg. arr[1] -> second element , arr[-1] -> last element of array

arr[2] -> 3rd element, arr[-2] -> last but one element

arr[3] -> 4th element, arr[-3] -> last but 2nd element

List Slicing Visualization



Further reading to understand list slicing better:

https://www.codingem.com/reverse-slicing-in-python/#:~:text=Indexing%20in%20Python,-To%20access%20an&text=In%20Python%2C%20indexing%20is%20zero,the%201st%20and%202nd%20elements.

What's next?

Homework will be posted on Piazza by tomorrow!

You won't learn anything if you don't try the homeworks

Class 6: Recursion

How to use functions that call themselves

How to solve combinatorial problems in Python