COMPSCI 101

Principles of Programming

Lecture 17 - Slicing lists, some list methods, is operator vs. "=="

1

Learning outcomes

At the end of this lecture, students should be able to use:

- lists and the + and * operators
- list slices
- list methods

and,

- understand the difference between '==' and 'is'

2

Recap

Slide 14 from Lecture 16.

Do the following two sections of code give the same output? If not, what is the difference in output?

Code A

```
list1 = [1, 2, 3]
list2 = list1

for index in range(len(list1)):
    list2[index] = list1[index] * 2

print("1.", list1)
print("2.", list2)
```

```
Output Code A
1. [2, 4, 6]
2. [2, 4, 6]
```

Code B

```
list1 = [1, 2, 3]
list2 = [1, 2, 3]

for index in range(len(list1)):
    list2[index] = list1[index] * 2

print("1.", list1)
print("2.", list2)
```

```
Output Code B
1. [1, 2, 3]
2. [2, 4, 6]
```

3

Lists and the + Operator (concatenation)

Applying the + operator to two lists produces **a new list** containing all the elements of the first list followed by all the elements of the second list.

You can only **concatenate two list objects** (not a list object and a string object, not a list object and an integer object, …).

```
list1 = [10, 20, 30, 40, 50]
list2 = [100, 200]
list3 = list1 + list2

print("1.", list3)
print("2.", 100 in list1)
print("3.", 40 not in list2)

list3 = list3 + [-4]
print("4.", list3)
```

```
1. [10, 20, 30, 40, 50, 100, 200]
2. False
3. True
4. [10, 20, 30, 40, 50, 100, 200, -4]
```

4

Lists and the * Operator (repeat)

The * operator produces **a new list** which "repeats" the original list's contents.

You can only repeat a list in combination with an integer, i.e., the_list * an_integer.

```
list1 = [10, 20]
list2 = list1 * 2
list3 = list2 * 3

print("1.", list1)
print("2.", list2)
print("3.", list3)
```

```
1. [10, 20]
2. [10, 20, 10, 20]
3. [10, 20, 10, 20, 10, 20, 10, 20, 10, 20, 10, 20]
```

5

Getting slices of lists

The slice operation behaves the same way as it does with the elements of a string. Within square brackets, you may have one or two colons (:). The number before the first colon is the start index, the number after the first colon is the end index (one greater than the last index in the slice), and the number after the second colon is the step.

The step indicates the gap between elements in the slice taken. The default step is 1.

Slicing returns a **new list** object.

```
list1 = [10, 20, 30, 40, 50]
list2 = list1[0:3:1]
print("1.", list2)

list3 = list1[3:5:1]
print("2.", list3)
```

Does the same job as:

```
list1 = [10, 20, 30, 40, 50]
list2 = list1[0:3]
print("1.", list2)

list3 = list1[3:5]
print("2.", list3)
```

```
1. [10, 20, 30]
2. [40, 50]
```

6

Getting slices of lists

The number before the first colon is the start index, the number after the first colon is the end index (one greater than the last index), and the number after the second colon is the step.

```
list1 = [10, 20, 30, 40, 50, 60]
list2 = list1[0:5:3]
print("1.", list2)

list3 = list1[2:5:2]
print("2.", list3)
```

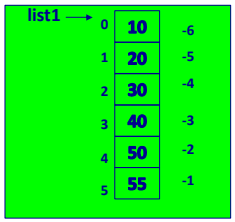
```
1. [10, 40]
2. [30, 50]
```

7

Getting slices of lists

The number after the second colon is the step. The step can be a negative number. (The default step is 1 - see previous slide.)

```
list1 = [10, 20, 30, 40, 50, 55]
list2 = list1[1:6:-3]
print("1.", list2)

list3 = list1[-1:-4:-2]
print("2.", list3)

list4 = list1[-1:-6:-2]
print("3.", list4)
```




```
1. []
2. [55, 40]
3. [55, 40, 20]
```

8

Getting slices of lists

Three numbers in square brackets separated by colons define the **start**, **end** and **step** of the slice, e.g., `list1[1:6:3]`.

The default for the first number is the beginning of the list, e.g.,

```
list1 = [10, 20, 30, 40, 50, 55]
list2 = list1[:4:1] #same as list2 = list1[0:4:1]
print(list2)
```

```
[10, 20, 30, 40]
```

The default for the second number is the end of the list, e.g.,

```
list1 = [10, 20, 30, 40, 50, 55]
list2 = list1[2::2]   #same as list2 = list1[2:len(list1):2]
print(list2)
```

```
[30, 50]
```

The default for the step value is 1.

Some inbuilt functions which work with lists

Below are four in-built functions which can be used with lists:

- **len(**a_list**)** returns the number of elements.
- **min(**a_list**)** returns the minimum element in the list.
- **max(**a_list**)** returns the maximum element in the list.
- **sum(**a_list**)** returns the sum of the elements in the list (only for numbers).

```
list1 = [10, 20, 30, 40, 50, 55]
minimum = min(list1)
total = sum(list1)
print("length: ", len(list1), ", min:", minimum, ", max:",
                              max(list1), ", sum:", total, sep="")
```

```
length: 6, min: 10, max: 55, sum: 205
```


Dot notation

We use **dot notation** to call a method on a specific object. In dot notation, a dot is placed between the object and the method which is to be applied to the object.

Each type of object has many methods which can be called with that type of object. For example a string object has the methods `find()`, `upper()`, `lower()`, `strip()`, `isdigit()`, `isalpha()`, `split()` and many more:

```
words = "Over the rainbow"
position = words.find("r")
words = words.lower()
result = words.isalpha()

print("position:", position,"words:", words, "result:", result)
```

```
position: 3 words: over the rainbow result: False
```


Some list methods

There are many methods which can be used with list objects. Below and on the next slides are five methods which we will use:

index(x**)** returns the index of the first element from the left in the list with a value equal to x. Python throws an error if there is no such value in the list. Because of this, `index(x)` is usually preceded by a check for that element using the **in** operator.

```
list1 = [10, 20, 30, 40, 50, 55]
if 40 in list1:                    #check first
  position = list1.index(40)
  print("40 is in position", position, "in the list")
else:
  print("40 is not in the list")
```

```
40 is in position 3 in the list
```

A list method

pop(index**)** removes and returns the item at the position given by the index number. The 'popped' element is returned by the method. An error results if there is no such index in the list.

pop() with no index removes and returns the last item.

```
list1 = [10, 20, 30, 40, 50, 55]
if len(list1) > 2:
   popped = list1.pop(2)

print("Popped", popped, "from the list", list1)
print(list1.pop())
print(list1)
```

```
Popped 30  from the list [10, 20, 40, 50, 55]
55
[10, 20, 40, 50]
```

13

Another list method

insert(i, x) inserts an element at a given index. The first argument is the index at which to insert the element, e.g., `my_list.insert(1, 62)` inserts 62 into position 1 of the list, moving the rest of the elements along one (the element at index 1 moves to index 2, the element at index 2 moves to index 3, and so on).

```
list1 = [10, 20, 30, 40, 50, 55]
list1.insert(3, 77)
print(list1)

list1.insert(6, 99)
print(list1)

list1.insert(0, 44)
print(list1)
```

```
[10, 20, 30, 77, 40, 50, 55]
[10, 20, 30, 77, 40, 50, 99, 55]
[44, 10, 20, 30, 77, 40, 50, 99, 55]
```

14

Another list method

append(x**)** adds the element to the end of the list.

```
list1 = [10, 20, 30, 40, 50, 55]
list1.append(77)
print("1.", list1)

list1.append(99)
print("2.", list1)

list1.append(44)
print("3.", list1)
```

```
1. [10, 20, 30, 40, 50, 55, 77]
2. [10, 20, 30, 40, 50, 55, 77, 99]
3. [10, 20, 30, 40, 50, 55, 77, 99, 44]
```

15

More list methods

sort() sorts the elements of the list, **in place**. Only the order of the list elements is modified (unless already sorted).

```
list1 = [60, 20, 80, 10, 30, 55]
print(list1)
list1.sort()
print(list1)
```

```
[60, 20, 80, 10, 30, 55]
[10, 20, 30, 55, 60, 80]
```

reverse() reverses the elements of the list, **in place**. Only the order of the list elements is modified.

```
list1 = [10, 20, 70, 80, 50, 55]
print(list1)
list1.reverse()
print(list1)
```

```
[10, 20, 70, 80, 50, 55]
[55, 50, 80, 70, 20, 10]
```

16

Exercise

Complete the `get_selected_numbers()` function which returns a **sorted** list of all the numbers from the `numbers` list which are at the **indices** given in `the indices_to_include` list. **Note:** the function should only use valid **non-negative** indices from the list.

```
def get_selected_numbers(numbers, indices_to_include):




def main():
    numbers = [5, 12, 4, 11, 9, 8]
    indices_to_include = [3, 0, 1, 16, 23, -2]
    numbers_found = get_selected_numbers(numbers,
                                         indices_to_include)
    print("Numbers from list:", numbers_found)

main()
```

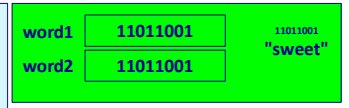
```
Numbers from list: [5, 11, 12]
```


The is operator

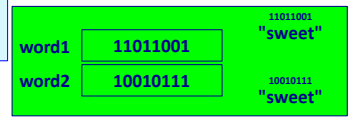
The == operator is used to test if two objects contain the same information.

The is operator is used to test if two variables reference (point to) the same object.

```
word1 = "sweet"
word2 = word1
print("1.", word1 == word2)
print("2.", word1 is word2)

word2 = word2.upper()
word2 = word2.lower()

print("3.", word1 == word2)
print("4.", word1 is word2)
```



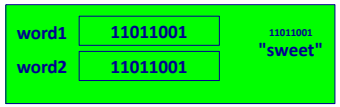

```
1. True
2. True
3. True
4. False
```

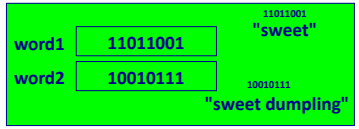


Strings are Immutable

Strings are "**immutable**", i.e., the characters in a string object cannot be changed. Whenever a string is changed in some way, a new string object (with a new memory address) is created.

```
word1 = "sweet"
word2 = word1
print("1.", word1, word2)
print("2.", word1 is word2)

word2 = word2 + " dumpling"
print("3.", word1, word2)
print("4.", word1 is word2)
```



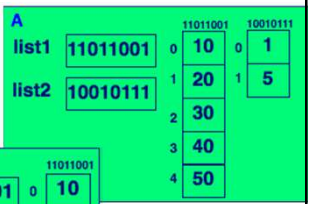


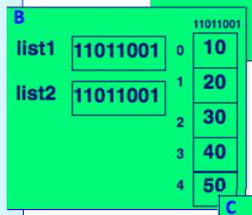

```
1. sweet sweet
2. True
3. sweet sweet dumpling
4. False
```


Lists are Mutable

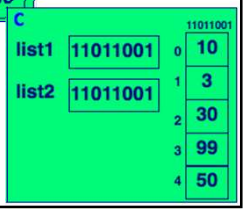
Lists are "**mutable**", i.e., the elements in a list object can be updated and adjusted.

```
list1 = [10, 20, 30, 40, 50]
list2 = [1, 5]  #A
print("1.", list1)
print("2.", list2)
print("3.", list1 is list2)

list2 = list1  #B
print("4.", list1 is list2)

list1[3] = 99
list2[1] = 3   #C

print("5.", list1)
print("6.", list2)
print("7.", list1 is list2)
```






```
1. [10, 20, 30, 40, 50]
2. [1, 5]
3. False
4. True
5. [10, 3, 30, 99, 50]
6. [10, 3, 30, 99, 50]
7. True
```



Exercise

Complete the `remove_multiples()` function which removes all the elements in the parameter `list, number_list`, which are multiples of the parameter, `multiples_of`.

```
def remove_multiples(number_list, multiples_of):




def main():
   numbers = [25, 5, 9, 10, 15, 8]
   print(numbers)
   remove_multiples(numbers, 5)  #remove multiples of 5
   print("Numbers left", numbers)

main()
```

```
[25, 5, 9, 10, 15, 8]
Numbers left [9, 8]
```

21

Summary

A list stores data as a sequence

- We use a for … in … to iterate through the elements of a list
- len() returns the number of elements in a list
- min() returns the minimum of the elements in a list
- max() returns the maximum of the elements in a list
- sum() returns the sum of the elements in a list
- Each element of the list can be accessed using the index operator. The index can be negative (starting from the end of the list)
- Slices of lists can be obtained by using [slice_start: slice_end: step]
- index(element) returns the index of the element in a list
- insert(index, element) inserts an element into a list into the required index
- append(element) adds the element to the end of the list
- reverse() reverses the elements of a list in place
- sort() sorts the elements of a list in place
- Lists are mutable

22

Examples of Python features used in this lecture

```
list1 = [4, 6, 2, 5, 8]
result = 8 in list1
for element in list1:
        …
min_value = len(list1)
min_value = min(list1)
max_value = max(list1)  #if the list elements are numbers
total = sum(list1)  #if the list elements are numbers
element_from_end = list1[-2]
list2 = list1[1:5:2]
position = list1.index(3)
element = list1.pop(1)
list1.insert(4, 66)
list1.append(54)
list1.reverse()
list1.sort()
```

23