**Lecture 17 - Slicing lists, some list methods**

**Learning outcomes**

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

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

**Recap**

Lecture 16 – lists are **mutable**.

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

**Code A**

```python
list1 = [1, 2, 3]
list2 = list1
for index in range(len(list1)):
    list2[index] = list1[index] * 2
print("1.", list1)
print("2.", list2)
```

**Code B**

```python
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]

**Output Code B**

1. [1, 2, 3]
2. [2, 4, 6]

**Lists Operators – 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)
print("4.", list3)
```python

1. [10, 20, 30, 40, 50, 100, 200]
2. False
3. True
4. [10, 20, 30, 40, 50, 100, 200, -4, 9]
Lists Operators – 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]

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.

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]
print("1.", list2)
list3 = list1[3:5]
print("2.", list3)
```

Does the same job as:

```
list1 = [10, 20, 30, 40, 50]
list2 = list1[0:3:1]
print("1.", list2)
list3 = list1[3:5:1]
print("2.", list3)
```

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

```
list4 = [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.
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.)

1. \([10, 20, 30, 40, 50, 55]\)
2. \([]\)
3. \([55, 40]\)
4. \([55, 40, 20]\)

```python
list1 = [10, 20, 30, 40, 50, 55]
list2 = list1[1:6:3]
print("1.", list1)
print("2.", list2)
list3 = list1[-1:-4:-2]
print("3.", list3)
list4 = list1[-1:-6:-2]
print("4.", list4)
```

list2 = list1[1:6:3]

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).

```python
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="")
```

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()` and many more:

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

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

- `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.

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

```python
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())
```

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

Another list method

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

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

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).

```python
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]

More list methods

`sort()` sorts the elements of the list, in place. If sorting a list of lists, only the first element in each list is considered in the comparison operations. Only the order of the list elements is modified (unless already sorted).

```python
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.

```python
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]
Exercise
Complete the \texttt{insert\_names()} function which goes through each name in the \texttt{names\_to\_insert} list, asks the user the name before which they wish to insert, and inserts the name. \textbf{Note:} a name from the \texttt{names\_to\_insert} list is not inserted if it is already in the \texttt{names\_list}. Only the first step of the output is shown below.

\begin{verbatim}
def insert_names(names_list, names_to_insert):
    for name in names_to_insert:
        if name not in names_list:
            print()
            print(names_list)
            prompt = " Insert " + name + " before (enter name): ")
            before_name = input(prompt)

def main():
    names_to_insert = ["Eleni", "Ed", "Anand"]
    insert_names(names, names_to_insert)
    print(names)

["Jane", "Lee", "Sian", "Ben", "Sio", "Shanti", "NoName"]
Insert Eleni before (enter name): Sian
...
\end{verbatim}

Exercise
Complete the \texttt{remove\_multiples()} function which removes all the elements in the parameter list, \texttt{number\_list}, which are multiples of the parameter, \texttt{multiples\_of}.

\begin{verbatim}
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]
\end{verbatim}

Summary
A list stores data as a sequence

- We use a \texttt{for ... in ...} to iterate through the elements of a list
- \texttt{len()} returns the number of elements in a list
- \texttt{min()} returns the minimum of the elements in a list
- \texttt{max()} returns the maximum of the elements in a list
- \texttt{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 \texttt{[slice\_start: slice\_end: step]}
- \texttt{index(element)} returns the index of the element in a list
- \texttt{insert(index, element)} inserts an element into a list into the required index
- \texttt{append(element)} adds the element to the end of the list
- \texttt{reverse()} reverses the elements of a list in place
- \texttt{sort()} sorts the elements of a list in place
- Lists are mutable

Examples of Python features used in this lecture
\begin{verbatim}
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()
\end{verbatim}