LIST

Presented By
M.Malarmathi
AP/IT

List loop

```
>>> mylist = [[1,2,3],[4,5,6,7],[8,9,10]]
>>> for x in mylist:
        if len(x)==3:
         print x
```

O/P

```
[1, 2, 3] [8, 9, 10]
```

List Mutability

- lists are mutable.
- bracket operator appears on the left side of an assignment, it identifies the element of the list that
- >>> numbers = [17, 123]
- >>> numbers[1] = 5
- >>> print numbers
- [17, 5]will be assigned

List indices work the same way as string indices:

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an IndexError.
- If an index has a negative value, it counts backward from the end of the list.

The in operator also works on lists

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
Falsekward from the end of the list.
```

Aliasing

- If a refers to an object and you assign b = a, then both variables refer to the same object:

```
>>> a = [1, 2, 3]
>>> b = a
>>> b is a
True
```

- The association of a variable with an object is called a **reference. In this example, there are** two references to the same object.
- An object with more than one reference has more than one name, so we say that the object is **aliased.**
- If the aliased object is mutable, changes made with one alias affect the other:

```
>>> b[0] = 17
>>> print a
[17, 2, 3]
```

- This behavior can be useful, it is error-prone
- It is safer to avoid aliasing when working with mutable objects
- For immutable objects like strings, aliasing is not as much of a problem.

```
a = 'banana'
b = 'banana'
```

- It almost never makes a difference whether a and b refer to the same string or not

Cloning lists

```
original_list = [10, 22, 44, 23, 4]
new_list = list(original_list)
print(original_list)
print(new_list)
```

O/P:

```
[10, 22, 44, 23, 4]
 [10, 22, 44, 23, 4]
```

```
a = [81, 82, 83]
b = a[:]
# make a clone using slice
print(a == b)
b[0] = 5
print(a)
print(b)
```

O/P:

```
True
[81, 82, 83]
[5, 82, 83]
```

List Parameters

- When you pass a list to a function, the function gets a reference to the list. If the function modifies a list parameter, the caller sees the change. For example, delete_head removes the first element from a list:

```
def delete_head(t):
del t[0]
```

- Here's how it is used:

```
>>> letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> print letters
['b', 'c']
```

List Parameters

- It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list:

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> print t1
[1, 2, 3]
>>> print t2
None
>>> t3 = t1 + [4]
>>> print t3
[1, 2, 3, 4]
```

- This difference is important when you write functions that are supposed to modify lists.

Thank You