Chapter 10

Dr. Raza Ul Mustafa Khokhar
American University
Computer Science Department - CSC-148

Lists

- A list is a sequential collection of Python data values, where each value is identified by an index. The values that make up a list are called its elements

```
[10, 20, 30, 40]
["spam", "bungee", "swallow"]
```

```
["hello", 2.0, 5, [10, 20]]
```

Lists examples

```
vocabulary = ["iteration", "selection", "control"]
numbers = [17, 123]
empty = []
mixedlist = ["hello", 2.0, 5*2, [10, 20]]

print(numbers)
print(mixedlist)
newlist = [ numbers, vocabulary ]
print(newlist)
```

List length

- As with strings, the function len returns the length of a list (the number of items in the list).
- However, since lists can have items which are themselves lists, it important to note that len only returns the top-most length. In other words, sublists are considered to be a single item when counting the length of the list.

```
alist =  ["hello", 2.0, 5, [10, 20]]
print(len(alist))
print(len(['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'],
[1, 2, 3]]))
```

Accessing elements

```
numbers = [17, 123, 87, 34, 66, 8398, 44]
print(numbers[2])
print(numbers[9 - 8])
print(numbers[-2])
print(numbers[len(numbers) - 1])
```

List membership

in and not in are boolean operators that test membership in a sequence. We used them previously with strings and they also work here.

```
fruit = ["apple", "orange", "banana", "cherry"]

print("apple" in fruit)
print("pear" in fruit)
```

Concatenation and Repetition

```
fruit = ["apple", "orange", "banana", "cherry"]
print([1, 2] + [3, 4])
print(fruit + [6, 7, 8, 9])

print([0] * 4)
print([1, 2, ["hello", "goodbye"]] * 2)
```

List slices

```
a_list = ['a', 'b', 'c', 'd', 'e', 'f']
print(a_list[1:3])
print(a_list[:4])
print(a_list[3:])
print(a_list[:])
```

Lists are mutable

```
fruit = ["banana", "apple", "cherry"]
print(fruit)

fruit[0] = "pear"
fruit[-1] = "orange"
print(fruit)
```

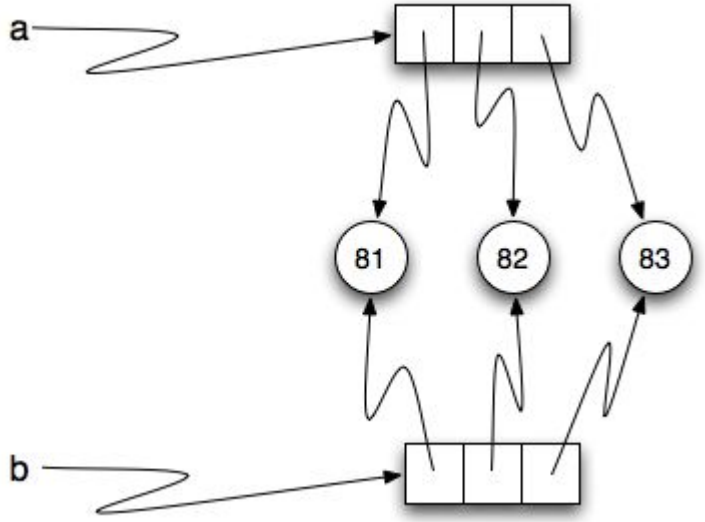
List deletion

The del statement removes an element from a list by using its position.

```
a = ['one', 'two', 'three']
del a[1]
print(a)

alist = ['a', 'b', 'c', 'd', 'e', 'f']
del alist[1:5]
print(alist)
```

Object and references

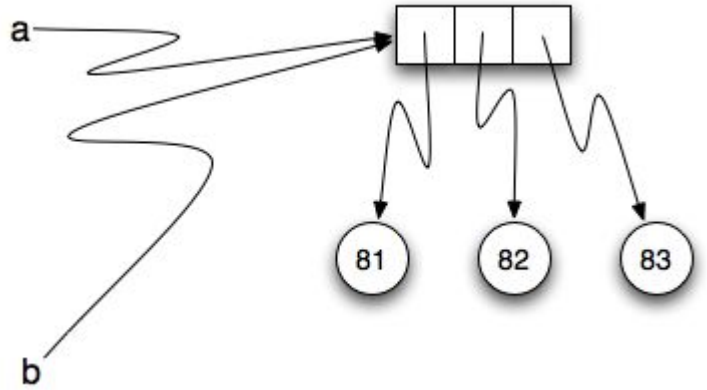
```
a = [81, 82, 83]
b = [81, 82, 83]

print(a is b)

print(a == b)
```



Aliasing

Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

```
a = [81, 82, 83]
b = a
print(a is b)
```



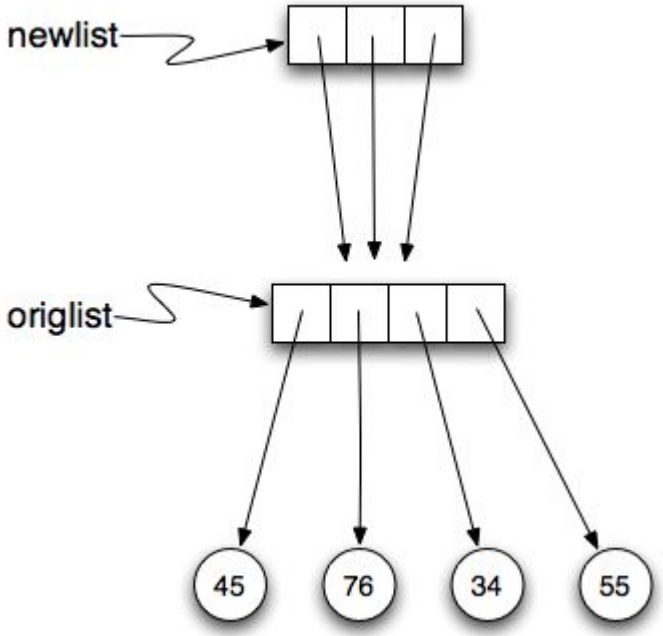
Repetitions and references

With a list, the repetition operator creates copies of the references.

```
origlist = [45, 76, 34, 55]
print(origlist * 3)
```

Repetitions and references cont...

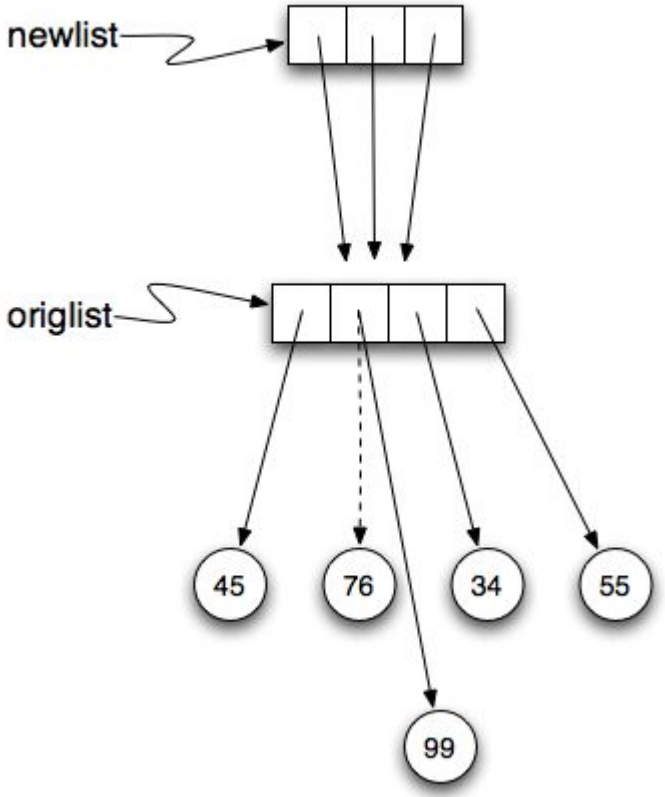
```
origlist = [45, 76, 34, 55]
print(origlist * 3)

newlist = [origlist] * 3

print(newlist)
```



newlist is a list of three references to origlist that were created by the repetition operator.

Repetitions and references cont...

```
origlist = [45, 76, 34, 55]

newlist = [origlist] * 3

print(newlist)

origlist[1] = 99

print(newlist)
```



List methods

```
mylist = []
mylist.append(5)
mylist.append(27)
mylist.append(3)
mylist.append(12)
print(mylist)
mylist.insert(1, 12)        # Insert 12 at pos 1, shift other items up
print(mylist)
print(mylist.count(12))     # How many times is 12 in mylist?
print(mylist.index(3))      # Find index of first 3 in mylist
print(mylist.count(5))
mylist.reverse()
print(mylist)
mylist.sort()
print(mylist)
mylist.remove(5)            # Removes the first 12 in the list
print(mylist)
lastitem = mylist.pop()     # Removes and returns the last item of the list
print(lastitem)
print(mylist)
```

Important to note !!!

It is important to remember that methods like append, sort, and reverse all return None. This means that re-assigning mylist to the result of sorting mylist will result in losing the entire list.

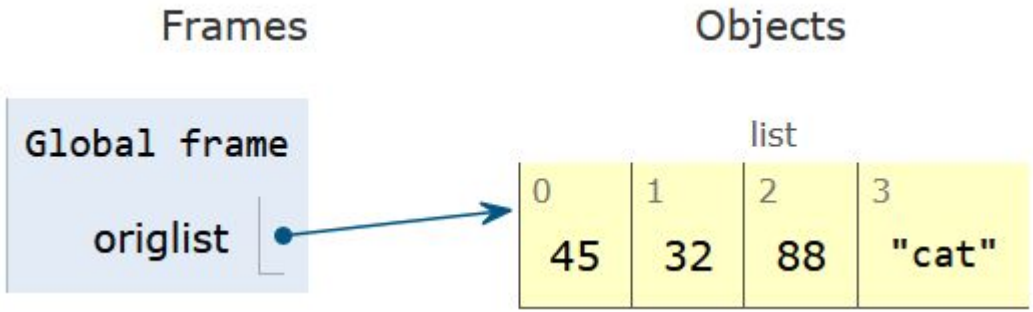
```
mylist = []
mylist.append(5)
mylist.append(27)
mylist.append(3)
mylist.append(12)
print(mylist)

mylist = mylist.sort()   #probably an error
print(mylist)
```

Append versus Concatenate

It is also important to realize that with append, the original list is simply modified. On the other hand, with concatenation, an entirely new list is created.

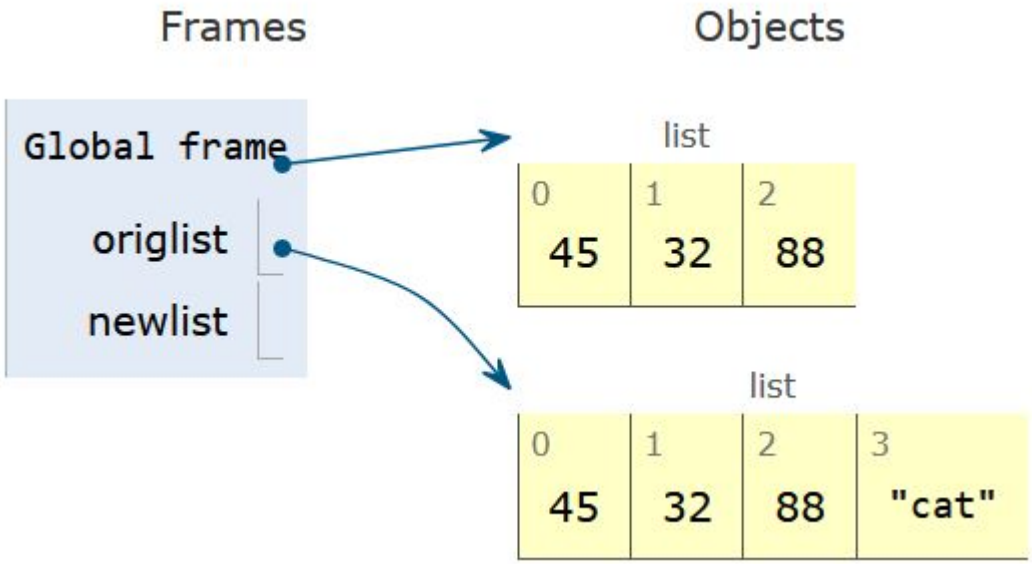
In order to use concatenation, we need to write an assignment statement that uses the accumulator pattern:

```
origlist = origlist + ["cat"]
```



A new list in concatenation

newlist refers to a list which is a copy of the original list, origlist, with the new item “cat” added to the end. origlist still contains the three values it did before the concatenation. This is why the assignment operation is necessary as part of the accumulator pattern.



Lists and for loops

```
fruits = ["apple", "orange", "banana", "cherry"]

for afruit in fruits:     # by item
    print(afruit)
```

Range

```
fruits = ["apple", "orange", "banana", "cherry"]

for position in range(len(fruits)):     # by index
    print(fruits[position])
```

```python
numbers = [1, 2, 3, 4, 5]
print(numbers)

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

print(numbers)
```

```
[1, 2, 3, 4, 5]
[1, 4, 9, 16, 25]
```

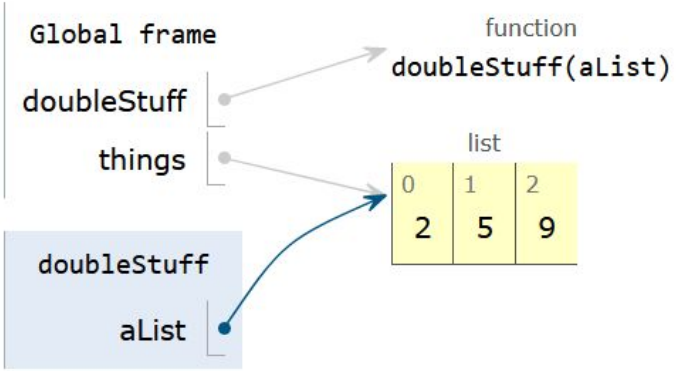
Names with length greater than 3?

```
long_names = 0
for name in ["Joe", "Sally", "Amy", "Brad"]:
    if len(name) > 3:
        long_names += 1
print(long_names)
```

https://shorturl.at/deirS

Very important ! Using list as a parameters

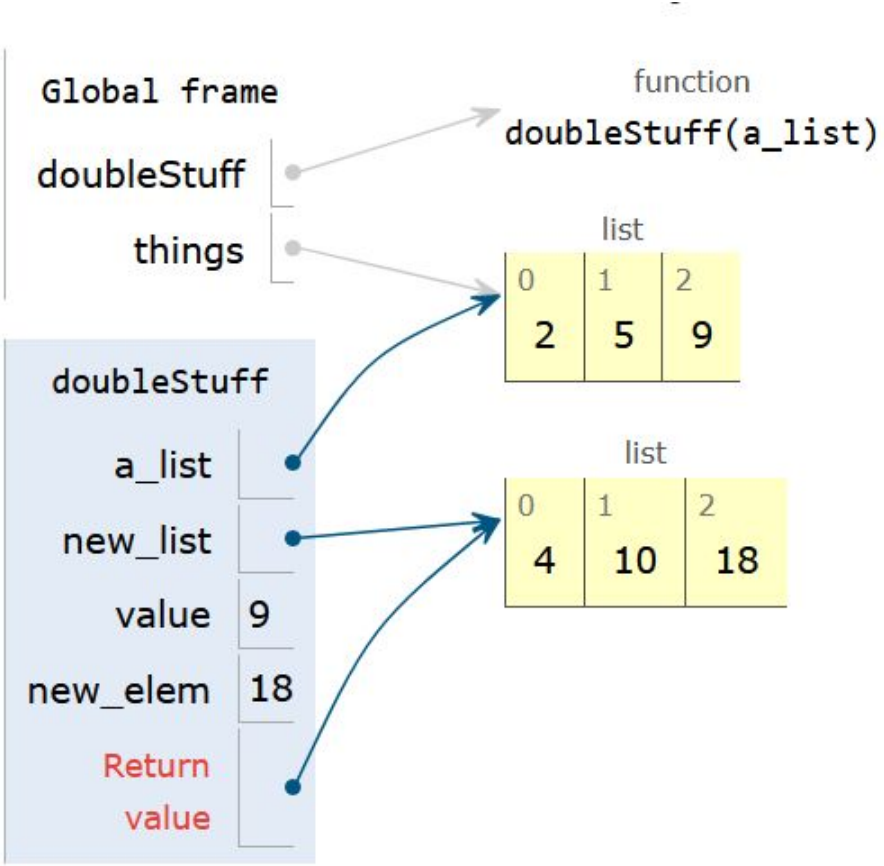
Passing a list as an argument actually passes a reference to the list, not a copy of the list. Since lists are mutable, changes made to the elements referenced by the parameter change the same list that the argument is referencing.




```
def doubleStuff(aList):
    """ Overwrite each element in aList with double
its value. """
    for position in range(len(aList)):
        aList[position] = 2 * aList[position]

things = [2, 5, 9]
print(things)
doubleStuff(things)
print(things)
```

Pure function

```
def doubleStuff(a_list):
    """ Return a new list in which contains doubles of
the elements in a_list. """
    new_list = []
    for value in a_list:
        new_elem = 2 * value
        new_list.append(new_elem)
    return new_list

things = [2, 5, 9]
print(things)
things = doubleStuff(things)
print(things)
```



List Comprehensions

```
[<expression> for <item> in <sequence> if  <condition>]
```

```
mylist = [1,2,3,4,5]

yourlist = [item ** 2 for item in mylist]

print(yourlist)
```

Nested lists

```
nested = ["hello", 2.0, 5, [10, 20]]
innerlist = nested[3]
print(innerlist)
item = innerlist[1]
print(item)

print(nested[3][1])
```

String & Lists

```
song = "The rain in Spain..."
wds = song.split()
print(wds)
```


```
song = "The rain in Spain..."
wds = song.split()
print(wds)
```

Inverse of split is join

The inverse of the split method is join

```
wds = ["red", "blue", "green"]
glue = ';'
s = glue.join(wds)
print(s)
print(wds)

print("***".join(wds))
print("".join(wds))
```

list Type Conversion Function

Python has a built-in type conversion function called list that tries to turn whatever you give it into a list. For example, try the following:

```
xs = list("Crunchy Frog")
print(xs)
```

Tuples and Mutability

- In other words, strings are **immutable** and lists are **mutable**.
- A **tuple**, like a list, is a sequence of items of any type. Unlike lists, however, tuples are immutable. Syntactically, a tuple is a comma-separated sequence of values. Although it is not necessary, it is conventional to enclose tuples in parentheses

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress", "Atlanta,
Georgia")
```

```
julia[0] = 'X'
TypeError: 'tuple' object does not support item assignment
```

Slice – New Tuple

```
julia = ("Julia", "Roberts", 1967, "Duplicity", 2009, "Actress",
"Atlanta, Georgia")
print(julia[2])
print(julia[2:6])

print(len(julia))

for field in julia:
    print(field)

julia = julia[:3] + ("Eat Pray Love", 2010) + julia[5:]
print(julia)
```

Important to note !

To create a tuple with a single element (but you're probably not likely to do that too often), we have to include the final comma, because without the final comma, Python treats the (5) below as an integer in parentheses:

```
tup = (5,)
print(type(tup))

x = (5)
print(type(x))
```

Tuple Assignment

Python has a very powerful **tuple assignment** feature that allows a tuple of variables on the left of an assignment to be assigned values from a tuple on the right of the assignment.

```
(name, surname, birth_year, movie, movie_year, profession, birth_place) = julia
```

Naturally, the number of variables on the left and the number of values on the right have to be the same.

```
(a, b, c, d) = (1, 2, 3)
ValueError: need more than 3 values to unpack
```

Tuples as a return value

- Functions can return tuples as a return value

```
def circleInfo(r):
    """ Return (circumference, area) of a circle of radius r """
    c = 2 * 3.14159 * r
    a = 3.14159 * r * r
    return (c, a)

print(circleInfo(10))
```

Slides & Material

razaulmustafa.us/cs148/