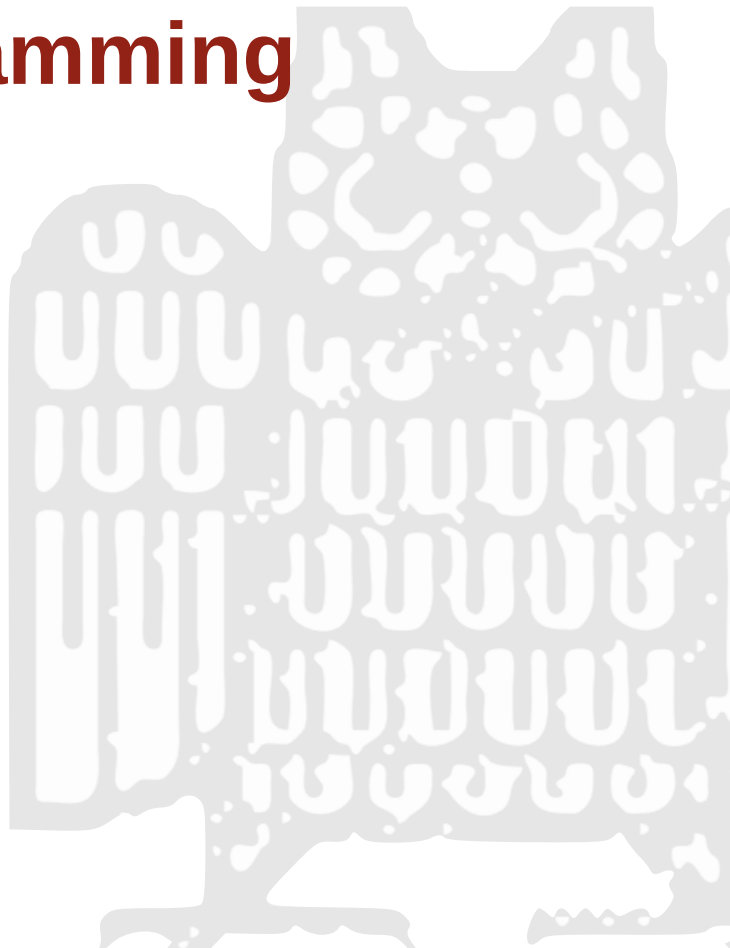
Introduction to Python Programming

07 – Collections: Lists

Josef van Genabith (Stefan Thater)
Dept. of Language Science & Technology
Universität des Saarlandes

WS 2022/23



Recap: Functions

```
def isEven(x):
   if x%2==0:
      return True
   else:
      return False

a = isEven(5)
b = isEven(8)
```

- Functions have a **name**.
  What is the function's name?
- Functions can have **parameters**.
  What is the function's parameter?
- Functions can be **called**.
  In which lines is the function called above, and what happens with the values in the parentheses?
- Functions can **return values**.
  What does the function return? How can we access the returned values after the function has been executed?

Exercise sheets 2/3 – Sample Solutions

```
# Factorial

number = int(input("Please enter a number: "))

result = 1

for i in range(1, number + 1):
    result *= i
    # result = result * i

print("The factorial of", number, "is", result)
```

Lists

```
           -5   -4   -3   -2     -1
myList = ["a", "b", "c", "d", "hello"]
           0    1    2    3      4

print(myList[1])      # b
print(myList[4])      # hello
print(myList[-2])     # ... what?
print(myList[4][1])   # ... what?
print(myList[4][4])   # ... what?
print(myList[5])      # ... what?
```

Lists

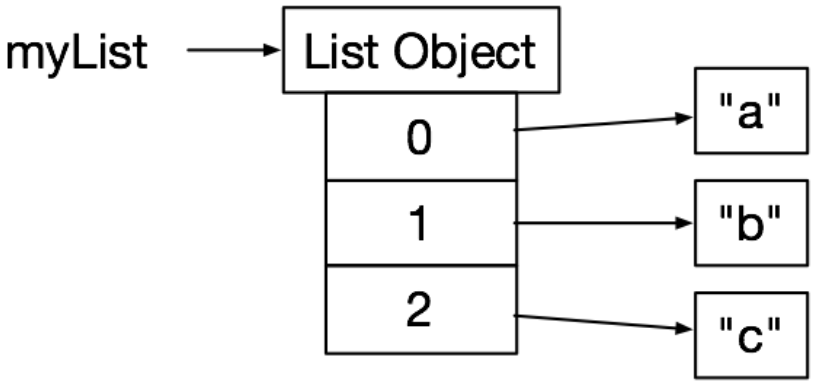
```
           -5   -4   -3   -2     -1
myList = ["a", "b", "c", "d", "hello"]
           0    1    2    3      4

print(myList[1])      # b
print(myList[4])      # hello
print(myList[-2])     # d
print(myList[4][1])   # e
print(myList[4][4])   # o
print(myList[5])      # … error message …
```

Lists are **mutable**

- Mutable objects:
  - lists, sets, dicts
  - can be modified
- Immutable objects:
  - ints, floats, …, tuples, strings
  - cannot be modified

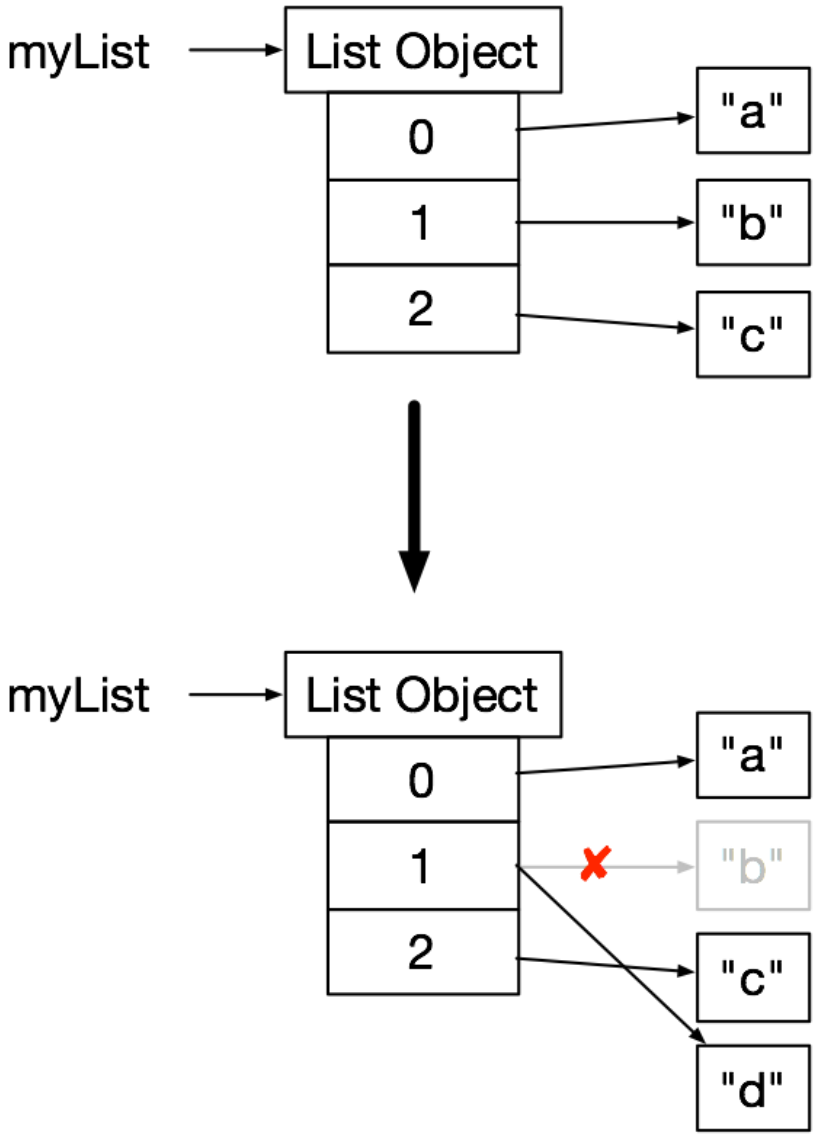
Replacing List Items

```
myList = ["a","b","c"]
```



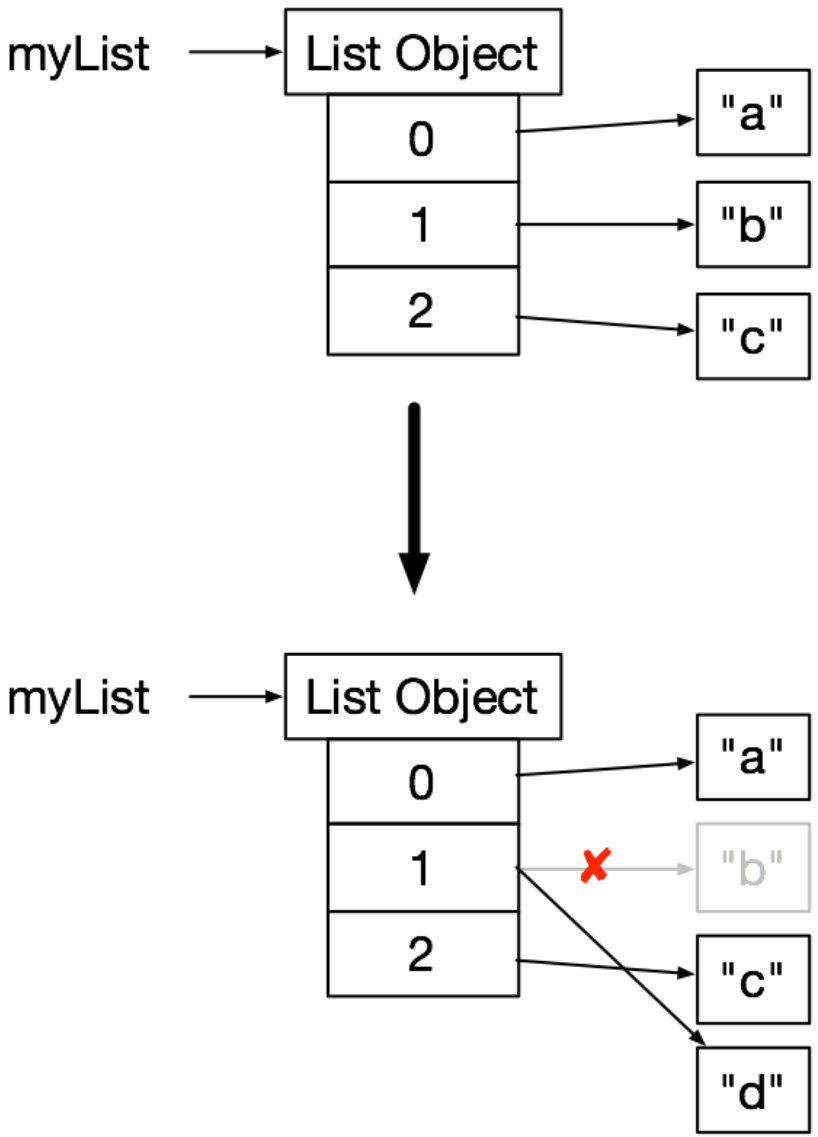
Replacing List Items

```
myList = ["a","b","c"]
...
myList[1] = 'd'
```



Replacing List Items

```
myList = ["a","b","c"]
...
myList[1] = 'd`
...
print(myList)

# ["a","d","c"]
```



Immutable vs Mutable Types

```
x = "ab"
y = x
x += "c"    # x = x + "c"
print(x)
# "abc"
print(y)
# "ab"
```

```
x = ['a', 'b']
y = x
x.append('c')
print(x)
# ['a', 'b', 'c']
print(y)
# ['a', 'b', 'c']
```

- Left box: We compute a new value (`"abc"`) and assign the new value to variable x
- Right box: We modify the list; changes are “visible” for all variables that refer to the list.

Append vs “+”

```
x = ['a', 'b']
y = x
x = x + ['c']
print(x)
# ['a', 'b', 'c']
print(y)
# ['a', 'b']
```

```
x = ['a', 'b']
y = x
x.append('c')
print(x)
# ['a', 'b', 'c']
print(y)
# ['a', 'b', 'c']
```

- Left box: We compute a new list and (re-) assign it to the variable x
- Right box: We modify the list; changes are “visible” for all variables that refer to the list.

Appending Items to a List

```
            0    1    2
myList = ["a", "b", "c"]

# does not work:
myList[3] = "d"

# this works:
myList.append("d")
print(myList)
# ['a', 'b', 'c', 'd']

print(myList[3])
# d
```

Removing Items from a List

```
           0    1    2    3
myList = ["a", "b", "c", "d"]

del myList[1] # removes "b"
print(myList)
# ["a", "c", "d"]

elt = myList.pop(1) # removes "c" and returns it
print(myList)
# ["a", "d"]
print(elt)
# c
```

Methods

- Methods = functions which are applied ‘on an object’
- `someObject.methodName(parameters)`
- often change the object on which they are called’

```
someList = [1, 2, 3]
someList.append(5)
print(someList)
# [1, 2, 3, 5]
```

More List Methods and Functions

```
1  myList=[3,2,6,1,8]
2  myList.reverse()
3  x = len(myList)
4  myList.sort()
5  myList.insert(2, 5)
6  myList.sort(reverse=True)
7  myList.append(3)
8  x = myList.count(3)
9  myList.remove(3)
10 x = myList.pop()
11 y = myList.pop()
12 z = 4 in myList
13 myList = myList + [7, 8, 9]
14 del myList[:]
```

What are the effects of the following list functions? For each line of the program, write down the current value of myList, x, y and z.

Homework: write down a short description of what the methods do for your reference.

Slicing

`myList[i:j]` – creates a new list containing all items of `myList` at positions `i, i+1, ..., j-1`

Slicing

```
>>> myList = ['a', 'b', 'c', 'd']
>>> myList
['a', 'b', 'c', 'd']
>>> myList[1:3]
['b', 'c']
>>> myList[0:1]
['a']
>>> myList[1:]
['b', 'c', 'd']
>>> myList[:-2]
['a', 'b']
>>> myList[:]
['a', 'b', 'c', 'd']
```

`myList[i:j]` – creates a new list containing all items of `myList` at positions `i, i+1, ..., j-1`

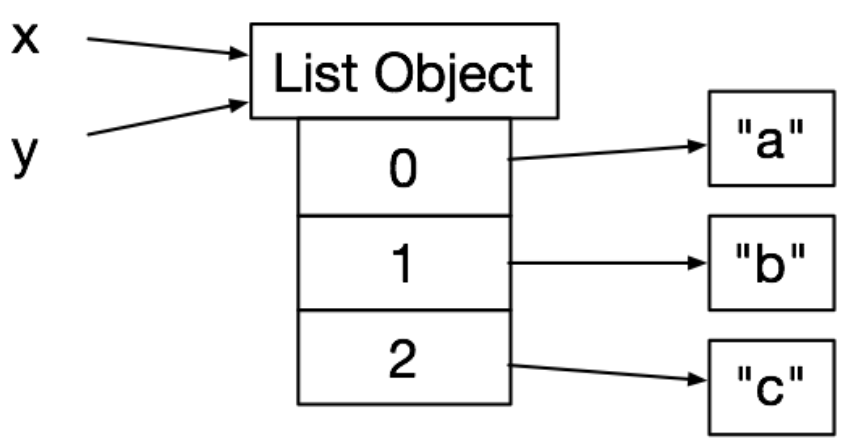
Lists vs Strings

- Strings are also sequences (of characters)
- We can access (read) the items of a string in the same way as we access the items of a list
  - ch = some_string[0] **# first character**
- But strings are immutable sequences: They can't change.
  - some_string[0] = "T" **# DOES NOT WORK!**
- Concatenation creates new strings.
  - some_string = "T" + some_string [1:]

Shared References

- Variables do not contain values (like a box does)
- Rather, variables **point** to positions in the memory where the value is stored – like a name points to a person.
- The difference (containing vs pointing) is particularly important for mutable values (like lists).

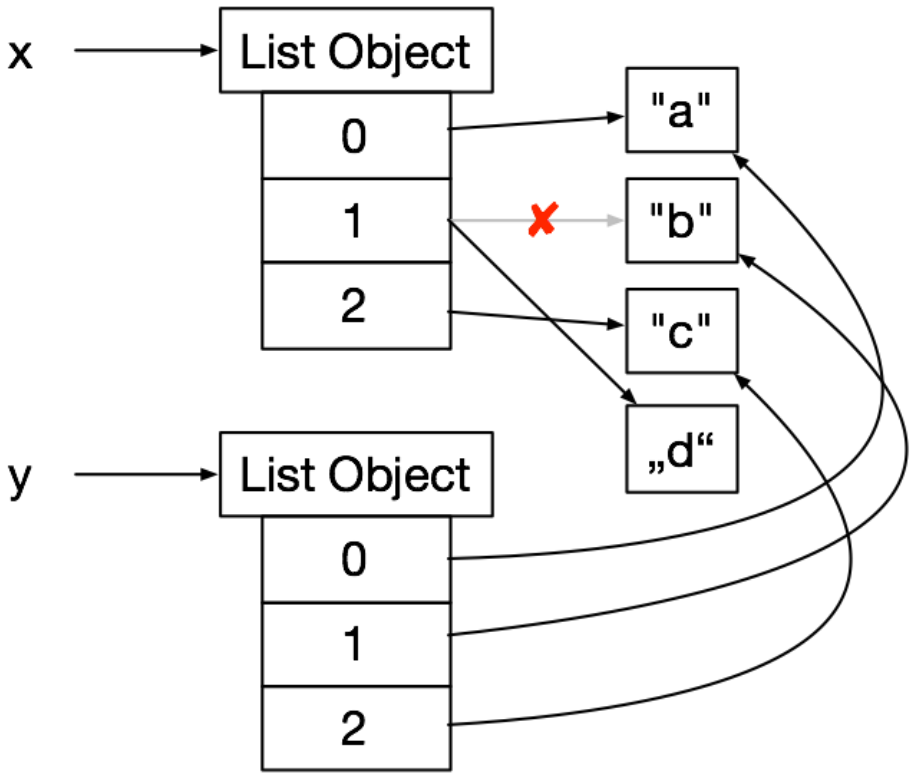
```
>>> x = ['a', 'b', 'c']
>>> y = x
>>> x[1] = 'd'
>>> x
['a', 'd', 'c']
>>> y
['a', 'd', 'c']
```



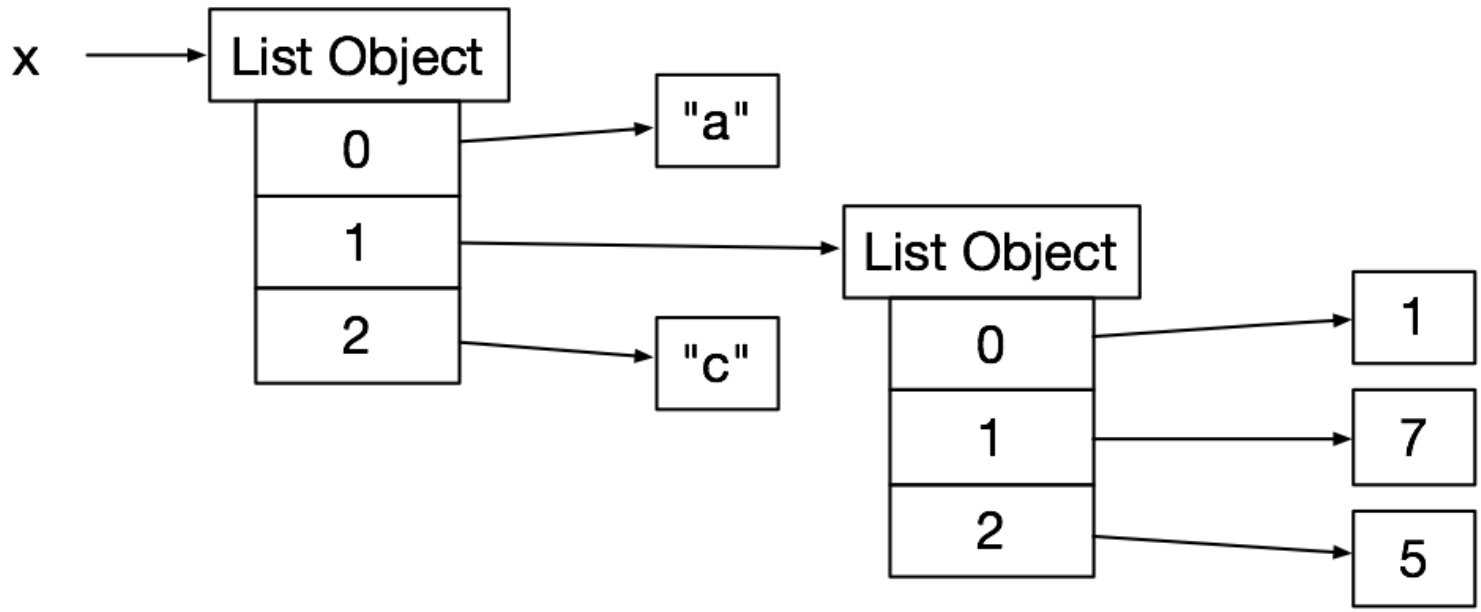
Shared References

- Slicing can be used to create a shallow copy of a list.
- A shallow copy constructs a new list object containing references to the items of the original list.

```
>>> x = ['a', 'b', 'c']
>>> y = x[:]
>>> x[1] = 'd'
>>> x
['a', 'd', 'c']
>>> y
['a', 'b', 'c']
```



Nested Lists




```
x = ['a', [1,7,5], 'c']

print("x[0] is: ", x[0])          # a
print("x[1] is: ", x[1])          # [1,7,5]
print("x[1][0] is: ", x[1][0])    # 1
print("x[1][1] is: ", x[1][1])    # 7
```

Exercise

- What are the values of x and y? Draw a diagram!

```
>>> x = ['a', ['b'], 'c']
>>> y = x[:]
>>> x[1].append('d')
>>> y[0] = 'e'
>>> x
... what?
>>> y
... what?
```

Exercise

- What is the value of x? Draw a diagram!

```
>>> x = ['a', 'b']
>>> y = [x, x]
>>> y[1].append('c')
>>> x
... what?
```

`is` vs `==`

```
>>> x = ['a', 'b', 'c']
>>> y = ['a', 'b', 'c']
>>> z = x
>>> x == y
True
>>> x == z
True
>>> x is y
False
>>> x is z
True
```

- The `==` operator tells whether two values have the same structure (“look the same”)
- The `is` operator tells whether two values are identical.
- **Achtung:** Use `is` only when you really have to; `==` is the appropriate operator in most cases.

Achtung: `is`

```
>>> 220 is 200 + 20
True
>>> 330 is 300 + 30
True
```

- Achtung: `is` checks for identity. Not all values that are equal (==) are represented by the same object.

Assignments and Lists

```
>>> x, y = 1, 2
>>> x, y = [1, 2]
>>> x, y = [1, 2, 3]
ValueError: too many values to unpack (expected 2)
>>> x, *y = [1, 2, 3]
>>> x
1
>>> y
[2, 3]
```

- with “*y” one can specify a “catch-all” variable.