ANNOUNCEMENT

- We will have a one week break as follows:
- **NO lecture** next week (Oct 25th)
- **NO demonstrations** at the following week (Oct 30th to Nov 3rd)

ANNOUNCEMENT (2)

- Hence, the timetable for the future weeks looks like this:

Monday	Tuesday	Wednesday	Thursday	Friday
23rd DEMO 1	24rd DEMO 2	25th *No lecture*	26th DEMO 3	27th DEMO 4
30th *No demos*	31st *No demos*	Nov 1st Lecture	2nd *No demos*	3rd *No demos*
6th *Demos & lectures continue normally from this week on*				

7: Lists

Storing data

- Storing data is very important in most programs

- Although individual variables are fine in lot of cases (and vital in almost any program), they are not sufficient for all purposes

Data structures

- Consider a case, where program needs to store and manipulate thousands (or millions) of data points

- Using individual variables for each case would not be feasible

- Hence, we need something called **data structures**

Why data structures

- Data structures allow **saving several objects into a single structure** and **sequential manipulation** of objects

- In practice, this means for example manipulating a list of objects easily.

- Objects are usually **indexed**

Data structures in Python

- Python supports several data structures. In this course, we are going to discuss
  - Lists (and matrices as special case of lists)
  - Tuples
  - Dictionaries
  - Sets
- In addition, Python supports for example
  - Queues
  - Stacks
  - Trees
  - etc.

What is a list?

- **List** is a dynamic sequence of zero or more consecutive items.
- An item can be of any Python type (integer, floating point, string, another list etc.)
- Strings in Python are considered as sequences of characters; however, strings are *immutable*, while ordinary lists are not.

What is a list? (2)

- Lists are created as objects into memory.

- The objects are accessed via **reference**, that is stored into a variable. The mechanism is similar to referencing strings.

- List items are **ordered** and **indexed starting from zero**.

Defining a list

- Syntax for defining a list with pre-set values:

```
listVariable = [value1, value2, ... ,value_n]
```

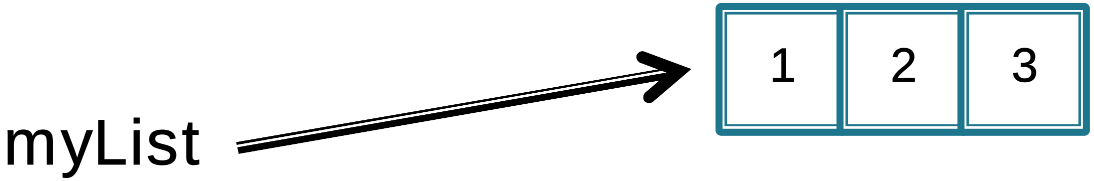
- For example:

```
myList = [1,2,3] # Create a list with values 1,2,3
print myList # Will ouput [1, 2, 3]
n = [] # Define an empty list
```

Defining a list (cont.)

- Defining a list creates a list object into memory, and stores a reference to list into variable.

```
myList = [1,2,3]
```



Defining a list (cont.)

- Note, that list definition accepts all expressions, including variables and function calls:

```
myList = [1 + 1, 2*2, 6 / 2]
print myList # Would output [2, 4, 3]

a = 2
list2 = [a, a + 1, a + 2]
print list2 # Would output [2, 3, 4]
```

Defining a list (cont.)

```
def getBigger(a, b):
   if a > b:
      return a
   else:
      return b

bg = [getBigger(1,4), getBigger(10,2)]
print bg # output [4, 10]
```

Defining a list (cont.)

- The list can alternatively be defined by using the * operator. Operator initializes a list with number of given values:

```
myList = [0] * 5
print myList # Output [0, 0, 0, 0, 0]

m = [5] * 3
print m # Output [5, 5, 5]
```

Defining a list (cont.)

- The **range** function generates a list:

```
myList = range(1,10)
print myList # Output [1, 2, 3, 4, 5, 6, 7, 8, 9]

otherList = range(15,5,-3)
print otherList # Output [15,12,9,6]
```

Range and xrange

- As a reminder: if you want to iterate through an existing set of values, but do not need the list, **`xrange`** is more efficient.

- However, if you need the actual list, you need to use **`range`**.

Length of a list

- The length of a list can be returned by using the **`len()`** function:

```
a = [1,2,3,5,8]
print len(a) # Would output 5
```

List length and last item

- As a general rule:

- Any non-empty list contains items between indices

```
(0 ... len(list) - 1)
```

Extracting an item from a list

- A single item can be retrieved by using the [] operator (just like when extracting characters from a string):

```
n = [0,2,4,6,8]
print n[0] # Would output 0
print n[2] # Would output 4
print n[len(n) – 1] # Would output 8
```

Assigning a value into an item

- ...however, with lists it is also possible to assign a value into an item:

```
n = [2,4,6,8]
n[0] = 1
print n # Would output [1, 4, 6, 8]
```

- Note, that this is **not** possible with strings!

Adding items to a list

- Items can be added into end of a list by using the **append()** method:

```
myList = [1,2,3]
myList.append(4)
print myList # Would output [1, 2, 3, 4]
```

Adding items to a list (cont.)

- To insert an item into specific point at a list, we can use the **insert(***index, object***)** method. The method insert's given object into given index at a list:

```
myList = [1,2,3,4]

myList.insert(0, 5) # Insert 5 at index 0
print myList # Output [5, 1, 2, 3, 4]

myList.insert(2, 10) # Insert 5 at index 0
print myList # Output [5, 1, 10, 2, 3, 4]
```

Adding items to a list (cont.)

- Moreover, multiple lists can be joined by using the + operator:

```
a = [1,2,3]
b = [6,7,8]
a = a + b
print a # Would output [1, 2, 3, 6, 7, 8]
```

- ...and again, the lists can be multiplied with the * operator:

```
a = [1,2,3]
b = a * 3
print b # would output [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Deleting items from a list

- To remove **the first item with given value** from a list, we use the remove() method:

```
a = [1,2,3,4,3,2,1,4]
a.remove(2)
print a # Would output [1, 3, 4, 3, 2, 1, 4]
```

- If no item with such value is found, an error is thrown!

Item in a list?

- To check whether an item can be found in a list, we can use the **in** operator:

```
a = [1,2,3,4,5]
print 3 in a # True
print 0 in a # False
print a[1] + a[2] in a # 2 + 3 = 5 à True
```

- Hence, a routine for checking if a value exist before deleting it could look something like:

```
def deleteItem(myList, item):
    if item in myList:
       myList.remove(item)
```

The in operator

- Note, that the in operator also works with strings:

```
def containsVowels(string):
    vow = "aeiou"
    for vowel in vow:
        if vowel in string:
            return True
    return False
```

Deleting item from a list (cont.)

- To remove (and return) an item at the *given index*, we can use the `pop(index)` method:

```
a = [1,2,3,4,5]
d = a.pop(1)
print d # Would output 2
print a # Would ouput [1, 3, 4, 5]
```

Slicing a list

- A slice (or a "sublist") of a list can be returned by using the syntax familiar from substrings:

```
myList = [1,2,3,4,5,6]
slice = myList[1:3]
print slice # Would output [2, 3]
```

Slicing a list (cont.)

- Again, the start or the end index can be omitted:

```
a = [1,2,3,4,5]
print a[2:] # output starting from index 2
print a[:3] # output until index 3 is reached
```

Slicing a list (cont.)

- If we omit both indexes, Python returns the whole list. This is very useful, as it can be used to create a copy of a list:

```
a = [1,2,3,4]
b = a[:]
print b # Would output [1, 2, 3, 4]
```

List variables are references

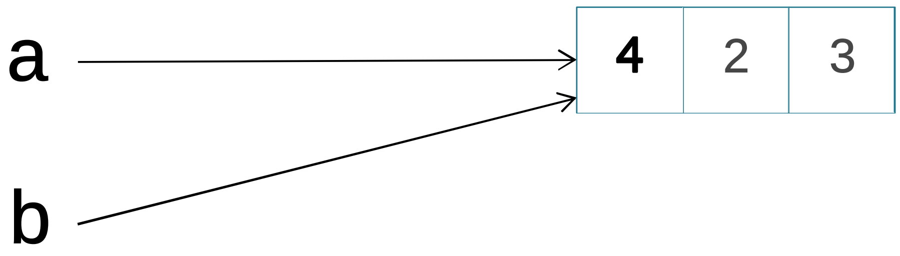
- But why do we need to copy a list instead of just using b = a ?

- Consider this:

```
a = [1,2,3,4]
b = a
b[0] = 5
print a # What would this output...?
```

List variables are references

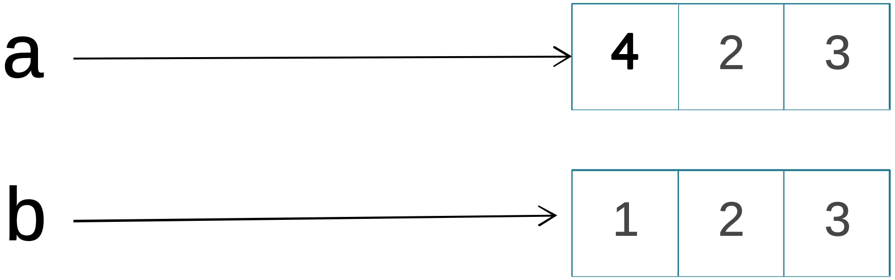
- Using the = operator with lists copies a reference to given list:



```
a = [1,2,3]
b = a
a[0] = 4
```

List variables are references

- However, using the [:] operator creates a copy of the original list:



```
a = [1,2,3]
b = a[:]
a[0] = 4
```

List variables are references

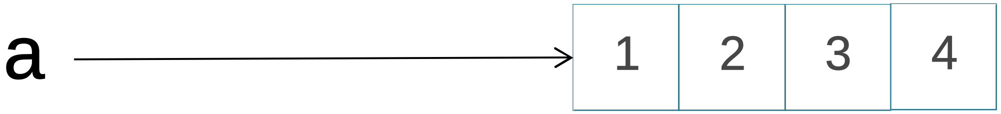
- Note, that because list variables are always references to actual lists, their usage in functions is also different:

```
def changeFirst(myList):
  myList[0] = 0

a = [1,2,3,4]
changeFirst(a) # passes the reference to a as an argument!
print a # Would output [0, 2, 3, 4] !
```

Lists as arguments

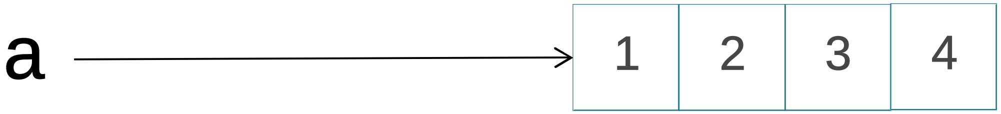
```
def changeFirst(myList):
  myList[0] = 0

a = [1,2,3,4]
changeFirst(a)
print a
```

Lists as arguments



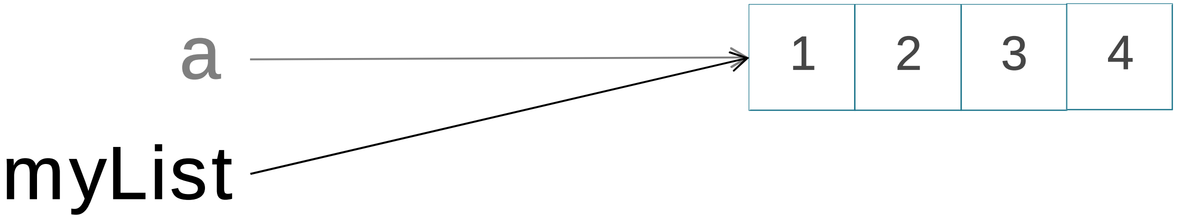
```
def changeFirst(myList):
  myList[0] = 0

a = [1,2,3,4]
changeFirst(a)
print a
```

Lists as arguments



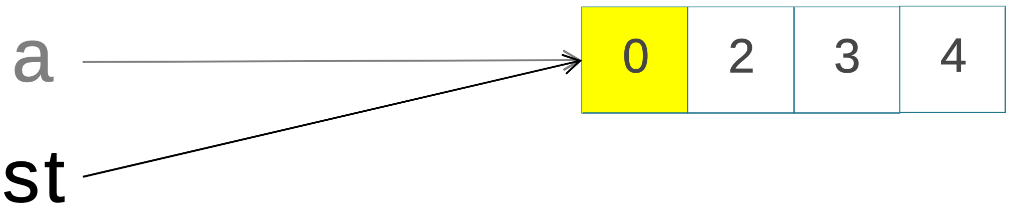
```
def changeFirst(myList):
  myList[0] = 0

a = [1,2,3,4]
changeFirst(a)
print a
```

Lists as arguments



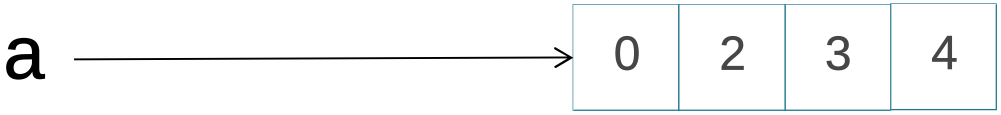
```
➡ def changeFirst(myList):
    myList[0] = 0

  a = [1,2,3,4]
  changeFirst(a)
  print a
```

Lists as arguments



```
def changeFirst(myList):
    myList[0] = 0

a = [1,2,3,4]
changeFirst(a)
print a
```

Lists as arguments



```
def changeFirst(myList):
  myList[0] = 0

a = [1,2,3,4]
changeFirst(a)
print a
```

References and equality

- It is different to compare whether two lists are *equal* or the *same list*.

- The equality can be (again) compared with == operator

- To check whether two variables reference the same list, we use Python's **is** operator:

```
a = [1,2,3,4]
b = a[:]
print a == b # True, because lists hold equal values
print a is b # False, not the same list
print a is not b # ..and hence True
```

Other useful methods for list

- Methods count() and index() work similarly to strings:

```
m = [2,4,6,8,10,12,10,8]
print m.count(8) # Output 2
print m.index(10) # Output 4
```

Other useful methods for list (cont.)

- To sort a list into increasing order **in place**, we can use the sort() method:

```
a = [5,1,2,4,3]
a.sort()
print a # Would output [1, 2, 3, 4, 5]
```

- To reverse a string in place, we can use the **reverse()** method:

```
a.reverse()
print a # Would output [5, 4, 3, 2, 1]
```

Other useful methods for list (cont.)

- To return a new list with items sorted, we can use the **sorted** function

```
a = [1,3,2,4]
b = sorted(a)
print a # Outputs [1, 3, 2, 4]
print b # Outputs [1, 2, 3, 4]
```

sort() or sorted()?

- In short:

`sort()` sorts the list **in place**

`sorted()` returns **a new list** with items sorted

Iterating through a list

- Again, Python's for statement is very useful for iterating through values in the list:

```
myList = [2,4,6,9,12]
for i in myList:
    print i
```

...would iterate through a list and output the values one by one.

Iterating through a list (2)

- To iterate a list in reverse order, we can use the **reversed** function

```
myList = [2,4,6,9,12]
for i in reversed(myList):
    print i
```

Using reversed function

- Note, that using reversed() does not reverse the items in the list.

- Instead, it creates a **reverse iterator** that can be used to travel the items in reverse order.

Using for statement in list expression

- Pythons for statement can be used in an expression to create **a new list** based on an existing one.

- The syntax for *list comprehension* is

```
[<expression with x> for x in list]
```

List comprehension

- Consider this example:

```
m = [1, 3, 5]
m2 = [x * 2 for x in m]
print m2 # outputs [2, 6, 10]
```

List comprehension

```
m2 = [x * 2 for x in m]
```

- ...means, that items are picked from list **m** one at a time, multiplied by two, and inserted into a new list in the same order.
- Finally, a reference to the new list is assigned to variable m2.

List comprehension

- These two are hence equivalent:

```
myList = [1,2,3,4]
mySecondList = [x + 1 for x in myList]
```

OR

```
myList = [1,2,3,4]
mySecondList = []
for item in myList:
  mySecondList.append(item + 1)
```

Case example 1: Swap first and last values in a list

```
def swapFirstAndLast(myList):
  tmp = myList[0]
  myList[0] = myList[len(myList) – 1]
  myList[len(myList) - 1] = tmp

a = [2,4,6,8,10,12]
swapFirstAndLast(a)
print a # output [12, 4, 6, 8, 10, 2]
```

Case example 1 (cont.)

- ...or utilize Python's variable value swapping syntax:

```
def swapFirstAndLast(lst):
  lst[0],lst[-1] = lst[-1],lst[0]
```

Case example 2: Remove even numbers from a list (or does it?)

```
def removeEvenNumbers(myList):
   for value in myList:
      # Check and remove if even number
      if value % 2 == 0:
         myList.remove(value)

theList = range(1,11) # range 1...10
removeEvenNumbers(theList)
print theList # Outputs [1, 3, 5, 7, 9]
```

QUESTION

- Then again, what would this output? Why?

```
def removeEvenNumbers(myList):
   for value in myList:
      # Check and remove if even number
      if value % 2 == 0:
         myList.remove(value)

lst = [1,2,2,3]
removeEvenNumbers(lst)
print lst
```