8: More Lists & Tuples

Practical stuff

- What's left?
  - Five lectures with ViLLE exercises including this one:
    - Dictionaries
    - File operations
    - External modules
    - Error handling
- Four more demonstrations
- Course project (announced next week)

Items other than integers

- The items stored in lists can be of any Python type, including numbers, strings, other lists etc.

```
myList = ["abc","def","ghi"]
otherList = [1.0, 2.3, 56.123]
booleanList = [True, True, False]
```

Mixing object types

- Note, that different items can be of different types.

```
myList = ["John Smith", 23, 11.234, True]

ageList = ["Dad", 32, "Mom", 29, "Kid", 3]
```

Mixing object types (cont.)

- However, mixing types can lead to misbehavior, if you're not sure about the types of objects in list:

```
for value in myList:
   print value / 3 # what if a string...?
```

- Hence, it is *not recommended* to mix different types of objects in the same list!

Checking object types

- To check the type of any object in Python, we can use the **isinstance** function.

- Function gets the object and the type name as a parameter, and returns True, if object is of that type.

Checking object types (cont.)

- Some type names for the objects discussed so far:

- **int** – integer numbers (e.g. 3, 21, -4)
- **long** – longer integer values (> 2147483647)
- **float** – floating point numbers (e.g. -2.3, 0.1)
- **bool** – boolean values (True, False)
- **str** – strings (e.g. "abc", "Hello all!")
- **list** - lists

Checking object types (cont.)

- Function calls that return True:

```
isinstance(24, int)
isinstance(3.11, float)
isinstance("Hi there", str)
isinstance(range(1,10), list)
isinstance(not False, bool)
```

Checking object types (cont.)

- Function calls that return False:

```
isinstance(31, float) # int is no float
isinstance("24", int) # string is no int
isinstance([2, 3], int) # list is no int
```

Example

- Calculate the sum of all integers and floats in a list given as an argument

```
def numberSum(lst):
    """ Calculates the sum of all numbers in lst"""
    sum = 0
    for item in lst:
        if isinstance(item, int) or isinstance(item, float):
            sum += item
    return sum
```

The type function

- Note, that a type of an object can be returned with the **type** function

```
a = 4
print type(a) # Outputs <type ’int’>
```

The type function (2)

- This can also be utilized to check the type of an object, which is sometimes more convenient.

```
a = "hi all!"
# following is equal to
# if isinstance(a, str):
if type(a) == str:
   print "It’s a string"
```

The type function (3)

- A function that returns True if given value is a number type:

```
def isNumber(object):
    types = [int, long, float]
    return type(object) in types
```

Converting between types

- The type name can also be used as a function to convert between types (if a conversion can be made):

```
a = 4
b = float(a) # 4.0
c = long(a) # 4L
d = str(a) # "4"
```

Lists inside lists

- Items in list can also be lists:

```
myList = [[1,2,3], [4,5,6]]
```

...would initialize a new list with two lists as items.

Lists inside lists (cont.)

- To retrieve a value from list stored as an item:

```
myList = [ [1,2,3], [4,5,6] ]
a = myList[0]
print a[0] # Would output 1
print a[2] # Would output 3
print myList[0][0] # Would output 1
print myList[1][1] # Would output 5
```

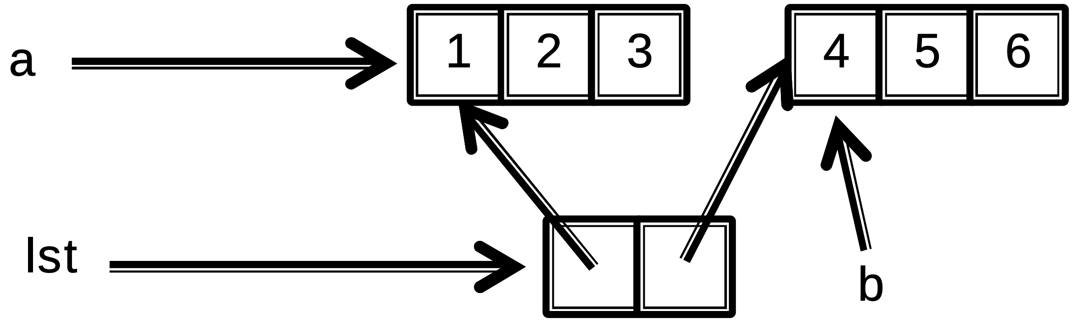
Lists are stored as references

- If a list contains other lists, it actually holds the references to those lists

- Again, a single list can have multiple references pointing to it

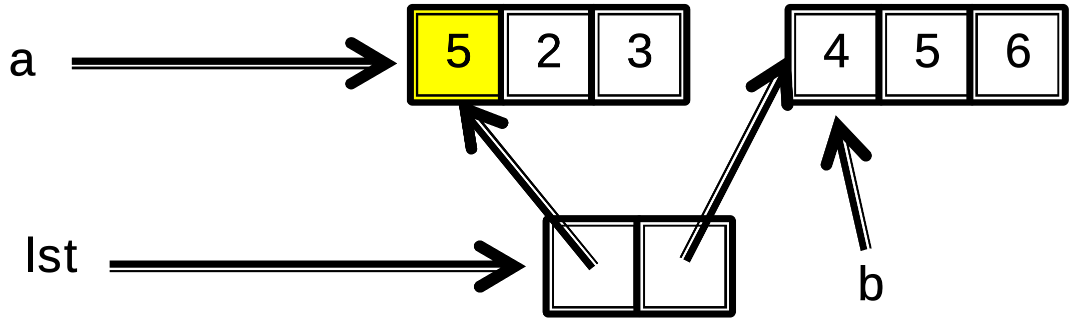
Example

```
a = [1,2,3]
b = [4,5,6]
lst = [a,b]
```



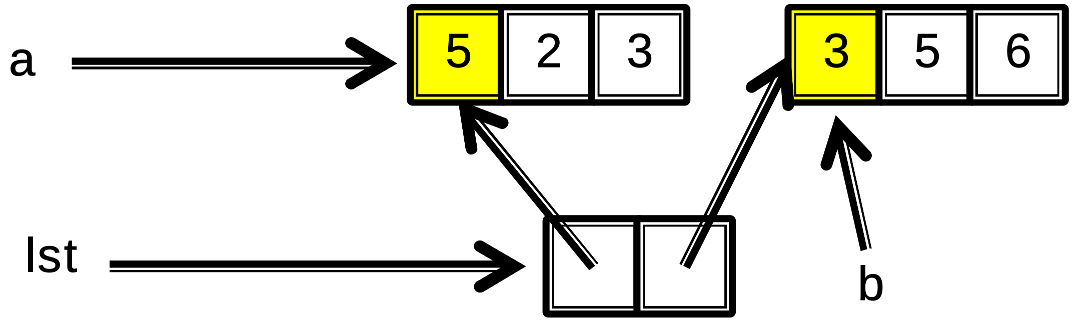
Example

```
lst[0][0] = 5
```



Example

```
lst[1][0] = 3
print a # outputs [5, 2, 3]
print b # outputs [3, 5, 6]
```



Matrices

- Lists stored as list items is a good method to store matrices. Each row is stored as a list.

1	2	3
4	5	6
7	8	9

```
m = [ [1,2,3] , [4,5,6], [7,8,9] ]
```

Matrices (cont.)

- To access the first item in the second row in matrix m, we use

```
print m[1][0] # Output row 1, item 0
m[1][0] = 24 # Set the value of item 0 in row 1
```

Length of inner lists

- Note that the `len()` function returns the number of items in a list, despite the type of the item.

```
m = [ [1,2,3,4], [2,3,4,5], [3,4,5,6] ]
print len(m) # would output 3
print len(m[0]) # would output 4
row2 = m[1]
print len(row2) # would output 4
```

Matrices (cont.)

- The inner lists can of course contain different number (and different types) of items:

a	b		
c	d	e	
f	g	h	i

```
m = [ ["a","b"], ["c","d","e"], ["f","g","h","i"] ]
```

Iterating through matrices

- To iterate through all values in a matrix, we need to use two for loops:

```
m = [ [1,2,3], [4,5], [6,7,8,9] ]
# first loop iterates through lists in list m
for row in m:
   # second loop iterates through items in row
   for value in row:
      print value
```

Iterating through matrices (cont.)

- A procedure that adds a single item into the end of each row in matrix:

```
def addItemToRows(m):
   for row in m:
      # check that item is a list before adding
      if isinstance(row, list):
         row.append(0)
```

Tuples

- A **tuple** is a data structure in Python.
- Tuples are in many ways similar to lists, except that they are **immutable**; hence, once created, you can't the change the stucture or the values of items in tuple.

Initializing a tuple

- The syntax for initalizing a tuple resembles the syntax for initializing a list; note, that parentheses are used instead of brackets.

```
myTuple = (1,2,3)
otherTuple = ("John Smith", 23)
coordinates = (23.11, 3.01)
emptyTuple = ()
multi = (1,2) * 4
joinedTuple = (1,2,3) + (4,5,6)
```

Tuple operations

- Some list operations work for tuples also:

```
mt = (1,2,3,4)
print len(mt) # outputs 4
print mt[0] # outputs 1
print mt.index(3) # outputs 2
print mt.count(2) # outputs 1
other = mt[0:2]
print other # outputs (1, 2)
mt = (1,2) * 2 + (3,4,5)
print mt # outputs (1, 2, 1, 2, 3, 4, 5)
```

Tuple operations (cont.)

- However, since tuples are immutable, some list operations don't work with them.
  - You can't change the value of an item in tuple
  - You can't sort or reverse a tuple
  - You can't append, insert, remove or pop items in tuple
- In short: once a tuple is created, you **can't change it in any way!**

Using tuples

- Tuples are useful when saving value groups:

```
p1 = ("John Smith", 33)
p2 = ("Helen Keller", 51)
p3 = ("Adam Python", 76)

# Store names and ages to list to retrieve later
personsAges = [ p1, p2, p3 ]

# Display all names and ages
for person in personsAges:
   print "Name: " + person[0] + ", age:" + person[1]
```

Using tuples (cont.)

- Program that queries the user for coordinate pairs and saves them into a list:

```
coords = []
x = 1
while x != 0:
   x = input ("Give x (0 to stop) :")
   if x != 0:
      y = input ("Give y :")
      pair = (x,y)
      coords.append(pair)
      # or use coords.append((x,y)) in one line
```

Why use tuples?

- If the tuples are like lists, but immutable, why can't we just use lists?

- Tuples are far more **memory efficient**, since Python interpreter doesn't need to prepare for changes in the object.

Tuples vs. lists

- There is also a **semantic difference**
- Tuples are *heterogeneous:* it is semantically correct to store different types of objects into a tuple (like name and age); hence a tuple is a *combination of values*.
- List are *homogeneous:* if used semantically correctly, each item of a list *should be* of same type.

Tuples and lists

- Hence, it is quite common to store individual data points as tuples, and store the similar tuples into a list

- In short:
  - Tuple is used as a value group, joining the properties of a single entity in a single structure
  - List can be used to store similar items into a single structure

Converting between tuple and list

- If, for some reason, you need to convert a tuple to a list, you can use the list() function.

```
tup = (1,2,4,3,5)
lis = list(tup)
lis.sort() # list is sortable, tuple is not...
```

- To convert from list to a tuple, you can use the tuple() function:

```
tup = tuple(lis)
```

Converting between tuple and list (cont.)

- If you need to convert between a tuple and a list (e.g. to sort data) various times in the program, it's probably better to just use a list.

The zip function

- The zip function in Python can be used to join multiple lists into a single list of tuples.

- The syntax is

```
newList = zip(list_1, list_2, …, list_n)
```

The zip function (cont.)

- The function picks items from lists given as arguments, and creates tuples out of them.
- The first items of each list are joined into the first tuple in the new list, second items form the second tuple and so on.

The zip function (cont.)

- Example:

```
list1 = [1, 2, 3]
list2 = [10, 20, 30]
list3 = zip(list1, list2)
print list3
[(1, 10), (2, 20), (3, 30)]
```

The zip function (cont.)

- Another example

```
x = [11.0, 2.4, -6.3]
y = [0, 22.1, 13.9, 5.0, 4.0]
coord = zip(x,y)
print coord
[(11.0, 0), (2.4, 22.1), (-6.3, 13.9)]
```

The zip function (cont.)

- Yet another example:

```
first = ["John", "Jack", "Jane", "Jill"]
last = ["Smith", "Davis", "Doe", "White"]
ages = [24, 56, 30, 9]

persons = zip(first, last, ages)
```

Case example 1

- A function that gets a list of name-age tuples, and returns the oldest person.

```python
def getOldest(m):
    oldest = ("", 0) # empty person
    for person in m:
        # compare ages
        if person[1] > oldest[1]:
            oldest = person
    return oldest
```

Case example 2

- A procedure that modifies a matrix of numbers by adding the average of items in row to each row.

```
def addAverages(m):
   for row in m:
      sum = 0.0 # init to float to prevent casting
      for value in row:
          sum = sum + value
      # add the average to end of row
      row.append(sum / len(row))
```