Content Delivered in class_6_04-August-2016

- Chapter 4: Collections
  - Lists
    - Creating a Stack with Lists
    - Creataing a Queue with Lists
    - Hard COPY vs Shallow COPY vs Deep COPY
    - Other Built-in functions
  - Tuples
    - Indexing and Slicing
    - Tuples are immutable
    - Buit-in functions on Tuples
    - Tuple Unpacking
    - Tuples in lists
    - lists in tuples
    - Tuple Comprehensions

Interview Questions Discussed

Interview Question 1: Implement the stack mechanism using 'List'

Interview Question 2: Implement a queue mechanism, using collections module

Interview Question 3: what is the difference between = and ==

Assignments Given

Assignment 1: Implement Queue mechanism with lists

Hint: FIFO insert(0, ..) pop()

Assignment 2: Write a Program to convert this heterogeneous tuple completely to flat tuple.

```
Input = (2,23, 34, [55, 'six six', (77, 88, ['nine nine', 0])])
Output = (2, 23, 34, 55, 'six six', 77, 88, 'nine nine', 0)
```

Assignment 3: practice all the exercises performed on list comprehensions, on tuple comprehensions

---

Lists ..

Interview Question 1: Implement the stack mechanism using 'List'

Creating a Stack with Lists

- stack works based on Last-In First-Out (LIFO) mechanism.
- The push and pop operations of stack can be mimicked using append() and pop() methods of list, respectively.

In [1]:
```
stack = [12, 34, 45, 5677]
```

In [2]:
```
print type(stack)
```

```
<type 'list'>
```

In [3]:
```
stack.append(25)
```

In [4]:
```
stack
```

Out[4]: [12, 34, 45, 5677, 25]

In [5]:
```
stack.pop()    # LIFO
```

Out[5]: 25

In [6]:
```
stack
```

Out[6]: [12, 34, 45, 5677]

In [7]:
```
stack.pop()
```

Out[7]: 5677

In [8]:
```
stack.append(0)
```

In [9]:
```
stack
```

Out[9]: [12, 34, 45, 0]

Creataing a Queue with Lists

Assignment 1: Implement Queue mechanism with lists

Hint: FIFO insert(0, ..) pop()

Interview Question 2: Implement a queue mechanism, using collections module

In [11]:
```
from collections import deque
queue = deque(['Python', 'Programming', 'Pearl'])
```

In [12]:
```
print queue     # returns a named tuple
```

```
deque(['Python', 'Programming', 'Pearl'])
```

In [13]:
```
type(queue)      # It is of 'collections.deque' type. Different from basic col
 types
```

Out[13]:
```
collections.deque
```

In [14]:
```
queue.appendleft('George')
```

In [15]:
```
print queue
```

```
deque(['George', 'Python', 'Programming', 'Pearl'])
```

In [16]:
```
queue.appendleft('Bush')
```

In [17]:
```
print queue
```

```
deque(['Bush', 'George', 'Python', 'Programming', 'Pearl'])
```

In [18]:
```
queue.pop()
```

Out[18]:
```
'Pearl'
```

In [20]:
```
print queue
```

```
deque(['Bush', 'George', 'Python', 'Programming'])
```

In [21]:
```
queue.pop()
```

Out[21]:
```
'Programming'
```

In [22]:
```
queue.pop()
```

Out[22]:
```
'Python'
```

In [23]:
```
queue.pop()
```

Out[23]:
```
'George'
```

In [24]:
```
queue.pop()
```

Out[24]:
```
'Bush'
```

In [25]:
```
print queue
```

```
deque([])
```

In [26]:
```
queue.pop()     # Because, queue.pop() can't be applied to an empty queue
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-26-af3c799162a2> in <module>()
----> 1 queue.pop()

IndexError: pop from an empty deque
```

In [28]:
```
queue.remove()    # queue.remove() requires to specify the element to delete
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-28-6c81ca995379> in <module>()
----> 1 queue.remove()

TypeError: remove() takes exactly one argument (0 given)
```

In [175]:
```
queue
```

Out[175]:
```
deque([])
```

In [177]:
```
queue.appendleft('NewWord')
```

In [178]:
```
queue.appendleft(['Canada', 'Australia'])
```

In [179]:
```
queue
```

Out[179]:
```
deque([['Canada', 'Australia'], 'NewWord'])
```

In [180]:
```
queue.clear()    # clears the queue; but retains the queue object
```

In [181]:
```
queue           # empty queue Object
```

Out[181]:
```
deque([])
```

In [182]:
```
del queue       # deletes the object 'queue' from the heap memory
```

Interview Question 3: what is the difference between = and ==

= assignment operation ; Also called as Hard COPY

== equivalence checking operation

In [29]:
```
a = 23  # assigning 23 to 'a'
```

In [30]:
```
a == 23  # checking whether value in 'a' is 23 or not; results in boolean
```

Out[30]: True

Hard COPY vs Shallow COPY vs Deep COPY

In [31]:
```
parList = [1,2,3,4,54,5,56,6]
```

In [32]:
```
childList = parList  # Assignment Operation (or)  Hard COPY
```

Assignment operation wont create a new object; rather the new identifier (variable) refers to the same Object

In [35]:
```
print parList, type(parList)
```

```
[1, 2, 3, 4, 54, 5, 56, 6] <type 'list'>
```

In [36]:
```
print childList, type(childList)
```

```
[1, 2, 3, 4, 54, 5, 56, 6] <type 'list'>
```

In [37]:
```
parList == childList
```

Out[37]: True

In [38]:
```
parList is childList   # becoz both are refering to the same object.
```

Out[38]: True

In [39]:
```
print id(parList), id(childList)   # id() -- returns the storage address of th
e identifier
```

```
58329008 58329008
```

In [40]:
```
parList[4]
```

Out[40]: 54

In [41]:
```
parList[4] = 'Five Four'   # overwriting an elemnet
```

In [42]:
```
parList
```

Out[42]: [1, 2, 3, 4, 'Five Four', 5, 56, 6]

In [43]:
```
childList                  # modifications are reflected in childList
```

Out[43]: [1, 2, 3, 4, 'Five Four', 5, 56, 6]

In [44]:
```
childList[6]
```

Out[44]: 56

In [45]:
```
childList[6] = 'Five Six'
```

In [46]:
```
childList
```

Out[46]: [1, 2, 3, 4, 'Five Four', 5, 'Five Six', 6]

In [47]:
```
parList                    # modifications are reflected in parList
```

Out[47]: [1, 2, 3, 4, 'Five Four', 5, 'Five Six', 6]

In [ ]:
```
Reason: Here, two objects are
```

In [48]:
```
import copy
```

In [ ]:
```
copy.copy()       ->   Shallow COPY
copy.deepcopy()   ->   Deep COPY
```

In [49]:
```
parList = [12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming']]
 # re-assigning
```

In [50]:
```
print parList
```

```
[12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming']]
```

In [51]:
```
hardCopyList = parList   # Hard COPY or assignment operation
```

In [52]:
```
print hardCopyList
```

```
[12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming']]
```

In [53]:
```
shallowCopyList = copy.copy(parList)    # shallow COPY
```

In [54]:
```
deepCopyList = copy.deepcopy(parList)   # Deep COPY
```

In [56]:
```
print 'parList = %r \nhardCopyList = %r \nshallowCopyList = %r \ndeepCopyList
 = %r'%(parList, hardCopyList, shallowCopyList, deepCopyList)
```

```
parList = [12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming']]
hardCopyList = [12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programmin
g']]
shallowCopyList = [12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Program
ming']]
deepCopyList = [12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programmin
g']]
```

In [57]:
```
print 'id(parList) = %r \nid(hardCopyList) = %r \nid(shallowCopyList) = %r \ni
d(deepCopyList) = %r'%(id(parList), id(hardCopyList), id(shallowCopyList),
id(deepCopyList))
```

```
id(parList) = 58330168
id(hardCopyList) = 58330168
id(shallowCopyList) = 74579080
id(deepCopyList) = 58330968
```

With this, we can draw inference that shallowCopyList and deepCopyList are creating a new objects

In [58]:
```
parList == hardCopyList == shallowCopyList == deepCopyList
```

Out[58]: True

In [59]:
```
parList is hardCopyList is shallowCopyList is deepCopyList
```

Out[59]: False

In [60]:
```
parList is hardCopyList
```

Out[60]: True

In [61]:
```
parList is shallowCopyList
```

Out[61]: False

In [62]:
```
parList is deepCopyList
```

Out[62]: False

In [63]:
```
shallowCopyList is deepCopyList
```

Out[63]: False

In [64]:
```
parList
```

Out[64]: [12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming']]

In [65]:
```
parList[4]
```

Out[65]: True

In [66]:
```
parList[4] = False
```

In [67]:
```
parList
```

Out[67]: [12, 23.34, '1223', 'Python', False, [12, 23, '34', 'Programming']]

In [68]:
```
hardCopyList
```

Out[68]:
```
[12, 23.34, '1223', 'Python', False, [12, 23, '34', 'Programming']]
```

In [69]:
```
shallowCopyList
```

Out[69]:
```
[12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming']]
```

In [70]:
```
deepCopyList
```

Out[70]:
```
[12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming']]
```

Now, let us try in second dimension

In [71]:
```
parList
```

Out[71]:
```
[12, 23.34, '1223', 'Python', False, [12, 23, '34', 'Programming']]
```

In [72]:
```
parList[5]
```

Out[72]:
```
[12, 23, '34', 'Programming']
```

In [73]:
```
parList[5][3]
```

Out[73]:
```
'Programming'
```

In [74]:
```
parList[5][3]  = 'Scripting'
```

In [75]:
```
parList
```

Out[75]:
```
[12, 23.34, '1223', 'Python', False, [12, 23, '34', 'Scripting']]
```

In [77]:
```
hardCopyList
```

Out[77]:
```
[12, 23.34, '1223', 'Python', False, [12, 23, '34', 'Scripting']]
```

In [78]:
```
shallowCopyList             # Observe that shallow copyed list gets affected i
n second dimension
```

Out[78]:
```
[12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Scripting']]
```

In [79]:
```
deepCopyList
```

Out[79]:
```
[12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming']]
```

Now, let us try in 3rd dimension

In [81]:
```
parList = [12, 23.34, '1223', 'Python', True, [12, 23, '34', 'Programming', [5
6, 45.56, '98.45', 'Flask', ['Bottle']]]]
```

In [82]:
```
deepCopyList = copy.deepcopy(parList)
```

In [83]:
```
parList[5]
```

Out[83]:
```
[12, 23, '34', 'Programming', [56, 45.56, '98.45', 'Flask', ['Bottle']]]
```

In [84]:
```
parList[5][4]
```

Out[84]:
```
[56, 45.56, '98.45', 'Flask', ['Bottle']]
```

In [85]:
```
parList[5][4][1] = '23.23'
```

In [86]:
```
parList
```

Out[86]:
```
[12,
 23.34,
 '1223',
 'Python',
 True,
 [12, 23, '34', 'Programming', [56, '23.23', '98.45', 'Flask', ['Bottle']]]]
```

In [87]:
```
deepCopyList       # not modified
```

Out[87]:
```
[12,
 23.34,
 '1223',
 'Python',
 True,
 [12, 23, '34', 'Programming', [56, 45.56, '98.45', 'Flask', ['Bottle']]]]
```

In [88]:
```
parList[5][4][4]
```

Out[88]:
```
['Bottle']
```

In [89]:
```
parList[5][4][4] = 'Chimney'
```

In [90]:
```
parList
```

Out[90]:
```
[12,
 23.34,
 '1223',
 'Python',
 True,
 [12, 23, '34', 'Programming', [56, '23.23', '98.45', 'Flask', 'Chimney']]]
```

In [91]:
```
deepCopyList        # not modified
```

Out[91]:
```
[12,
 23.34,
 '1223',
 'Python',
 True,
 [12, 23, '34', 'Programming', [56, 45.56, '98.45', 'Flask', ['Bottle']]]]
```

Conclusions:

- In **single dimension** lists, if you do not want the copied list to get affected to the changes in source list, go for **shallow COPY.**
- In **multi-dimensional** lists, if you do not want the copied list to get affected to changes in source list, go for **deep COPY.**

Other Built-in functions

all() Verifies whether all the elements in the collection(list,tuple,set,dictionary) are True or False

any() Verifies whether any of the elements in the collection(list,tuple,set,dictionary) are True or False

In [95]:
```
l = [1, 2, -4, [34, 556, [56, 67, 0]]]
```

In [96]:
```
any(l)
```

Out[96]:
```
True
```

In [97]:
```
all(l)     # doesn't consider deep dimensions; only considers the first dimens
ion elements
```

Out[97]:
```
True
```

In [98]:
```
l.append(0)
```

In [99]:
```
l
```

Out[99]:
```
[1, 2, -4, [34, 556, [56, 67, 0]], 0]
```

In [100]:
```
all(l)
```

Out[100]:
```
False
```

Tuple

- Tuples are immutable (means can't be edited)
- Tuples have all the capabilities of lists, expect modification.
- Tuples can be indexed

In [102]:
```
t = ()   # empty tuple
```

In [103]:
```
print t, type(t)
```

```
() <type 'tuple'>
```

In [104]:
```
print dir(t)
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
 '__getslice__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__l
en__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex_
_', '__repr__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subcla
sshook__', 'count', 'index']
```

In [105]:
```
t2 = ('Apple', 'Mango', 'Goa')  # simple tuple
```

In [106]:
```
t3 = ('Apple', 123, 34.56, True)
```

Buit-in functions can be applied on them

In [108]:
```
len(t3)
```

Out[108]: 4

In [109]:
```
all(t2)
```

Out[109]: True

Indexing and slicing Tuples

In [110]:
```
print t2
```

```
('Apple', 'Mango', 'Goa')
```

In [111]:
```
print t2[2]
```

```
Goa
```

In [112]:
```
t3[:2]
```

Out[112]: ('Apple', 123)

In [113]:
```
t3[-1]
```

Out[113]: True

In [114]:
```
t3[1:]
```

Out[114]: (123, 34.56, True)

In [118]:
```
t3.count(True)
```

Out[118]: 1

In [119]:
```
t3.index(True)
```

Out[119]: 3

In [120]:
```
t3
```

Out[120]: ('Apple', 123, 34.56, True)

In [121]:
```
t3[:]
```

Out[121]: ('Apple', 123, 34.56, True)

In [122]:
```
t3[::]
```

Out[122]: ('Apple', 123, 34.56, True)

In [123]:
```
t3 is t3[:] is t3[::]
```

Out[123]: True

In [125]:
```
numbers = range(9); print numbers, type(numbers)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8] <type 'list'>
```

tuple() - built-in function for converting to tuple

In [127]:
```
nt = tuple(numbers)
```

In [128]:
```
print nt, type(nt)
```

```
(0, 1, 2, 3, 4, 5, 6, 7, 8) <type 'tuple'>
```

In [129]:
```
t2 + t3   # concatenation
```

Out[129]: ('Apple', 'Mango', 'Goa', 'Apple', 123, 34.56, True)

In [130]:
```
t2 + t3 != t3 + t2   # commutative Property not satisfied
```

Out[130]: True

In [131]:
```
t2*2  # Repition
```

Out[131]: ('Apple', 'Mango', 'Goa', 'Apple', 'Mango', 'Goa')

In [133]:
```
len(t2)
```

Out[133]: 3

In [134]:
```
t5 = t2[0:len(t2)-1]    # same as t2[0:2]
```

In [135]:
```
t5
```

Out[135]:
```
('Apple', 'Mango')
```

in

- Membership verification operator. Used on lists, tuples, strings, dictionaries

In [136]:
```
t2
```

Out[136]:
```
('Apple', 'Mango', 'Goa')
```

In [137]:
```
'Goa' in t2
```

Out[137]:
```
True
```

In [138]:
```
'Goas' in t2
```

Out[138]:
```
False
```

In [139]:
```
'Goas' not in t2
```

Out[139]:
```
True
```

In [140]:
```
t2+(3,4)
```

Out[140]:
```
('Apple', 'Mango', 'Goa', 3, 4)
```

In [141]:
```
t2+34
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-141-a11ba87efefe> in <module>()
----> 1 t2+34

TypeError: can only concatenate tuple (not "int") to tuple
```

In [ ]:
```
__Assigment :__ Try all the operations performed on Lists, to tuples, and obse
rve the difference
```

Tuples are Immutable

In [115]:
```
t3[2]
```

Out[115]:
```
34.56
```

In [117]:
```
t3[2] = 98.98
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-117-20f54bb1e418> in <module>()
----> 1 t3[2] = 98.98

TypeError: 'tuple' object does not support item assignment
```

In [132]:
```
t2**2   # power operation
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-132-7e012b19a351> in <module>()
----> 1 t2**2   # power operation

TypeError: unsupported operand type(s) for ** or pow(): 'tuple' and 'int'
```

Tuple are Immutable; So, they can't be edited. But, can be overwritten

Buit-in functions on Tuples

In [142]:
```
t1 = tuple(range(9)); t2 = tuple(range(3,12))
```

In [143]:
```
t1, t2
```

Out[143]: ((0, 1, 2, 3, 4, 5, 6, 7, 8), (3, 4, 5, 6, 7, 8, 9, 10, 11))

In [144]:
```
t3 = tuple(xrange(9))
```

In [145]:
```
t3
```

Out[145]: (0, 1, 2, 3, 4, 5, 6, 7, 8)

In [146]:
```
cmp(t1,t2)
```

Out[146]: -1

In [147]:
```
min(t2)
```

Out[147]: 3

In [148]:
```
max(t2)
```

Out[148]: 11

In [149]:
```
sorted(t2)   # convert any collection type, to list; then sorts; Creates new o
bject
```

Out[149]: [3, 4, 5, 6, 7, 8, 9, 10, 11]

In [150]:
```
list(t2)     # converting to list
```

Out[150]: [3, 4, 5, 6, 7, 8, 9, 10, 11]

Tuple Unpacking

In [151]:
```
a = 12
```

In [152]:
```
a,b  = 12, 23
```

In [153]:
```
print a
```

```
12
```

In [154]:
```
print b
```

```
23
```

In [155]:
```
(a,b,c,d,e) = 12, 23, 34, 45, 45
```

In [157]:
```
(a,b,c,d,e) = (12, 23, 34, 45, 45)
```

In [158]:
```
print a, type(a)
```

```
12 <type 'int'>
```

In [159]:
```
(a,b,c,d,e) = [12, 23, 34, 45, 45]
```

In [160]:
```
print a, type(a)
```

```
12 <type 'int'>
```

In [161]:
```
[a,b,c,d,e] = [12, 23, 34, 45, 45]     # List unpacking
```

In [162]:
```
print a, type(a)
```

```
12 <type 'int'>
```

Lists within Tuples

In [163]:
```
th = (12, 23, 34, [54, 54, 65,(23, 45), [34, 45]], ('python', 'programming'))
```

In [164]:
```
len(th), type(th)
```

Out[164]: (5, tuple)

In [165]:
```
th[4]
```

Out[165]:
```
('python', 'programming')
```

In [167]:
```
type(th[3]), type(th[4])
```

Out[167]:
```
(list, tuple)
```

In [168]:
```
th[3][0]
```

Out[168]:
```
54
```

In [169]:
```
th[3][0] = 'Five Four'
```

In [170]:
```
th
```

Out[170]:
```
(12,
 23,
 34,
 ['Five Four', 54, 65, (23, 45), [34, 45]],
 ('python', 'programming'))
```

Assignment 2: Write a Program to convert this heterogeneous tuple completely to flat tuple.

```
Input = (2,23, 34, [55, 'six six', (77, 88, ['nine nine', 0])])
Output = (2, 23, 34, 55, 'six six', 77, 88, 'nine nine', 0)
```

Tuples in list

In [171]:
```
myList = [12, 23, 45, (23, 45)]
```

In [172]:
```
myList[3]
```

Out[172]:
```
(23, 45)
```

In [173]:
```
myList[3][0]
```

Out[173]:
```
23
```

In [174]:
```
myList[3][0] = 'two three'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-174-39997b78eded> in <module>()
----> 1 myList[3][0] = 'two three'

TypeError: 'tuple' object does not support item assignment
```

Tuple Comprehensions (or) Generator expressions

In [183]:
```
tc = (i for i in range(9))
```

In [185]:
```
print tc
```

```
<generator object <genexpr> at 0x04785968>
```

In [186]:
```
type(tc)
```

Out[186]:
```
generator
```

In [187]:
```
len(tc)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-187-a4c5085a07fe> in <module>()
----> 1 len(tc)

TypeError: object of type 'generator' has no len()
```

In [188]:
```
print tc.next()
```

```
0
```

In [189]:
```
print tc.next()
```

```
1
```

In [190]:
```
print tc.next()
```

```
2
```

In [191]:
```
print tc.next()
```

```
3
```

In [192]:
```
print tc.next(), tc.next(), tc.next()
```

```
4 5 6
```

In [193]:
```
print tc.next(), tc.next(), tc.next()
```

```
7 8

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-193-eecdb7e2229b> in <module>()
----> 1 print tc.next(), tc.next(), tc.next()

StopIteration:
```

calling tc.next() when there is no value in that, results in StopIteration exception

In [194]:
```
tc = (i for i in range(9))
```

In [195]:
```
[i for i in tc]
```

Out[195]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8]
```

In [196]:
```
tl = [i for i in tc]
```

In [198]:
```
type(tl), type(tc)
```

Out[198]:
```
(list, generator)
```

In [199]:
```
[i.next() for i in tc]    # no exception, as there is no element in 'tc'
```

Out[199]:
```
[]
```

In [200]:
```
tc = (i for i in range(9))
```

In [201]:
```
[i.next() for i in tc]    # During iteration, elements become basic data types
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-201-e92a1f63326e> in <module>()
----> 1 [i.next() for i in tc]

AttributeError: 'int' object has no attribute 'next'
```

In [202]:
```
[type(i) for i in tc]
```

Out[202]:
```
[int, int, int, int, int, int, int, int]
```

In [203]:
```
tc = (i for i in [(12, 34), 12, 23, 'String', 'Python', True, 23.2])
```

In [204]:
```
[i for i in tc]
```

Out[204]:
```
[(12, 34), 12, 23, 'String', 'Python', True, 23.2]
```

In [206]:
```
print [type(i) for i in tc]
```

```
[]
```

In [207]:
```
tc = (i for i in [(12, 34), 12, 23, 'String', 'Python', True, 23.2])
```

In [208]:
```
print [type(i) for i in tc]
```

```
[<type 'tuple'>, <type 'int'>, <type 'int'>, <type 'str'>, <type 'str'>, <typ
e 'bool'>, <type 'float'>]
```

Assignment 3: practice all the exercises performed on list comprehensions, on tuple comprehensions