

# CS1010S

## Tutorial 7: List Processing

Nicholas Russell Saerang ([russellsaerang@u.nus.edu](mailto:russellsaerang@u.nus.edu))

# Table of contents

**1**

## Python Concepts

Lists

**2**

## Tutorial 7

List Processing



# Key Concepts

Lists



# Lists

Immutable sequences



# Tuples vs Lists

## Tuple sequences

- `()`, `(1,)` and `(2,3)`
- Immutable
- Can be used as **dict** keys

```
tup = (1,2,3)
tup[0] == 1 # True
tup[0] = 4   # TypeError
```

```
print(tup)   # (1,2,3)
```

## List sequences

- `[]`, `[1]` and `[2,3]`
- Mutable
- Cannot be used as **dict** keys

```
lst = [1,2,3]
lst[0] == 1 # True
lst[0] = 4
```

```
print(lst)   # [4,2,3]
```

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

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

```
lst.count(2)
```

```
lst.count(4)
```

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4)
```

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0
```

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |



# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0
```

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0

lst.reverse()
lst
lst.append(4)
lst.remove(2)
lst
```

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0

lst.reverse()
lst # [2,3,2,1]
lst.append(4)
lst.remove(2)
lst
```

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0
```

```
lst.reverse()
lst # [2,3,2,1]
lst.append(4)
lst.remove(2)
lst # [3,2,1,4]
```

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0
```

```
lst.reverse()
lst # [2,3,2,1]
lst.append(4)
lst.remove(2)
lst # [3,2,1,4]
```

```
lst = sorted(lst)
lst
lst.pop()
lst
```

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0
```

```
lst.reverse()
lst # [2,3,2,1]
lst.append(4)
lst.remove(2)
lst # [3,2,1,4]
```

```
lst = sorted(lst)
lst # [1,2,3,4]
lst.pop()
lst
```

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0
```

```
lst.reverse()
lst # [2,3,2,1]
lst.append(4)
lst.remove(2)
lst # [3,2,1,4]
```

```
lst = sorted(lst)
lst # [1,2,3,4]
lst.pop() # 4
lst
```

# List Methods

<https://docs.python.org/3/tutorial/datastructures.html>

| Tuple | List | Methods          |
|-------|------|------------------|
| ✓     | ✓    | count(x)         |
|       | ✓    | append(x)        |
|       | ✓    | extend(iterable) |
|       | ✓    | insert(I, x)     |
|       | ✓    | remove(x)        |
|       | ✓    | pop([i])         |
|       | ✓    | clear()          |
|       | ✓    | reverse()        |
|       | ✓    | copy()           |

```
lst = [1,2,3,2]
lst.count(2) # 2
lst.count(4) # 0
```

```
lst.reverse()
lst # [2,3,2,1]
lst.append(4)
lst.remove(2)
lst # [3,2,1,4]
```

```
lst = sorted(lst)
lst # [1,2,3,4]
lst.pop() # 4
lst # [1,2,3]
```



# Operations and more

```
# lists are mutable  
lst = [0]  
lst[0] = lst  
print(lst[0][0][0] == lst)  
print(lst)
```

# Operations and more

```
# lists are mutable
lst = [0]
lst[0] = lst
print(lst[0][0][0] == lst) # True
print(lst)
```

# Operations and more

```
# lists are mutable
lst = [0]
lst[0] = lst
print(lst[0][0][0] == lst) # True
print(lst) # [...]
```

# List operations are in-place

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

```
# Concatenation
```

```
print(lst + [4, 5])
```

```
print(lst)
```

```
# lst.extend(iterable)
```

```
print(lst.extend([4, 5]))
```

```
print(lst)
```

```
# lst.append(element)
```

```
print(lst.append(6))
```

```
print(lst)
```

# List operations are in-place

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

```
# Concatenation
```

```
print(lst + [4, 5])      # [1, 2, 3, 4, 5]
```

```
print(lst)               # [1, 2, 3]
```

```
# lst.extend(iterable)
```

```
print(lst.extend([4, 5]))
```

```
print(lst)
```

```
# lst.append(element)
```

```
print(lst.append(6))
```

```
print(lst)
```

# List operations are in-place

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

```
# Concatenation
```

```
print(lst + [4, 5])      # [1, 2, 3, 4, 5]
```

```
print(lst)               # [1, 2, 3]
```

```
# lst.extend(iterable)
```

```
print(lst.extend([4, 5])) # None
```

```
print(lst)               # [1, 2, 3, 4, 5]
```

```
# lst.append(element)
```

```
print(lst.append(6))
```

```
print(lst)
```

# List operations are in-place

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

```
# Concatenation
```

```
print(lst + [4, 5])      # [1, 2, 3, 4, 5]
```

```
print(lst)               # [1, 2, 3]
```

```
# lst.extend(iterable)
```

```
print(lst.extend([4, 5])) # None
```

```
print(lst)               # [1, 2, 3, 4, 5]
```

```
# lst.append(element)
```

```
print(lst.append(6))      # None
```

```
print(lst)               # [1, 2, 3, 4, 5, 6]
```

# Box-pointer diagram

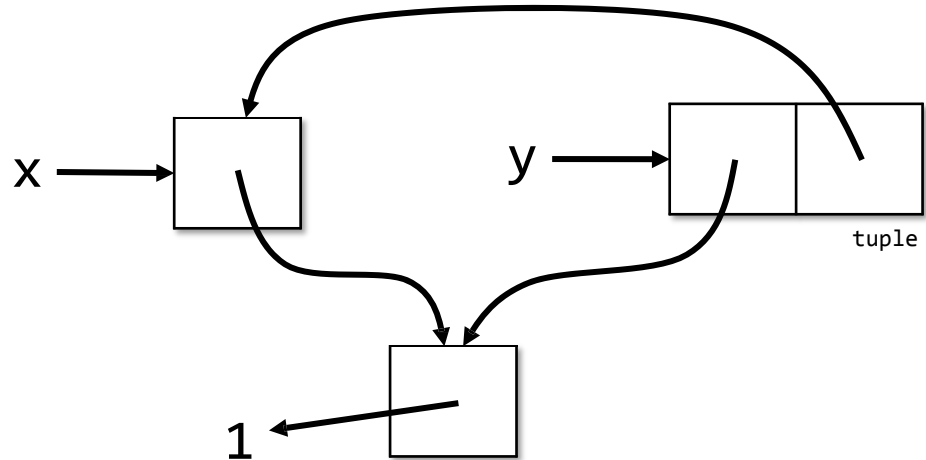
EXTREMELY USEFUL TOOL TO MODEL REFERENCE TYPE OBJECT

```
1.  x = [[1]]  
    y = (x[0], x)  
  
2.  x[0] += [y]  
    y[1].append(2)  
  
3.  y += (3,)  
    y[1][0] = y[0][0]  
    print(y is y[0][1])  
    print(x[0] in y)
```



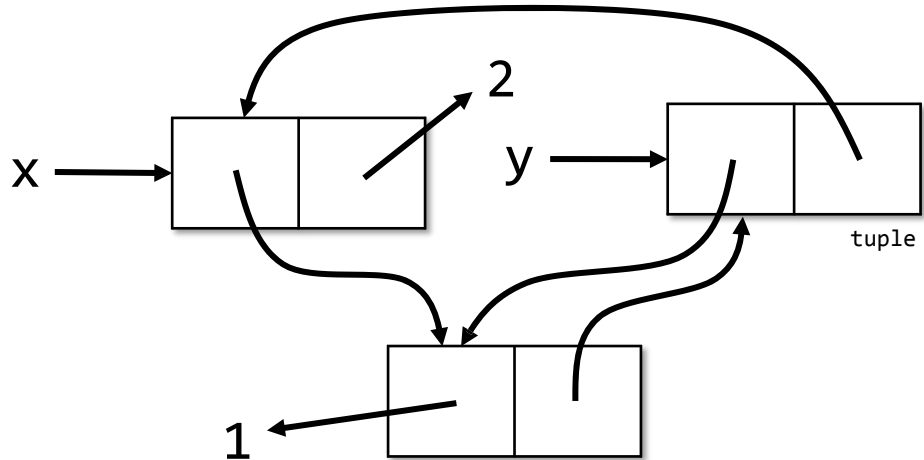
# Box-pointer diagram

```
1. x = [[1]]  
   y = (x[0], x)
```



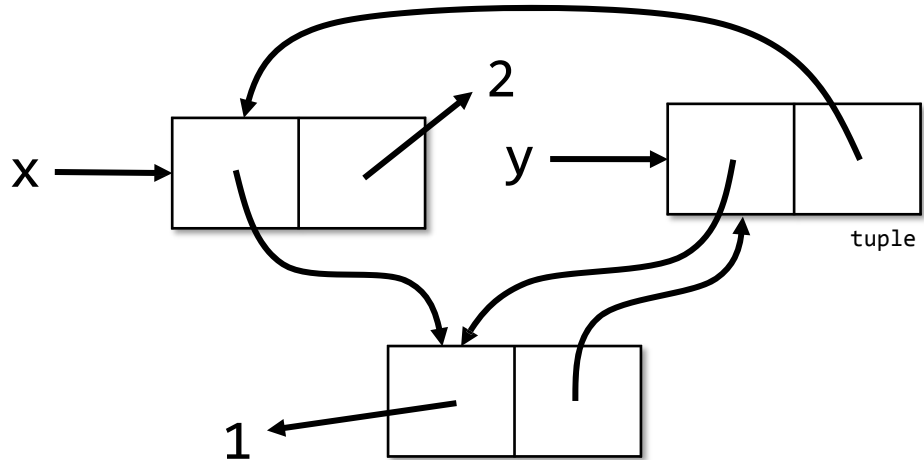
# Box-pointer diagram

```
2. x[0] += [y]
   y[1].append(2)
   print(y is y[0][1]) # True/False?
   print(x[0] in y) # True/False?
```



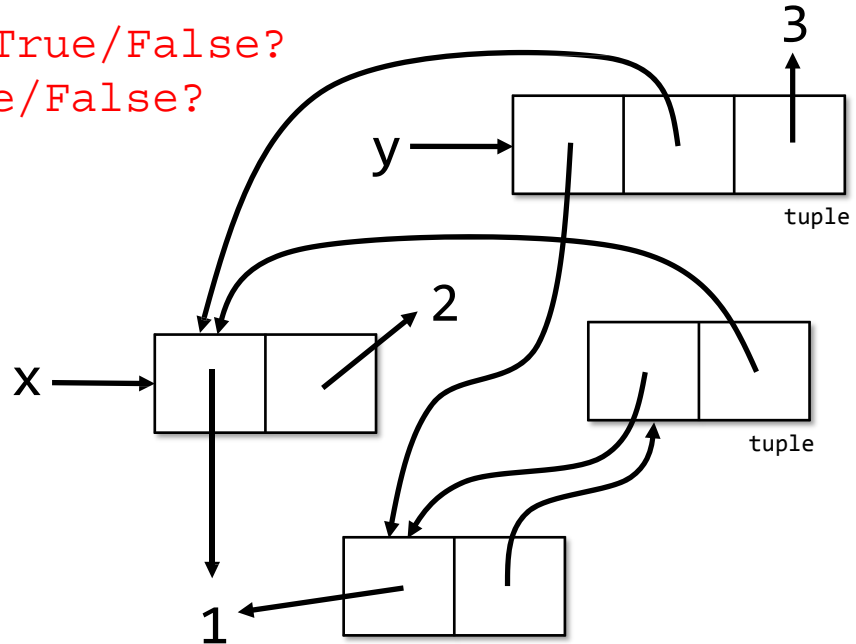
# Box-pointer diagram

```
2. x[0] += [y]
   y[1].append(2)
   print(y is y[0][1]) # True
   print(x[0] in y) # True
```



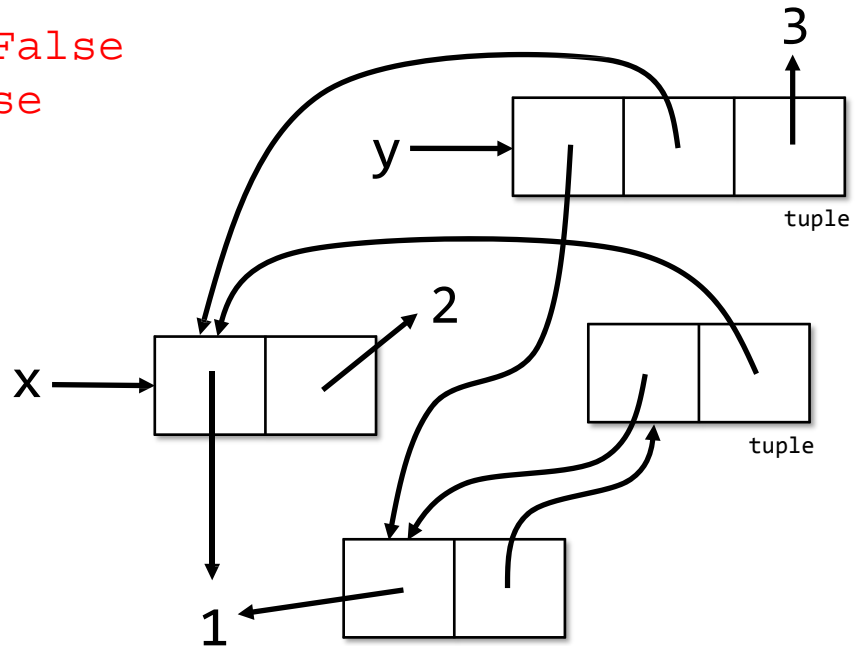
# Box-pointer diagram

```
3. y += (3,) # since y is a tuple  
y[1][0] = y[0][0]  
print(y is y[0][1]) # True/False?  
print(x[0] in y) # True/False?
```



# Box-pointer diagram

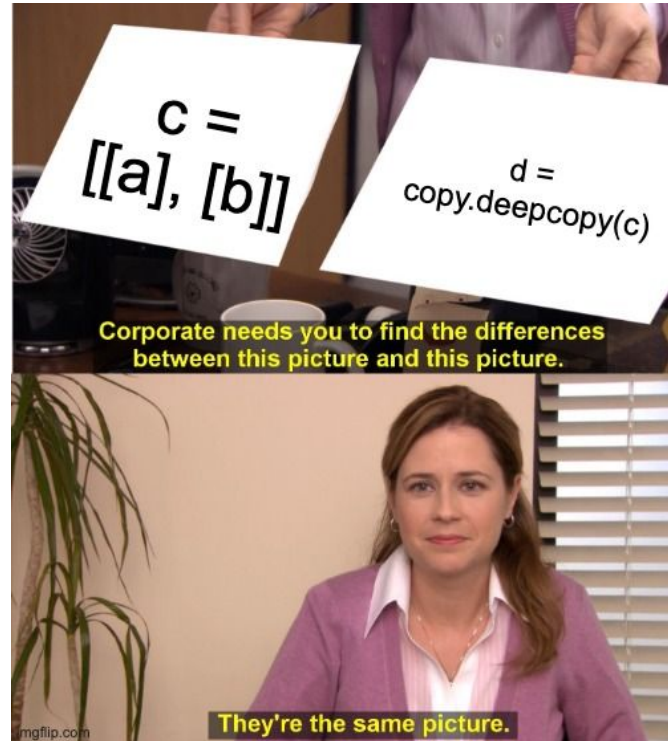
```
3. y += (3,) # since y is a tuple  
y[1][0] = y[0][0]  
print(y is y[0][1]) # False  
print(x[0] in y) # False
```



# Time to copy

Shallow Copy vs Deep Copy

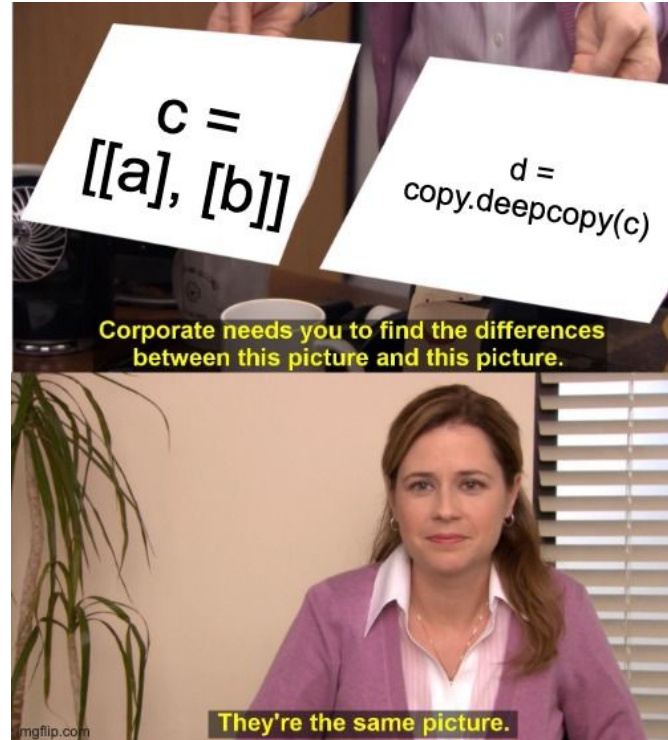
- Why do we need to deep copy?



# Time to copy

## Shallow Copy

```
>>> a = [[1,2]]  
>>> b = a.copy()  
>>> a[0] is b[0]  
True
```



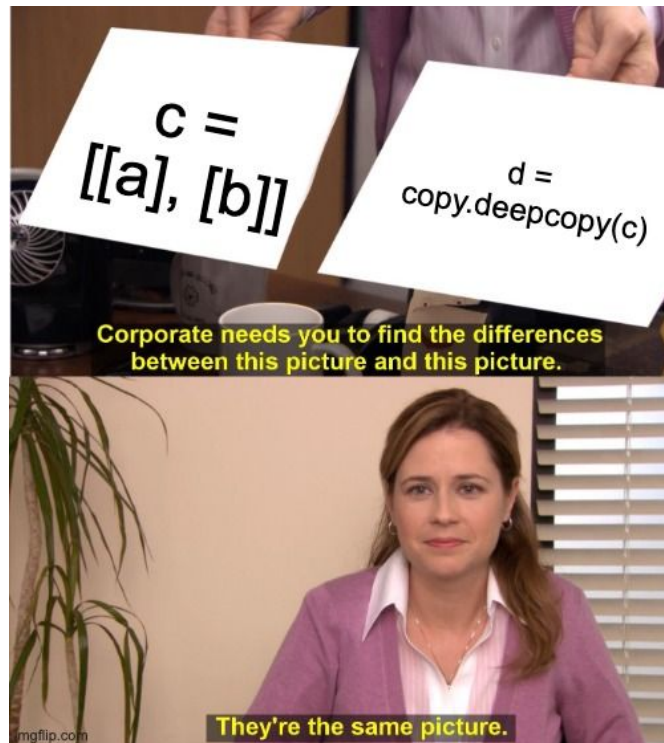
# Time to copy

Deep Copy

```
from copy import deepcopy
```

```
>>> a = [[1,2]]  
>>> b = deepcopy(a)  
>>> a[0] is b[0]  
False
```

```
# YOU ARE NOT ALLOWED TO USE  
# ANY LIBRARY IN CS1010S  
# (unless explicitly allowed)
```





# Tutorial 7

List Processing



# Question 1: At Least N

Ben Bitdiddle is required to implement a function `at_least_n` which takes in **a list of integers** and an integer `n`, and returns the original list with all the integers smaller than `n` removed.

```
>>> lst = list(range(0,10,1))
>>> lst2 = at_least_n(lst, 5)
>>> lst2
[5, 6, 7, 8, 9]
>>> lst is lst2
True
```

## Question 1a: At Least N

```
def at_least_n(lst, n):  
    for i in range(0, len(lst), 1):  
        if lst[i] < n:  
            lst.remove(lst[i])  
    return lst
```

```
at_least_n(list(range(0, 7, 1)), 5)
```

What's wrong with this implementation?

## Question 1a: At Least N

```
def at_least_n(lst, n):  
    for i in range(0, len(lst), 1):  
        if lst[i] < n:  
            lst.remove(lst[i])  
    return lst
```

```
at_least_n(list(range(0, 7, 1)), 5) # IndexError
```

What's wrong with this implementation?

```
# IndexError, size of list decreases
```

## Question 1b: At Least N

```
def at_least_n(lst, n):  
    for i in lst:  
        if i < n:  
            lst.remove(i)  
    return lst
```

```
at_least_n(list(range(0,7,1)), 5)
```

What's wrong with this implementation?

## Question 1b: At Least N

```
def at_least_n(lst, n):  
    for i in lst:  
        if i < n:  
            lst.remove(i)  
    return lst
```

```
at_least_n(list(range(0,7,1)), 5) # [1, 3, 5, 6]
```

What's wrong with this implementation?

# Index preserved across iterations

[Click here for python tutor](#)

## Question 1c: At Least N

```
def at_least_n(lst, n):  
    for i in list(lst):  
        if i < n:  
            lst.remove(i)  
    return lst
```

```
at_least_n(list(range(0,7,1)), 5)
```

## Question 1c: At Least N

```
def at_least_n(lst, n):  
    for i in list(lst):  
        if i < n:  
            lst.remove(i)  
    return lst
```

```
at_least_n(list(range(0,7,1)), 5) # [5, 6]
```

```
# In-place, iterate over list copy
```



## Question 1c: At Least N

```
def at_least_n(lst, n):  
    for i in reversed(range(0, len(lst), 1)):  
        if lst[i] < n:  
            lst.pop(i)  
    return lst
```

```
at_least_n(list(range(0, 7, 1)), 5)
```

## Question 1c: At Least N

```
def at_least_n(lst, n):  
    for i in reversed(range(0, len(lst), 1)):  
        if lst[i] < n:  
            lst.pop(i)  
    return lst
```

```
at_least_n(list(range(0, 7, 1)), 5)
```

```
# In-place, iterate from rear of list  
# reversed(range(0, 10, 1))  $\equiv$  range(9, -1, -1)
```

## Question 1c++: At Least N

```
def at_least_n(lst, n):  
    i = len(lst) - 1  
    while i >= 0:  
        if lst[i] < n:  
            lst.pop(i)  
            i -= 1  
    return lst
```

```
at_least_n(list(range(0, 7, 1)), 5)
```

## Question 1c++: At Least N

```
def at_least_n(lst, n):  
    i = len(lst) - 1  
    while i >= 0:  
        if lst[i] < n:  
            lst.pop(i)  
            i -= 1  
    return lst
```

```
at_least_n(list(range(0, 7, 1)), 5)
```

```
# In-place, iterate from rear of list
```

## Question 1d: At Least N

```
def at_least_n(lst, n):  
    return list(filter(lambda x: x>=n, lst))
```

```
# New list is created
```

```
lst = list(range(0,10,1))  
lst2 = at_least_n(lst , 5)  
lst is lst2 # False
```

## Question 1d: At Least N

```
def at_least_n(lst, n):  
    new_lst = []  
    for i in lst:  
        if i >= n:  
            new_lst.append(i)  
    return new_lst
```

# New list is created

```
lst = list(range(0,10,1))  
lst2 = at_least_n(lst , 5)  
lst is lst2 # False
```

## Question 2: Hanoi

In this question, you will be required to build a variant of the solution to the “Towers of Hanoi” problem presented in class. We define a *disk move* to be a pair of two numbers: the source pole and the destination pole. For example, (1, 3) indicates the move of a disk from the first pole to the third.

Implement a function called **hanoi** that takes in 4 parameters:

- the number of disks,
- the source pole,
- the destination pole,
- the auxiliary pole,

and *returns a list of disk moves (in tuple)* that, if executed in that sequence, will move all the disks from the source pole to the destination pole and comply with the rules of the Tower of Hanoi game. (Hint: your solution should not print a sequence of moves, since that has already been given in class).

## Question 2: Hanoi

Example execution:

```
>>> hanoi(1, 1, 2, 3)
[(1, 2)]
```

```
>>> hanoi(1, 1, 3, 2)
[(1, 3)]
```

```
>>> hanoi(3, 1, 2, 3)
[(1, 2), (1, 3), (2, 3), (1, 2), (3, 1), (3, 2), (1, 2)]
```



## Question 2: Hanoi tuples (printing)

```
def hanoi(n, src, dsc, aux):  
    if n == 0:  
        return  
    hanoi(n-1, src, aux, dsc) # Move n-1 to aux  
    print(f"{src} -> {dsc}") # Move nth to dsc  
    hanoi(n-1, aux, dsc, src) # Move n-1 to dsc  
  
>>> hanoi(2, 1, 2, 3)
```

## Question 2: Hanoi tuples (printing)

```
def hanoi(n, src, dsc, aux):  
    if n == 0:  
        return  
    hanoi(n-1, src, aux, dsc) # Move n-1 to aux  
    print(f"{src} -> {dsc}") # Move nth to dsc  
    hanoi(n-1, aux, dsc, src) # Move n-1 to dsc
```

```
>>> hanoi(2, 1, 2, 3)  
1 -> 3  
1 -> 2  
3 -> 2
```

## Question 2: Hanoi tuples (returning)

```
def hanoi(n, src, dsc, aux):  
    if n == 0:  
        return []  
    else:  
        return hanoi(n-1, src, aux, dsc)    # Concatenation  
            + [(src, dsc),]  
            + hanoi(n-1, aux, dsc, src)  
  
>>> hanoi(2, 1, 2, 3) # [(1, 3)] + [(1, 2)] + [(3, 2)]  
    [(1, 3), (1, 2), (3, 2)]
```

# Extra Questions

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

```
b = (1, 2, 3, a, 4, 5)
```


```
print(b)
```

```
a.clear()
```

```
print(b)
```

```
a = [1]
```

```
print(b)
```



# Extra Questions

```
a = [1, 2]
```

```
a += [a]
```


```
print(a)
```

```
b = a.copy() # shallow copy vs deep copy
```

```
a[2] = 0
```

```
print(a)
```

```
print(b)
```



# The End