Lecture 5, 23 August 2022

Iterating over on lists

- Compute sum and average (mean) of a list

In [1]:
```
def sumlist(l):
  sum = 0
  for x in l:
    sum = sum + x
  return(sum)

def average(l):
  if l == []:
    return
  return(sumlist(l)/len(l))
```

In [2]:
```
mylist = [15, 19, 3, 12, 22, 44]
sumlist(mylist), average(mylist)
```

Out[2]:
```
(115, 19.166666666666668)
```

- Compute values above the mean
  - Requires two passes over the list
- `aboveaverage` is an example of *filtering* a list
  - Extracting a sublist satisfying a certain property

In [3]:
```
def aboveaverage(l):
  if l == []:
    return
  avg = average(l)
  aboveavglist = []
  for x in l:
    if x >= avg:
      aboveavglist.append(x)
  return(aboveavglist)
```

In [4]:
```
aboveaverage(mylist)
```

Out[4]:
```
[22, 44]
```

- What happens if we don't store `average(l)` in `avg` ?

In [5]:
```
def aboveaverage(l):
  if l == []:
    return
  aboveavglist = []
  for x in l:
    if x >= average(l):
      aboveavglist.append(x)
  return(aboveavglist)
```

- Wasteful recomputation of `average(l)`

Many useful functions on lists are built-in to Python

In [6]:
```
sum([1,2,3,4,5]),max([1,2,3,4,5]),min([1,2,3,4,5])
```

Out[6]:
```
(15, 5, 1)
```

Nested loops

- find all elements common to `l1` and `l2`
  - for each `x` in `l1` , check if `x` is in `l2`
  - for each `y` in `l2` , check if `x == y`

In [7]:
```
def listcommon(l1,l2):
  commonlist = []
  for x in l1:          # In set theoretic terms, l1 x l2
    for y in l2:        # Nested loop - takes len(l1)*len(l2) steps
      if x == y:
        commonlist.append(x)
  return(commonlist)
```

In [8]:
```
listcommon([2,4,3,4],[3,4,7])
```

Out[8]:
```
[4, 3, 4]
```

In [9]:
```
listcommon([3,4,7],[2,4,3,4])
```

Out[9]:
```
[3, 4, 4]
```

- Nested loops can be expensive
- 10^8 operations take about 10 seconds in Python

In [10]:
```
i = 0
for x in range(10000):
  for y in range(10000):
    i = i+1
print(i)
```

```
100000000
```

- Can we use the same idea to check if l has duplicates?
- Nested loop over positions in the list rather than values of the list
- Be careful to generate each pair of positions `(i,j)` only once, inner loop starts from `i+1`

In [11]:
```
def checkduplicate(l):
  for i in range(len(l)):
    for j in range(i+1,len(l)):
      if l[i] == l[j]:
        return(True)
  return(False)  # Nested loop exited, no duplicates found
```

In [12]:
```
checkduplicate([3,2,3,2,3])
```

Out[12]:
```
True
```

Modify this to return a list of duplicates

- If there are more than 2 copies, duplicates get flagged multiple times

In [13]:
```
def checkduplicate2(l):
  duplist = []
  for i in range(len(l)):
    for j in range(i+1,len(l)):
      if l[i] == l[j]:
        duplist.append(l[i])
  return(duplist)
```

In [14]:
```
checkduplicate2([3,2,3,2,3])
```

Out[14]:
```
[3, 3, 2, 3]
```

- `x in l` returns `True` if `x` is an element of `l`
- Note that this is implicitly a loop running over all elements in `l`

In [15]:
```
def listcommon2(l1,l2):
  commonlist = []
  for x in l1:          # In set theoretic terms, l1 x l2
    if x in l2:         # Membership check, implicitly a loop
      commonlist.append(x)
  return(commonlist)
```

Compare the behaviour of the `listcommon` and `listcommon2` when there are duplicates in one or both lists

In [16]:
```
listcommon([2,4,3,4],[3,4,7]), listcommon2([2,4,3,4],[3,4,7])
```

Out[16]:
```
([4, 3, 4], [4, 3, 4])
```

In [17]:
```
listcommon([2,4,3],[3,4,4,7]), listcommon2([2,4,3],[3,4,4,7])
```

Out[17]:
```
([4, 4, 3], [4, 3])
```

In [18]:
```
listcommon([2,4,3,4],[3,4,4,7]), listcommon2([2,4,3,4],[3,4,4,7])
```

Out[18]:
```
([4, 4, 3, 4, 4], [4, 3, 4])
```

if-elif-else

- `sgn(x) = -1` if `x` is negative, `0` if `x` is `0`, `1` if `x` is positive
- Nested `if`, indentation increases
- `if`, `elif` ... `else`

In [19]:
```
def sgn(x):
  if x < 0:
    return(-1)
  else:
    if x == 0:
      return(0)
    else:
      return(1)
```

In [20]:
```
def sgn2(x):
  if x < 0:
    return(-1)
  elif x == 0:
    return(0)
  elif x > 0:
    return(1)
  else:
    return
```

In [21]:
```
sgn(-7),sgn(0),sgn(0.52),sgn2(3.5)
```

Out[21]:
```
(-1, 0, 1, 1)
```

True and False

- Other values can also be interpreted as True / False
- Numeric 0 is interpreted as False
- Empty list [] is interpreted as False
- Anything that is not intepreted as False is True

Intended use is to simplify conditionals like `if x == 0` or `if l != []`

In [22]:
```
def average2(l):
  if l:     # Does l evaluate to True, that is, is l != []?
    return(sum(l)/len(l))
```

In [23]:
```
print(average2([1,3,5,7]))
print(average2([]))
```

```
4.0
None
```

Behaviour can be unpredictable if non-booleans are used recklessly in boolean expressions

In [24]:
```
True + 7, 7 + True, 7 and 0, [] or 7, 7 or [], 8 and [3], [3] and 8
```

Out[24]:
```
(8, 8, 0, 7, 7, [3], 8)
```

Slice of list

- sublist from position `i` to position `j`
- `l[i:j]` is `[l[i],l[i+1],...,l[j-1]]`
- If `i <= i`, result is empty

In [25]:
```
mylist = list(range(100))
```

In [26]:
```
mylist[45:54]    # Slice from mylist[45] to mylist[53]
```

Out[26]:
```
[45, 46, 47, 48, 49, 50, 51, 52, 53]
```

Omitting an endpoint implicitly uses `0` or `len(l)` , as appropriate

In [27]:
```
mylist[:17], mylist[89:]   # If you leave out an endpoint it is assumed 0 / len(l)
```

Out[27]:
```
([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16],
 [89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99])
```

- Upper bound beyond `len(l)` truncates to `len(l)`
- Positions `-1` to `-n` are mapped to their positive equivalents
- Lower bound below `-n` truncates to `0`

In [28]:
```
mylist[90:101]   # Slice is more forgiving about positions out of range
```

Out[28]:
```
[90, 91, 92, 93, 94, 95, 96, 97, 98, 99]
```

In [29]:
```
mylist[7:5], mylist[-101:-97], mylist[-5]  # mylist[-5:5] is same as mylist[95:5]
```

Out[29]:
```
([], [0, 1, 2], 95)
```

In [30]:
```
mylist[-101:10]
```

Out[30]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Omitting both endpoints gives a *full slice*

In [31]:
```
mylist[:]
```

Out[31]:
```
[0,
 1,
 2,
 3,
 4,
 5,
 6,
 7,
 8,
 9,
 10,
 11,
 12,
 13,
 14,
 15,
 16,
 17,
 18,
 19,
 20,
 21,
 22,
 23,
 24,
 25,
 26,
 27,
 28,
 29,
 30,
 31,
 32,
 33,
 34,
 35,
 36,
 37,
 38,
 39,
 40,
 41,
 42,
 43,
 44,
 45,
 46,
 47,
 48,
 49,
 50,
 51,
 52,
 53,
 54,
 55,
 56,
 57,
 58,
 59,
 60,
 61,
 62,
 63,
 64,
 65,
 66,
 67,
 68,
 69,
 70,
 71,
 72,
 73,
 74,
 75,
 76,
 77,
 78,
 79,
 80,
 81,
 82,
 83,
```

```
 84,
 85,
 86,
 87,
 88,
 89,
 90,
 91,
 92,
 93,
 94,
 95,
 96,
 97,
 98,
 99]
```

Can provide a third parameter to a slice, like the step size in `range()`

In [32]:
```
mylist[0:100:15], mylist[:52:7], mylist[::10]
```

Out[32]:
```
([0, 15, 30, 45, 60, 75, 90],
 [0, 7, 14, 21, 28, 35, 42, 49],
 [0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

Mutable and immutable values

What is the value of `y` after the following code fragment?

In [33]:
```
x = 7
y = x
x = x+1
```

In [34]:
```
(x,y)
```

Out[34]:
```
(8, 7)
```

What are the values of `l1` and `l2` after the following code fragment?

In [35]:
```
l1 = [1,2,3]
l2 = l1
l2[0] = 8  # Reassign value at position 0 of l1 to 4
```

In [36]:
```
(l1,l2)
```

Out[36]:
```
([8, 2, 3], [8, 2, 3])
```

- When we assign `y = x` , the value is copied - *immutable value*
- When we assign `l2 = l1` , both names point to the same value - *mutable value*

How can we "safely" copy a list?

- Make a copy of `l1` in `l2` that does not point to the same value
- Any slice `l[i:j]` creates a new list
- Assign a full slice `l[:]`

In [37]:
```
l = [0,1,2,3,4,5,6,7,8,9]
```

In [38]:
```
l[:]
```

Out[38]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [39]:
```
l1 = [1,2,3]
l2 = l1[:]
l1[0] = 4  # Reassign value at position 0 of l1 to 4
l2[2] = 7
```

What are the values of `l1` and `l2` ?

In [40]:
```
(l1,l2)
```

Out[40]:
```
([4, 2, 3], [1, 2, 7])
```

Nested lists

- A list can contain lists as elements
- Use multiple subscripts to extract inner values

In [41]:
```
m = [ [10,11], [12,13]]
```

In [42]:
```
m[0],m[1]
```

Out[42]:
```
([10, 11], [12, 13])
```

In [43]:
```
m[0][0], m[0][1]
```

Out[43]:
```
(10, 11)
```

In [44]:
```
m[1][0]
```

Out[44]:
```
12
```

Pitfalls with mutability

- Multiple references to same list value

In [45]:
```
zerolist = [0,0]
matrix = [zerolist,zerolist]
```

In [46]:
```
matrix
```

Out[46]:
```
[[0, 0], [0, 0]]
```

In [47]:
```
matrix[0][0] = 7
```

In [48]:
```
matrix
```

Out[48]:
```
[[7, 0], [7, 0]]
```

In [49]:
```
zerolist
```

Out[49]:
```
[7, 0]
```

In [50]:
```
zerolist = [0,0]
matrix = [zerolist,zerolist[:]]
matrix[0][0] = 7
matrix, zerolist
```

Out[50]:
```
([[7, 0], [0, 0]], [7, 0])
```