Lecture 5, 04 October 2021

Operating on lists

Recall functions that operate on lists, using `for`

```
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))

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

# aboveaverage requires a second pass over the list, sequence of loops
```

`aboveaverage` is an example of *filtering* a list

- extracting a sublist satisfying a certain property

Many useful functions on lists are built-in to Python

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

```
(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`

```
def listcommon(l1,l2):
```

```
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)


listcommon([2,4,3,4],[3,4,7])
```

```
[4, 3, 4]
```

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

```
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`

```
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
```

Modify this to return a list of duplicates

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

```
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)


checkduplicate2([3,2,3,2,3])
```

```
[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`

```
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

```
listcommon([2,4,3,4],[3,4,7]),·listcommon2([2,4,3,4],[3,4,7])
```

```
([4, 3, 4], [4, 3, 4])
```

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

```
([4, 4, 3], [4, 3])
```

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

```
([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`

```
def sgn(x):
  if x < 0:
    return(-1)
  else:
    if x == 0:
      return(0)
    else:
      return(1)
```

```
def sgn2(x):
  if x < 0:
    return(-1)
  elif x == 0:
    return(0)
  elif x > 0:
    return(1)
```

```
  else:
    return

sgn(-7),sgn(0),sgn(0.52),sgn2(3.5)
```

```
(-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 != []`

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

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

```
4.0
None
```

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

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

```
(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 `j <= i`, result is empty

```
mylist = list(range(100))

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

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

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

```
mylist[:17], mylist[89:]   # If you leave out an endpoint it is assumed
```

```
([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`

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

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

```
mylist[7:5], mylist[-5:5], mylist[-5]  # mylist[-5:5] is same as mylist
```

```
([], [], 95)
```

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

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

Omitting both endpoints gives a *full slice*

```
mylist[:]
```

```
 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()`

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

```
([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

```
x = 7
y = x
x = x+1
# What is the value of y after this?
```

```
(x,y)
```

```
(8, 7)
```

```
l1 = [1,2,3]
l2 = l1
l1[0]·=·4· # Reassign value at position 0 to 4
# What are the values of l1 and l2?
```

```
(l1,l2)
```

```
([4, 2, 3], [4, 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[:]`

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

```
l[:]
```

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

```
l1 = [1,2,3]
l2 = l1[:]
l1[0] = 4  # Reassign value at position 0 to 4
# What are the values of l1 and l2?
```

```
(l1,l2)
```

```
([4, 2, 3], [1, 2, 3])
```

Nested lists

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

```
m = [ [10,11], [12,13]]
```

```
m[0],m[1]
```

```
([10, 11], [12, 13])
```

```
m[0][0], m[0][1]
```

```
(10, 11)
```

```
m[1][0]
```

```
12
```

Pitfalls with mutability and multiple references to same list value

```
zerolist = [0,0]
matrix = [zerolist,zerolist]
```

```
matrix
```

```
    [[0, 0], [0, 0]]
```

```
matrix[0][0] = 7
```

```
matrix
```

```
    [[7, 0], [7, 0]]
```

```
zerolist
```

```
    [7, 0]
```

Difference between `l.append(x)` and `l = l + [x]`

- `l.append(x)` modifies `l` in place
- `l = l + [x]` creates a new list `l`

```
l1 = [1,2,3]
l2 = l1
l1.append(4)
```

```
(l1,l2)
```

```
    ([1, 2, 3, 4], [1, 2, 3, 4])
```

```
l3 = [1,2,3]
l4 = l3
l3 = l3+[4]
```

```
(l3,l4)
```

```
    ([1, 2, 3, 4], [1, 2, 3])
```

```
l4[0]=5
```

```
l3,l4
```

```
    ([1, 2, 3, 4], [5, 2, 3])
```

Depending on the need, it may be useful to modify a list in place or return a modified list without changing the original

- `l.sort()` sorts a list in place
- `sorted(l)` returns a sorted copy of the list, leaving the original unchanged

```
mylist = [7,3,1,5,6]
```

```
mylist.sort()  # sorts in place
```

```
mylist
```

```
    [1, 3, 5, 6, 7]
```

```
mylist = [7,3,1,5,6]
sorted(mylist)  # Does not modify argument, returns sorted list
```

```
    [1, 3, 5, 6, 7]
```

```
mylist
```

```
    [7, 3, 1, 5, 6]
```