Programming and Data Structures with Python

Lecture 05, 31 December 2020

Indexing in lists

- Positions are **0** to **len(l)-1**
- Reverse indexing from the right: positions now are **-1, -2, ..., -len(l)**
- The last element of a list is **l[-1]**, easier than **l[len(l)-1]**

In [1]:

```
l = [0,1,2,3,4,5]
```

In [2]:

```
l[4]
```

Out[2]:

```
4
```

In [3]:

```
l[-3]
```

Out[3]:

```
3
```

Updating a list

- Update a position by assigning **l[i]**
- Update a slice by assigning **l[i:j]** (recall **l[i:j]** is from **l[i]** to **l[j-1]**)

In [4]:

```
l[3] = 33
```

In [5]:

```
l
```

Out[5]:

```
[0, 1, 2, 33, 4, 5]
```

In [6]:

```
l[1:3] = [11,22]
```

In [7]:

```
l
```

Out[7]:

```
[0, 11, 22, 33, 4, 5]
```

In [8]:

```
l[4:6] = [44,55,66]  # replace 2 positions by 3 values, l[4] and l[5]
```

In [9]:

```
l
```

Out[9]:

```
[0, 11, 22, 33, 44, 55, 66]
```

In [10]:

```
l[1:4] = [10,15,20,25,30]
```

In [11]:

```
l
```

Out[11]:

```
[0, 10, 15, 20, 25, 30, 44, 55, 66]
```

In [12]:

```
l[1:6] = [11,22,33]  # Contract 5 positions to 3
```

In [13]:

```
l
```

Out[13]:

```
[0, 11, 22, 33, 44, 55, 66]
```

In [14]:

```
l[7] = 77     #  Not a valid position
```

```
---------------------------------------------------------------
-----------------
IndexError                                Traceback (most
 recent call last)
<ipython-input-14-9193c936fa48> in <module>
----> 1 l[7] = 77     #  Not a valid position

IndexError: list assignment index out of range
```

In [16]:

```
l = [0, 11, 22, 33, 44, 55, 66]
```

In [17]:

```
l[2:5] = []  # Erase a slice
```

In [18]:

```
l
```

Out[18]:

```
[0, 11, 55, 66]
```

Mutable and immutable values

- When we assign the value, is it copied (immutable) or given an additional reference (mutable)
- Alternatively: if **x** holds an immutable value, and we update **x**, **x** now points to a new value. The old value does not change, what **x** is pointing to changes
- Instead: if **l** holds a mutable value and we update **l**, the value that **l** is pointing to changes, but **l** still points to the same "space", so any other reference to that space is also updated

implicitly

In [19]:

```
l1 = [0,1,2,3,4,5]
l2 = l1
```

In [20]:

```
l2[4]
```

Out[20]:

```
4
```

In [21]:

```
l1[4] = 44
```

In [22]:

```
l2[4]
```

Out[22]:

```
44
```

In [23]:

```
l = [[1,2],[3,4]]  # Is there a clever slice to get [[1],[3]]  -- I beli
```

In [24]:

```
l[1:2]
```

Out[24]:

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

In [25]:

```
# Example 1
l = [3,4]
y = l[0]
l[0] = 7
```

In [26]:

```
l, y
```

Out[26]:

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

In [27]:

```
# Example 2
l = [[1,2], [3,4]]
g = l[0]
g[0] = 7
```

In [28]:

```
l, g
```

Out[28]:

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

In [29]:

```
l[0][1] = 22
```

In [30]:

```
l, g
```

Out[30]:

```
([[7, 22], [3, 4]], [7, 22])
```

How do I copy a list?

Use a slice. Slice is always a *fresh* list

In [31]:

```
l1 = [0,1,2,3]
l2 = l1[0:len(l1)]  # "Full" slice
```

In [32]:

```
l1, l2
```

Out[32]:

```
([0, 1, 2, 3], [0, 1, 2, 3])
```

In [33]:

```
l1[3] = 33
```

In [34]:

```
l1, l2
```

Out[34]:

```
([0, 1, 2, 33], [0, 1, 2, 3])
```

In [35]:

```
# l1[:] is a shortcut for l1[0:len(l)]
l3 = l2[:]
```

In [36]:

```
l1, l2, l3
```

Out[36]:

```
([0, 1, 2, 33], [0, 1, 2, 3], [0, 1, 2, 3])
```

In [37]:

```
l2[1] = 11
```

In [38]:

```
l1, l2, l3
```

Out[38]:

```
([0, 1, 2, 33], [0, 11, 2, 3], [0, 1, 2, 3])
```

Using a for loop

In [39]:

```
l5 = [10,11,12,13]
l6 = []
for x in l5:
    l6 = l6 + [x]
```

In [40]:

```
l5, l6
```

Out[40]:

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

In [41]:

```
l5[0] = 100
```

In [42]:

```
l5, l6
```

Out[42]:

```
([100, 11, 12, 13], [10, 11, 12, 13])
```

Assignment creates a new list

In [43]:

```
l7 = [7,8,9]
l8 = l7
```

In [44]:

```
l7 = l7 + [10]   # Does this update to l7 affect l8? - No, we have reass
```

In [45]:

```
l7,l8
```

Out[45]:

```
([7, 8, 9, 10], [7, 8, 9])
```

In [46]:

```
# Instead
l9 = [9,10,11]
l10 = l9
l9.append(12)   # l.append(x) updates l in place
```

In [47]:

```
l9, l10
```

Out[47]:

```
([9, 10, 11, 12], [9, 10, 11, 12])
```

Equality

- **x == y** checks if **x** and **y** have the same value --- but need not be the same "box" in memory for mutable values
- **x is y** checks if **x** and **y** point to the same "box" in memory
- if **x is y** is **True**, then necessarily **x == y** is **True**, but not vice versa

In [48]:

```
l11 = [11,12,13]
l12 = l11[:]
l13 = l11
```

In [49]:

```
l11 == l12, l11 == l13, l12 == l13
```

Out[49]:

```
(True, True, True)
```

In [50]:

```
l11 is l12, l11 is l13, l12 is l13
```

Out[50]:

```
(False, True, False)
```

Passing parameters to functions

- **def f(a,b,c): ...** When **f** is called as **f(x,y,z)**, it as though we start with assignments **a = x**, **b = y**, **c = z**

In [51]:

```
def listupdate(l1,pos,y):  # Update l1[pos] = y
    l1[pos] = y
    return(True)
```

In [52]:

```
l = [1,2,3]
listupdate(l,0,11)
```

Out[52]:

```
True
```

In [53]:

```
l  # Has changed --- as though listupdate started with l1 = l, which is
```

Out[53]:

```
[11, 2, 3]
```

In [54]:

```
def factorial(n):
    ans = n
    while (n > 1):
        n = n - 1
        ans = ans * n
    return(ans)
```

In [55]:

```
factorial(11)
```

Out[55]:

```
39916800
```

In [56]:

```
m = 17
y = factorial(m)   # factorial starts with n = m
```

In [57]:

```
m, y   # No change in m since 17 was immutable
```

Out[57]:

```
(17, 355687428096000)
```

In [58]:

```
def listappend1(l,v):   # Append v to l
    l.append(v)
    return(l)

def listappend2(l,v):   # Append v to l, wrong
    l = l + [v]
    return(l)
```

In [59]:

```
l1 = [17]
listappend1(l1,18)
```

Out[59]:

```
[17, 18]
```

In [60]:

```
l1
```

Out[60]:

```
[17, 18]
```

In [61]:

```
listappend2(l1,19)
```

Out[61]:

```
[17, 18, 19]
```

In [62]:

```
l1  # Not updated since listappend2 created a new list l inside
```

Out[62]:

```
[17, 18]
```

Mutability and functions

It is useful to be able to update a list inside a function --- e.g. sorting it

- Built in list functions update in place
- l.append(v) -> in place versoin of l = l+[v]
- l.extend(l1) -> in place versoin of l = l + l1

In [63]:

```
l20 = [20,21,22]
l21 = [21,22,13]
l22 = l20
l20.extend(l21)
```

In [64]:

```
l20
```

Out[64]:

```
[20, 21, 22, 21, 22, 13]
```

In [65]:

```
l20.reverse()
```

In [66]:

```
l20
```

Out[66]:

```
[13, 22, 21, 22, 21, 20]
```

In [67]:

```
l20.sort()
```

In [68]:

```
l20
```

Out[68]:

```
[13, 20, 21, 21, 22, 22]
```

In [69]:

```
l22
```

Out[69]:

```
[13, 20, 21, 21, 22, 22]
```

In [70]:

```
l31 = [31,32]
l32 = l31
```

In [71]:

```
l31 = l31.append(33)  # Very bad, do not ever do this!!!
```

In [72]:

```
l32, l31
```

Out[72]:

```
([31, 32, 33], None)
```

In [73]:

```
mylist = [True,7.0,[3,4]]  # No need for list values to be of uniform ty
```