Lists, Tuples and Dictionaries

Lists

A list is a sequence of values. The values are separated by commas and enclosed within square brackets. The values in a list need not be of a uniform type. Unlike an array in C/C++/Java, the size of a list is flexible. This has an impact on the time it takes to access a given element. For more details on the distinction between arrays and lists, see this NPTEL lecture (https://nptel.ac.in/courses/106106131/9).

```
In [1]: l = [ 17 , 18 ,  "me",True]
```

```
In [2]: l
Out[2]: [17, 18, 'me', True]
```

Lists are sequences, like strings. We can extract a value at a position, or extract a slice.

```
In [3]: l[2]
Out[3]: 'me'
```

```
In [4]: l[0:3]
Out[4]: [17, 18, 'me']
```

A list value can be another sequence: say a string or a list. We can extract a position from a nested string or list with an extra layer of indexing.

```
In [5]: l[2]
Out[5]: 'me'
```

```
In [6]: l[2][1]
Out[6]: 'e'
```

```
In [7]: l2 = [[0,1,2],3,[4,5]]
In [8]: l2[2][0]
```

```
Out[8]: 4
```

Nested lists are commonly used to represent multidimensional matrices. For instance, the two dimensional matrix

```
0 0 0
0 0 0
0 0 0
```

would be written `[[0,0,0], [0,0,0], [0,0,0]]`

Unlike strings, lists can be updated in place. Lists are therefore **mutable**.

```
In [9]: l2[2] = [7,8]
```

```
In [10]: l2
```

```
Out[10]: [[0, 1, 2], 3, [7, 8]]
```

Assignment works differently for mutable and immutable values. If we write

```
x = 5
y = x
y = 7
```

the value of `x` remains `5`. We do not expect the value of `x` to change as a result of updating `y`. So, for immutable values, an assignment like `y = x` *copies* the value of `x` into `y`.

For mutable values, assignment creates an **alias**: the new name and the old name both point to the same value. This means an update via either name is reflected in both names.

```
In [11]: l3 = l2
```

```
In [12]: l3
```

```
Out[12]: [[0, 1, 2], 3, [7, 8]]
```

```
In [13]: l3[2] = [9,10]
In [14]: l3
```

```
Out[14]: [[0, 1, 2], 3, [9, 10]]
```

```
In [15]: l2
```

```
Out[15]: [[0, 1, 2], 3, [9, 10]]
```

One good reason why assigning mutable values results in aliasing has to do with passing arguments to functions. When we define a function

```
def f(a,b):
  ... Code for f ...
```

and we call it as `f(m,n)` , we effectively start executing the code defining `f` by first substituting the arguments `m` and `n` for the paramaters `a` and `b` .

```
a = m
b = n
... Code for f ...
```

When we pass a mutable value to a function, we would like changes made in the function to reflect outside the function. Hence, we would like this implicit assignment from actual parameters to formal parameters to operate as aliasing rather than copying. On the other hand, for immutable values, we would not like changes made to the parameters to have side-effects outside the function, so the implicit assignment from actual parameters to formal parameters should copy the values and not alias them.

Aliasing can have unexpected consequences. Before we see an example, here is some notation.

The operator `+` denotes concatenation for lists, as for strings. Since multiplication is repeated addition, we can write `l*3` (or `3*l`) for `l+l+l` .

```
In [16]: [1,2]*3
```

```
Out[16]: [1, 2, 1, 2, 1, 2]
```

```
In [17]: 3*[1,2]
```

```
Out[17]: [1, 2, 1, 2, 1, 2]
```

Let us try to set up a 3x3 matrix of zeros as follows.

```
In [18]: zerorow = [0]*3
         matrix = [zerorow]*3
```

```
In [19]: matrix
```

```
Out[19]: [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Let us now try to set the middle element, `matrix[1][1]` , to `5`

```
In [20]: matrix[1][1] = 5
```

Here is the result.

```
In [21]: matrix
```

```
Out[21]: [[0, 5, 0], [0, 5, 0], [0, 5, 0]]
```

What happened? Each entry of `matrix` is an alias to the same value `zerorow` . So, updating any one of the copies affects all copies.

How can we copy a list, rather than alias it? Recall that `l[i:j]` creates a new list by extracting a slice of `l` . Also, `l[:]` is the same as `l[0:len(l)]` , a *full slice*. We can use this to create a copy.

```
In [22]: l1 = [[0,1],2,[3,4]]
```

```
In [23]: l2 = l1[:]
```

```
In [24]: l2
```

```
Out[24]: [[0, 1], 2, [3, 4]]
```

```
In [25]: l2[2] = [5,6]
```

```
In [26]: l2
```

```
Out[26]: [[0, 1], 2, [5, 6]]
```

```
In [27]: l1
```

```
Out[27]: [[0, 1], 2, [3, 4]]
```

However, this is only a *shallow* copy. Nested lists are still aliased.

```
In [28]: l1[0][0] = 7
```

```
In [29]: l1
```

```
Out[29]: [[7, 1], 2, [3, 4]]
```

```
In [30]: l2
```

```
Out[30]: [[7, 1], 2, [5, 6]]
```

There appears to be no simple way to make a *deep* copy of a list, where nested lists are also copied rather than being aliased.

The comparison `==` checks that two values are equal. For mutable values, there are two possibilities: we have two distinct values that are the same, or both names are aliased to the same value. To check the latter, there is another comparison called `is` .

```
In [31]: l1 = [[7,1],2,[5,6]]
         l2 = l1[:]
         l3 = l1
```

```
In [32]: l1 == l2
```

```
Out[32]: True
```

```
In [33]: l1 == l3
```

```
Out[33]: True
```

```
In [34]: l1 is l2
```

```
Out[34]: False
```

```
In [35]: l1 is l3
```

```
Out[35]: True
```

Be careful. The comparison `is` is not reliable when dealing with immutable values.

```
In [36]: x = 5
         y = 5
```

In [37]:
```
x is y
```
Out[37]:
```
True
```

List functions

One way of adding a value `v` to a list `l` is to use concatenation. However, each time we reassign the value of `l` using `l = ...` we create a new value. Below, when `l1` is updated, it is no longer aliased to `l2`.

In [38]:
```
l1 = [1,2,3]
l2 = l1
v = 4
l1 = l1 + [v]
```
In [39]:
```
l1
```
Out[39]:
```
[1, 2, 3, 4]
```
In [40]:
```
l2
```
Out[40]:
```
[1, 2, 3]
```

To grow lists in place, we have two built-in functions, `l.append()` and `l.extend()`.

`l.append(v)` adds a single value `v` at the end of `l`, in place. In the example below, `l2` continues to be aliased to `l1` after the append operation.

In [41]:
```
l1 = [1,2,3,4]
l2 = l1
l1.append(5)
```
In [42]:
```
l1
```
Out[42]:
```
[1, 2, 3, 4, 5]
```
In [43]:
```
l2
```
Out[43]:
```
[1, 2, 3, 4, 5]
```

`l.extend(vlist)` adds a list of values at the end of `l`.

```
In [44]: l2.extend([6,7,8])
```

```
In [45]: l2
Out[45]: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
In [46]: l1
Out[46]: [1, 2, 3, 4, 5, 6, 7, 8]
```

`l.reverse()` reverses a list in place. In contrast, a reverse full slice `l[::-1]` creates a fresh copy of `l` in reverse.

```
In [47]: l1.reverse()
```

```
In [48]: l1
Out[48]: [8, 7, 6, 5, 4, 3, 2, 1]
```

```
In [49]: l2
Out[49]: [8, 7, 6, 5, 4, 3, 2, 1]
```

`l.index(v)` is similar to `s.index(p)` for strings. It returns the first position where `v` occurs in `l` and generates an error otherwise.

```
In [50]: l1.index(5)
Out[50]: 3
```

```
In [51]: l1.index(10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-51-d31a4b514c7a> in <module>
----> 1 l1.index(10)

ValueError: 10 is not in list
```

As usual, we can test whether `v` is in `l` before finding its position to avoid errors.

```
In [52]: if 10 in l1:
             print(l1.index(10))
         else:
             print(10, "not in", l1)
```

```
10 not in [8, 7, 6, 5, 4, 3, 2, 1]
```

`l.sort()` sorts `l` in ascending order. This works only if the list is of a uniform type so that all values are comparable to each other.

```
In [53]: l1
```

```
Out[53]: [8, 7, 6, 5, 4, 3, 2, 1]
```

```
In [54]: l1.sort()
```

```
In [55]: l1
```

```
Out[55]: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
In [56]: l3 = [[7,1],2,[5,6]]
         l3.sort()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-56-39416beeadf4> in <module>
      1 l3 = [[7,1],2,[5,6]]
----> 2 l3.sort()

TypeError: '<' not supported between instances of 'int' and 'list'
```

To sort in descending order, provide an optional argument `reverse=True`

```
In [57]: l1
```

```
Out[57]: [1, 2, 3, 4, 5, 6, 7, 8]
```

```
In [58]: l1.sort(reverse=True)
```

```
In [59]: l1
```

```
Out[59]: [8, 7, 6, 5, 4, 3, 2, 1]
```

We can also customize the way list elements are compared. We provide a function `f`. The input list `[x1,x2,...,xn]` is then transformed to `[f(x1),f(x2),...,f(xn)]` and sorting is done with respect the default ordering on these new values.

Lists are normally sorted in dictionary order. For instance:

```
In [60]: l3 = [[77, 1], [2], [3, 4]]
```

```
In [61]: l3.sort()
```

```
In [62]: l3
Out[62]: [[2], [3, 4], [77, 1]]
```

Suppose we want to sort lists in dictionary order from right to left. We first provide a function that returns a reversed list.

```
In [63]: def myreverse(l):
             return(l[::-1])
```

Now, we can pass `myreverse` to `sort` as follows.`

```
In [64]: l3.sort(key=myreverse)
```

```
In [65]: l3
Out[65]: [[77, 1], [2], [3, 4]]
```

We can combine the options `key` and `reverse`.

```
In [66]: l3.sort(key=myreverse, reverse=True)
```

```
In [67]: l3
Out[67]: [[3, 4], [2], [77, 1]]
```

Sometimes we wanted a sorted copy of a list without disturbing the list. For this use, the function `sorted()`

```
In [68]: l5 = [13,2,17,4,6]
         l6 = sorted(l5)
         print("l5:", l5, ", l6:", l6)

         l5: [13, 2, 17, 4, 6] , l6: [2, 4, 6, 13, 17]
```

Tuples

A tuple is an immutable sequence. Like a list, the elements of a tuple need not be of a uniform type. However, since tuples are immutable, the length is fixed. As usual, we can extract the value at some position in a tuple, or extract a slice.

```
In [69]: point1 = (0,0)
         point2 = (3,4)
         distance = ((point2[0]-point1[0])**2 + (point2[1]-point1[1])**2)**0.5

In [70]: distance

Out[70]: 5.0

In [71]: row = ("Madhavan", "05-07-2010",3243)

In [72]: row[0:2]

Out[72]: ('Madhavan', '05-07-2010')
```

One use for tuples is to return multiple values from a function

```
In [73]: def firstvowel(s):
             minvowel = ""
             minpos = len(s)
             for v in "aeiou":
                 p = s.find(v)
                 if p >= 0 and p < minpos:
                     minvowel = v
                     minpos = p
             return((minvowel,minpos))

In [74]: firstvowel("chennai")

Out[74]: ('e', 2)
```

In [75]:
```
firstvowel("rhythm")
```

Out[75]:
```
('', 6)
```

We can also assign a tuple of values in a single statement. For instance, we can shorten the function above as follows.

In [76]:
```
def firstvowel(s):
    (minvowel, minpos) = ("", len(s))
    for v in "aeiou":
        p = s.find(v)
        if p >= 0 and p < minpos:
            (minvowel, minpos) = (v, p)
    return((minvowel,minpos))
```

One special case is the following, which swaps two values without explicitly using a temporary name. In any assignment statement, remember that the names on the right refer to the *old* values and the ones on the left refer to the *new* values, so the order in which the tuple is assigned does not matter.

In [77]:
```
x = 5
y = 7
(x,y) = (y,x)
print("x is", x, "y is ", y)
```

```
x is 7 y is  5
```

Dictionaries

Exercise: *Write a function that takes a list of non-negative integers, possibly with duplicates, and reports the most frequently occurring integer.*

If we know the range of values in then input list, say `[0..N]` , we can use a list of size `N+1` where position `i` counts the number of occurrences of the integer `i` . (If there are multiple values with the highest frequency, we return the largest one.)

In [78]:
```
def frequency(l,maxval):
    # Initialize count[0..maxval] to 0
    count = []
    for i in range(maxval+1):
        count = count + []

    for i in l:
        count[i] = count[i] + 1

    maxval = 0
    maxcount = 0
    for i in range(maxval+1):
        if count[i] > maxcount:
            maxval = i
            maxcount = count[i]
    return((maxval,maxcount))
```

How can we do away with assumption that we know the range of values? Also, many numbers in the range `0..maxval` may never occur in the input list, so maintaining counts for them is wasteful.

In a list `l`, we have positions `0..len(l)-1` and with each position `i` we have a value `l[i]`. We can thus think of `l` as a function with domain `0..len(l)-1`. If we relax the constraint that we must have a continuous range of positions, we could have a more flexible structure that stores values for a subset of position: `p1 -> v1`, `p2 -> v2`, ..., `pn -> vn`. This is known as a key-value store: `[p1,p2,...,pn]` are the keys and `[v1,v2,...,vn]` are the associated values. In Python, key-value stores are called **dictionaries**.

A dictionary of the type above would be written as follows in Python: {p1:v1, p2:v2, ..., pn:vn}. An empty dictionary is written {}. To extract the value stored against key `k` in dictionary `d`, we write `d[k]`, similar to lists. Unlike a list, if we have a dictionary `d` and we want to a new key value pair `k:v`, we can just write `d[k] = v`. Recall that in a list `l` of length n, we cannot just write `l[n] = v` to add a (n+1)th element to a list.

Here is how we can rewrite the function `frequency` using dictionaries. A few points to note.

- To process all the keys in the dictionary `d`, we use the function `d.keys()`. In the current version of Python, this will run through all keys in the order in which they were added to `d`.
- In the list version, when determining the value with maximum frequency, we assumed initially that this value was 0 with frequency 0. Here, we have made no assumptions about the values in the list, so we use the first value of the input list `l[0]` as the initial guess for the most frequent value. To ensure that we do this only when `l` is non-empty, we initially check if `l` is the empty list.

In [79]:
```
def frequency(l):
    # Special case, empty list
    if l == []:
        return()

    # Maintain counters as a dictionary
    count = {}
    for i in l:
        if i in count.keys():
            count[i] = count[i] + 1
        else:
            count[i] = 1

    # Find and report the most frequent value
    maxval = l[0]
    maxcount = count[l[0]]
    for i in count.keys():
        if count[i] > maxcount:
            maxval = i
            maxcount = count[i]
    return((maxval,maxcount))
```

Like `d.keys()` , there is a function `d.values()` that generates all the values in `d` . Though we can use `d.keys()` and `d.values()` like a list in a `for` statement, these are not actually lists. (The same is true for the value returned by `range(i,j,k)` . To get a list from any of these, we have to use the function `list()` that converts its input argument into a list if the argument is of an appropriate type.

In [80]:
```
d = {1:1, 2:4, 3:9, 4:16}
print(d.keys())
```

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

In [81]:
```
d.keys() + [5]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-81-189516413f51> in <module>
----> 1 d.keys() + [5]

TypeError: unsupported operand type(s) for +: 'dict_keys' and 'list'
```

In [82]:
```
list(d.keys())+[5]
```

Out[82]:
```
[1, 2, 3, 4, 5]
```

To delete a key (and its associated value) from a dictionary use `del` . In general `del x` "undefines" the name `x` .

```
In [83]: del d[2]
         print(d)

{1: 1, 3: 9, 4: 16}
```

```
In [84]: y = 5
```

```
In [85]: del y
```

```
In [86]: y
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-86-9063a9f0e032> in <module>
----> 1 y

NameError: name 'y' is not defined
```

We can use any immutable value for a dictionary key. So a key can be a number, or a string, or a tuple, but not another dictionary or a list. However, the value stored against a key can be a dictionary or a list.

```
In [87]: runs = {'Kohli':[53,24,155]}
```

```
In [88]: runs['Kohli'].append(88)
         runs

Out[88]: {'Kohli': [53, 24, 155, 88]}
```

```
In [89]: marks = {'Test1':{'Anil':44, 'Mythili':66}, 'Test2':{'Anil':75, 'Mythili':73}}
         marks['Test2']['Anil']

Out[89]: 75
```

List comprehension

The following notation to create new sets from old sets is called **set comprehension**.

```
    { x**2 | x belongs to {0,1,2,3,..,10}, x is even }
```

This expression generates the set `{0,4,16,36,...,100}` of squares of even numbers between 0 and 10. We can break up the expression into three parts:

```
 {    x**2       | x belongs to {0,1,2,...10}, x is even }
  --transform--   -------generate-----------  --filter--
```

Here

- `generate` lists out candidates for the output set
- `filter` extracts the candidates that meet our criteria
- `transform` converts the candidates into the final form

In Python, we can write a similar expression to generate a list. This notation is called **list comprehension**.

In [90]:
```python
def even(x):
    return(x%2 == 0)

[ x**2 for x in range(11) if even(x)]
```

Out[90]:
```
[0, 4, 16, 36, 64, 100]
```

We use `for` to generate elements and `if` to filter, but there is no punctuation.

We can use multiple generators, which act like nested loops. The following definition produces all pairs `(i,j)` with `i` in [0..3] and `j` in [0..4] such that `i+j` is even. Notice that the loop generating `j` is nested within the loop generating `i`.

In [91]:
```python
[ (i,j) for i in range(4) for j in range(5) if even(i+j)]
```

Out[91]:
```
[(0, 0),
 (0, 2),
 (0, 4),
 (1, 1),
 (1, 3),
 (2, 0),
 (2, 2),
 (2, 4),
 (3, 1),
 (3, 3)]
```

Here is an expression that generates all valid Pythorean triples `(x,y,z)` with values below 20.

In [92]:
```
[ (x,y,z) for x in range(1,21)
          for y in range(1,21)
          for z in range(1,21) if (x*x + y*y) == z*z]
```

Out[92]:
```
[(3, 4, 5),
 (4, 3, 5),
 (5, 12, 13),
 (6, 8, 10),
 (8, 6, 10),
 (8, 15, 17),
 (9, 12, 15),
 (12, 5, 13),
 (12, 9, 15),
 (12, 16, 20),
 (15, 8, 17),
 (16, 12, 20)]
```

Later generators can depend on earlier ones. For instance, to avoid "duplicates" like `(3,4,5)` and `(4,3,5)` we can generate `y` starting from the current value of `x`.

In [93]:
```
[ (x,y,z) for x in range(1,21)
          for y in range(x,21)
          for z in range(1,21) if (x*x + y*y) == z*z]
```

Out[93]:
```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (8, 15, 17), (9, 12, 15), (12, 16, 20)]
```

Induction on lists

We can define a function `f` inductively on numbers by specifying `f(0)` and then giving a rule to compute `f(n)` from `f(n-1)`, `f(n-2)`, ..., `f(0)`. For instance:

```
factorial(0) = 1
factorial(n) = n * factorial(n-1)
```

We can also define functions on lists inductively. Here the base case is for the empty list and the inductive assumption is that we know how to compute the function for smaller lists. Here are some examples.

In [94]:
```
def mysum(l):
    if l == []:
        return(0)
    else:
        return(l[0] + sum(l[1:]))
```

```
In [95]: mysum([13,46,12])
```

```
Out[95]: 71
```

```
In [96]: def myreverse(l):
             if l == []:
                 return([])
             else:
                 return([l[-1]] + myreverse(l[0:-1]))
```

```
In [97]: myreverse([13,46,79])
```

```
Out[97]: [79, 46, 13]
```