13A. Lists of Numbers

Topics:

- Lists of numbers
- List Methods:
- Void vs Fruitful Methods
- Setting up Lists
- A Function that returns a list

We Have Seen Lists Before

Recall that the rgb encoding of a color involves a triplet of numbers:

```
MyColor = [.3,.4,.5]

DrawDisk(0,0,1,FillColor = MyColor)
```

`MyColor` is a list.

A list of numbers is a way of assembling a sequence of numbers.

Terminology

```
x = [3.0, 5.0, -1.0, 0.0, 3.14]
```

How we talk about what is in a list:

5.0 is an item in the list **x**.

5.0 is an entry in the list **x**.

5.0 is an element in the list **x**.

5.0 is a value in the list **x**.

Get used to the synonyms.

A List Has a Length

The following would assign the value of 5 to the variable n:

```
x = [3.0, 5.0, -1.0, 0.0, 3.14]
n = len(x)
```

The Entries in a List are Accessed Using Subscripts

The following would assign the value of `-1.0` to the variable `a`:

```
x = [3.0, 5.0, -1.0, 0.0, 3.14]
a = x[2]
```

A List Can Be Sliced

This:

```
x = [10,40,50,30,20]
y = x[1:3]
z = x[:3]
w = x[3:]
```

Is same as:

```
x = [10,40,50,30,20]
y = [40,50]
z = [10,40,50]
w = [30,20]
```

Lists are Similar to Strings

s:	'x'	'L'	'1'	'?'	'a'	'C'

x:	3	5	2	7	0	4

A string is a sequence of characters.

A list of numbers is a sequence of numbers.

Lists in Python

Now we consider lists of numbers:

```
A = [10,20,30]
B = [10.0,20.0,30.0]
C = [10,20.0,30]
```

The items in a list usually have the same type, but that is not required.

Soon we will consider lists of strings:

```
Animals = ['cat','dog','mouse']
```

Later we will consider lists of objects.

The operations on lists that we are about to describe will be illustrated using lists of numbers. But they can be applied to any kind of list.

Visualizing Lists

Informal: **x:**

0	1	2	3
3	5	1	7

Formal: **x** ---->

```
0 ---> 3
1 ---> 5
2 ---> 1
3 ---> 7
```

A state diagram that shows the "map" from indices to elements.

Lists vs. Strings

There are some similarities, e.g., subscripts

But there is a huge difference:

1. Strings are immutable. They cannot be changed.
2. Lists are mutable. They can be change.

Exactly what does this mean?

Strings are Immutable

Before: `s:`

0	1	2	3
'a'	'b'	'c'	'd'

```
s[2]= 'x'
```

After:

```
TypeError: 'str' object does
not support item assignment
```

You cannot change the value of a string

Lists ARE Mutable

Before: **x:**

0	1	2	3
3	5	1	7

```
x[2] = 100
```

After: **x:**

0	1	2	3
3	5	100	7

You can change the values in a list

Lists ARE Mutable

Before **x:**

0	1	2	3
3	5	1	7

```
x[1:3] = [100,200]
```

After **x:**

0	1	2	3
3	100	200	7

You can change the values in a list

List Methods

When these methods are applied to a list, they affect the list.

`append`

`extend`

`insert`

`sort`

Let's see what they do through examples...

List Methods: `append`

Before: `x:`

0	1	2	3
3	5	1	7

```
x.append(100)
```

After: `x:`

0	1	2	3	4
3	5	1	7	100

Use append when you want to “glue” an item on the end of a given list.

List Methods: `extend`

Before: **x:**

0	1	2	3
3	5	1	7

```
t = [100,200]
x.extend(t)
```

After: **x:**

0	1	2	3	4	5
3	5	1	7	100	200

Use extend when you want to “glue” one list onto the end of another list.

List Methods: `insert`

Before: **x:**

0	1	2	3
3	5	1	7

```
i = 2
a = 100
x.insert(i,a)
```

After: **x:**

0	1	2	3	4
3	5	100	1	7

Use insert when you want to insert an item into the list. Items get "bumped" to the right if they are at or to the right of the specified insertion point.

List Methods: `sort`

Before:

x:

0	1	2	3
3	5	1	7

```
x.sort()
```

After:

x:

0	1	2	3
1	3	5	7

Use sort when you want to order the elements in a list from little to big.

List Methods: `sort`

Before: `x:`

0	1	2	3
3	5	1	7

```
x.sort(reverse=True)
```

An optional argument is being used to take care of this situation.

After: `x:`

0	1	2	3
7	5	3	1

Use sort when you want to order the elements in a list from big to little.

Void Methods

When the methods

`append` **`extend`** **`insert`** **`sort`**

are applied to a list, they affect the list but they do not return anything like a number or string. They are called "void" methods.

Void methods return the value of **`None`**. This is Python's way of saying they do not return anything.

Void Methods

A clarifying example:

```
>>> x = [10,20,30]
>>> y = x.append(40)
>>> print x
[10, 20, 30, 40]
>>> print y
None
```

`x.append(40)` does something to x.

In particular, it appends an element to x

It returns None and that is assigned to y.

Void Methods/Functions

The graphics procedures **`DrawDisk`**, **`DrawRect`**, etc., are examples of void functions.

They also return the value None. But we were never tempted to do something like this:

```
C = DrawDisk(0,0,1)
```

With lists, however, it is tempting to do something like this:

```
newValue = 10
y = x.append(newValue)
```

So we have to be careful!

(Fruitful) List Methods

When these methods are applied to a list, they actually return something:

`pop`
`count`

Let's see what they do through examples...

The List Method `pop`

Before:

x:

0	1	2	3
3	5	1	7

```
i = 2
m = x.pop(i)
```

After:

x:

0	1	2
3	5	7

m:

1

Use pop when you want to remove an element and assign it to a variable.

The List Method count

Before:

	0	1	2	3
x:	3	7	1	7

```
m = x.count(7)
```

After:

	0	1	2	3
x:	3	7	1	7

m:	2

Use count when you want to compute the number of items in a list that have a value.

Two Built-In Functions that Can be Applied to Lists

`len` returns the length of a list

`sum` returns the sum of the elements in a list provided all the elements are numerical.

len and sum

Before

	0	1	2	3
x:	3	7	1	5

```
m = len(x)
s = sum(x)
```

After

	0	1	2	3
x:	3	7	1	5

m:	4
s:	16

len and sum: Common errors

```
>>> x = [10,20,30]

>>> s = x.sum()
AttributeError: 'list' object
    has no attribute 'sum`

>>> n = x.len()
AttributeError: 'list' object
    has no attribute 'len'
```

Legal But Not What You Probably Expect

```
>>> x = [10,20,30]
>>> y = [11,21,31]
>>> z = x+y
>>> print z
[10,20,30,11,21,31]
```

Legal But Not What You Probably Expect

```
>>> x = [10,20,30]
>>> y = 3*x
>>>print y
[10,20,30,10,20,30,10,20,30]
```

Setting Up "Little" Lists

The examples so far have all been small.

When that is the case, the "square bracket" notation is just fine for setting up a list:

```
x = [10,40,50,30,20]
```

Don't forget the commas!

Working with Big Lists

Setting up a big list requires a loop.

Looking for things in a big list requires a loop.

Let's consider some examples.

A Big List of Random Numbers

```
from random import randint as randi
x = []
N = 1000000
for k in range(N):
    r = randi(1,6)
    x.append(r)
```

x starts out as an empty list and is built up through repeated appending.

Roll a dice one million times. Record the outcomes in a list.

This Does Not Work

```
from random import randint as randi
x = []
N = 1000000
for k in range(N):
    r = randi(1,6)
    x[k] = r
```

```
x[k] = r
IndexError: list assignment index out of range
```

`x[0] = r` does not work because x is the empty list—it has no components

A List of Square Roots

```
from math import sqrt
x = []
N = 1000000
for k in range(N):
    s = sqrt(k)
    x.append(s)
```

Same idea. Create a list through repeated appending.

A Random Walk Example

```
from random import randint as randi
x = [0]
k = 0
# x[k] is robot's location after k hops
while abs(x[k])<=10:
    # Flip a coin and hop right or left
    r = randi(1,2)
    if r==1:
        new_x = x[k]+1
    else:
        new_x = x[k]-1
    k = k+1
    x.append(new_x)
```

A Random Walk Example

Notice that x is initialized as a length-1 list. The robot starts at the origin.

```
from random import randint as randi
x = [0]
k = 0
# x[k] is robot's location after k hops
while abs(x[k])<=10:
    # Flip a coin and hop right or left
    r = randi(1,2)
    if r==1:
        new_x = x[k]+1
    else:
        new_x = x[k]-1
    k = k+1
    x.append(new_x)
```

Be Careful About Types

This is OK and synonymous with `x = [0,10]`:

```
x = [0]
x.append(10)
```

You need the square brackets. It is your way of telling Python that **x** is a list, not an int.

This is not OK:

```
x = 0
x.append(10)
```

```
AttributeError: 'int' object has
     no attribute 'append'
```

Be Careful About Types

```
>>> x = 0
>>> type(x)
<type 'int'>
>>> x = [0]
>>> type(x)
<type 'list'>
```

Functions and Lists

Let's start with a function that returns a list.

In particular, a function that returns a list of random integers from a given interval.

Then we will use that function to estimate various probabilities when a pair of dice are rolled.

A List of Random Integers

```
from random import randint as randi

def randiList(L,R,n):
    """ Returns a length-n list of
    random integers from interval [L,R]
    PreC: L,R,n ints with L<=R and n>=1
    """
    x = []
    for k in range(n):
        r = randi(L,R)
        x.append(r)
    return x
```

Outcomes from Two Dice Rolls

Roll a pair of dice N times

Store the outcomes of each dice roll
in a pair of length-N lists.

Then using those two lists, create a third
list that is the sum of the outcomes
in another list.

Outcomes from Two Dice Rolls

Example:

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

	0	1	2	3
D:	5	4	9	6

How to Do It

```
N = 1000000
D1 = randiList(1,6,N)
D2 = randiList(1,6,N)

D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

How It Works

k --> 0

N --> 4

D1:

0	1	2	3
2	1	5	4

D2:

0	1	2	3
3	3	4	2

At the start of the loop

D: []

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

How It Works

k --> 0

N --> 4

TwoThrows --> 5

TwoThrows = D1[0]+D2[0]

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

D: []

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

How It Works

k --> 0

N --> 4

TwoThrows --> 5

D.append(5)

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

D:	5

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

How It Works

k --> 1

N --> 4

TwoThrows --> 4

TwoThrows= D1[1]+D2[1]

	0	1	2	3
D1:	2	1	5	4
D2:	3	3	4	2
D:	5			

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

How It Works

k --> 1

N --> 4

TwoThrows --> 4

D.append(4)

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

D:	5	4

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
  ● D.append(TwoThrows)
```

How It Works

k --> 2

N --> 4

TwoThrows --> 9

D1:

0	1	2	3
2	1	5	4

D2:

0	1	2	3
3	3	4	2

D:

5	4

TwoThrows= D1[2]+D2[2]

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

How It Works

k --> 2

N --> 4

TwoThrows --> 9

D.append(9)

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

D:	5	4	9

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
  ● D.append(TwoThrows)
```

How It Works

k --> 3

N --> 4

TwoThrows --> 9

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

D:	5	4	9

TwoThrows = D1[3]+D2[3]

```
N = 4
D = []
for k in range(N):
  ● TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

How It Works

k --> 3

N --> 4

TwoThrows --> 6

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

D:	5	4	9

TwoThrows = D1[3]+D2[3]

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

How It Works

k --> 3

N --> 4

TwoThrows --> 6

D.append(6)

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

D:	5	4	9	6

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
  ● D.append(TwoThrows)
```

How It Works

k --> 4

N --> 4

TwoThrows --> 6

All Done!

	0	1	2	3
D1:	2	1	5	4

	0	1	2	3
D2:	3	3	4	2

D:	5	4	9	6

```
N = 4
D = []
for k in range(N):
    TwoThrows = D1[k] + D2[k]
    D.append(TwoThrows)
```

Tabulating Outcomes

We have simulated the rolling of a pair of dice N times.

The outcomes are recorded in the list D.

New problem:

How many 2's were there?
How many 3's were there?
:
How many 12's were there?

Tabulating Outcomes

```
count = [0,0,0,0,0,0,0,0,0,0,0,0,0]
for k in range(N):
   i = D[k]
   count[i] = count[i]+1
```

	0	1	2	3	4	5	6	7	8	9	10	11	12
`count:`	0	0	0	0	0	0	0	0	0	0	0	0	0

`count[2]` keeps track of the number of 2's thrown

`count[10]` keeps track of the numberof 10's thrown

Tabulating Outcomes

```
count = [0,0,0,0,0,0,0,0,0,0,0,0,0]
for k in range(N):
   i = D[k]
   count[i] = count[i]+1
```

The variable i is assigned the outcome of the k-th 2-die roll.

Tabulating Outcomes

```
count = [0,0,0,0,0,0,0,0,0,0,0,0,0]
for k in range(N):
   i = D[k]
   count[i] = count[i]+1
```

Suppose:

`i --> 7`

	0	1	2	3	4	5	6	7	8	9	10	11	12
count:	0	0	3	1	5	8	7	2	1	6	9	2	1

Tabulating Outcomes

```
count = [0,0,0,0,0,0,0,0,0,0,0,0,0]
for k in range(N):
   i = D[k]
   count[i] = count[i]+1
```

Suppose `i --> 7`

then the assignment `count[i] = count[i]+1`

effectively says `count[7] = count[7]+1`

Tabulating Outcomes

```
count = [0,0,0,0,0,0,0,0,0,0,0,0,0]
for k in range(N):
   i = D[k]
   count[i] = count[i]+1
```

`i -->` **7**

Before:

	0	1	2	3	4	5	6	7	8	9	10	11	12
count:	0	0	3	1	5	8	7	2	1	6	9	2	1

After:

	0	1	2	3	4	5	6	7	8	9	10	11	12
count:	0	0	3	1	5	8	7	3	1	6	9	2	1

Overall...

```
count = [0,0,0,0,0,0,0,0,0,0,0,0,0]
for k in range(N):
   i = D[k]
   count[i] = count[i]+1
```

A list of counters.

Sample Results, N = 10000

```
for k in range(2,13):
    print k,count[k]
```

```
 k    count[k]
----------------
  2     293
  3     629
  4     820
  5    1100
  6    1399
  7    1650
  8    1321
  9    1149
 10     820
 11     527
 12     292
```