6. Lists of Numbers

Topics:

- Lists of numbers
- List Methods:
- Void vs Fruitful Methods
- Setting up Lists

We Have Seen Lists Before

```
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 thesynonyms.

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

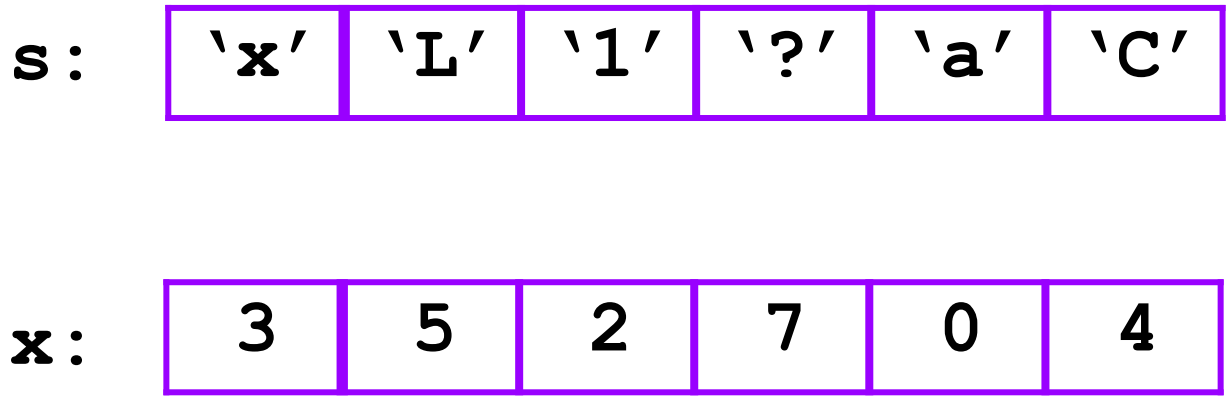
This:

```
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



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.

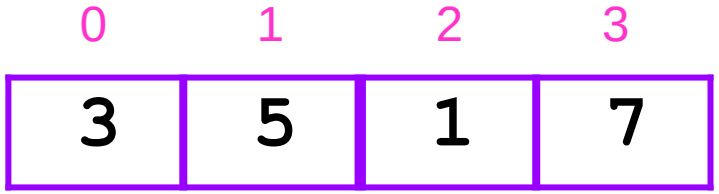
Soon we will consider lists of strings:

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

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:



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

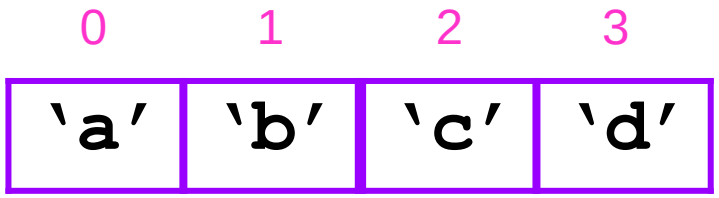
But there is a huge difference:

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[2]= 'x'
```

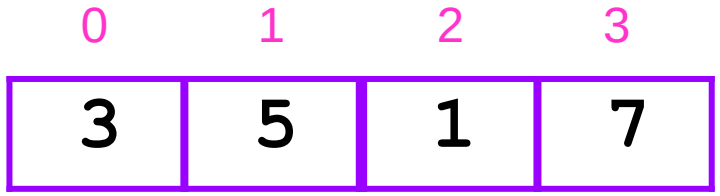
After:

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

You cannot change the value of a string

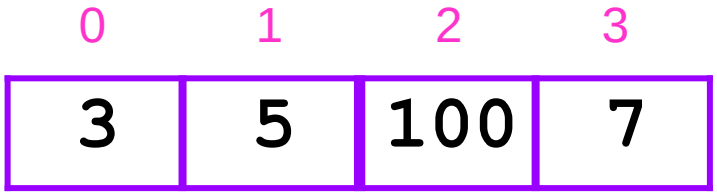
Lists ARE Mutable

Before:



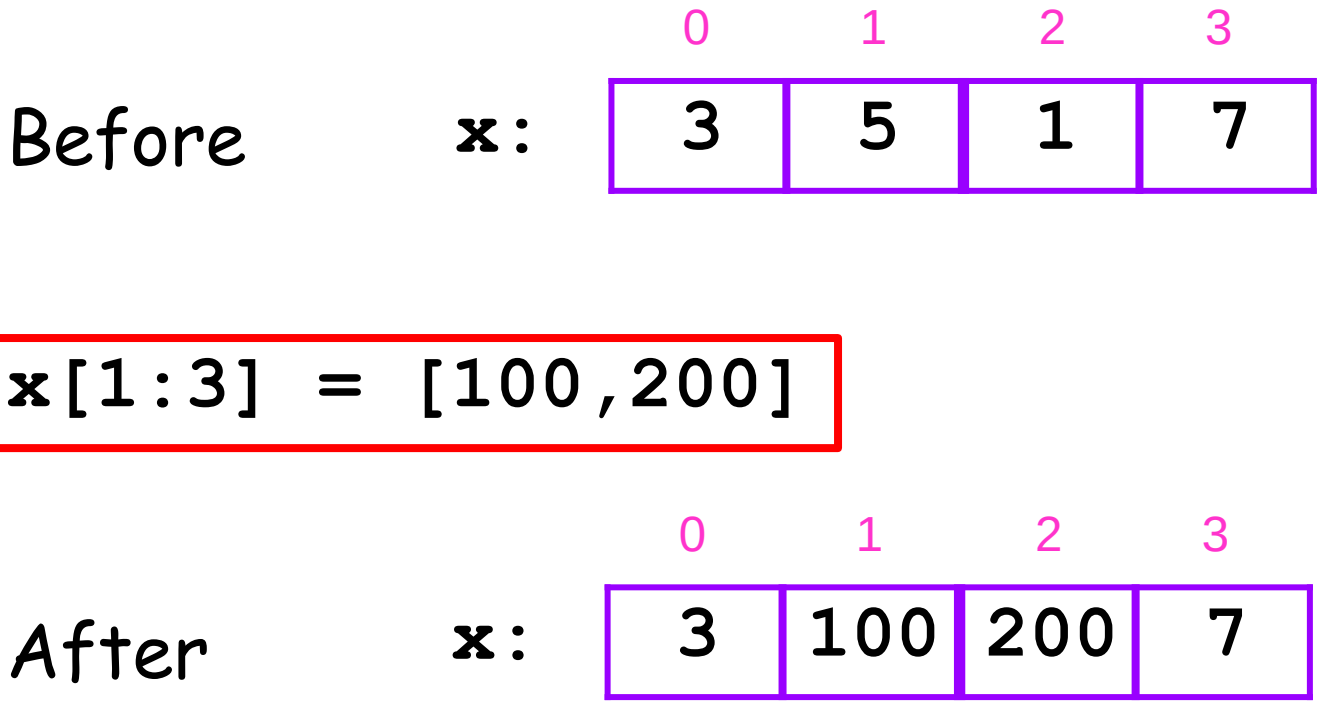

```
x[2] = 100
```

After:



You can change the values in a list

Lists ARE Mutable




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


You can change the values in a list

List Methods

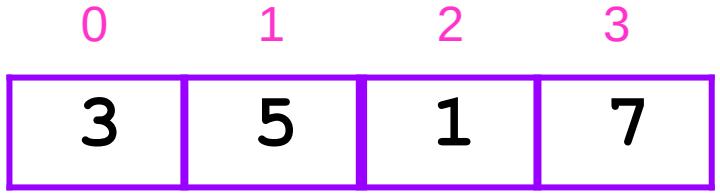
When these methods are applied to a list, they affect the list.

```
append
extend
insert
sort
```

Let's see what they do through examples…

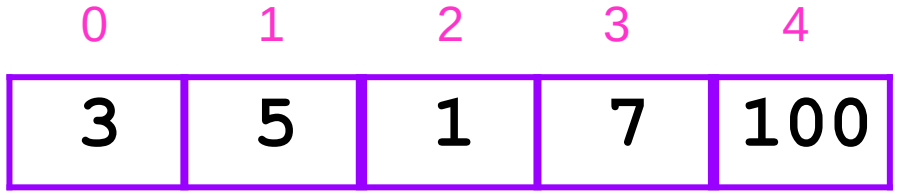
List Methods: append

Before:




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

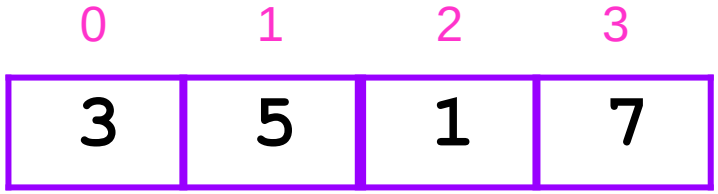
After:



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

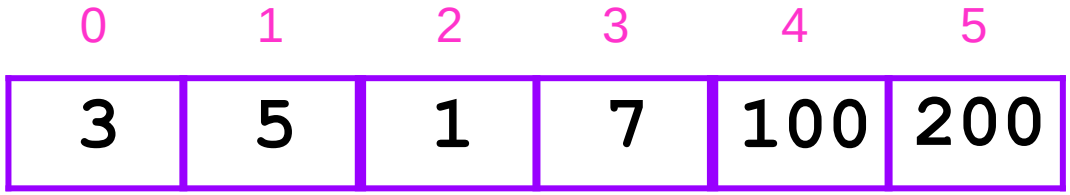
List Methods: extend

Before:




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

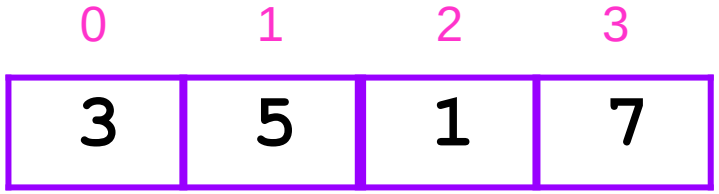
After:



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

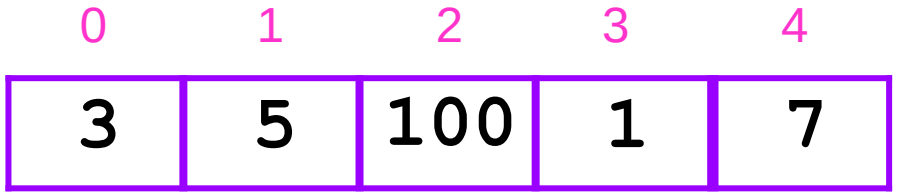
List Methods: `insert`

Before:




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

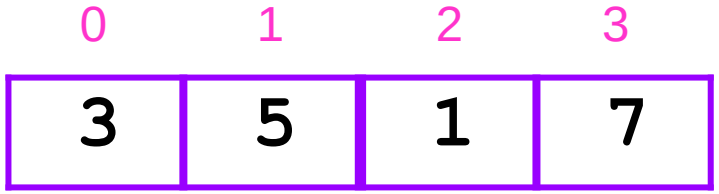
After:



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.

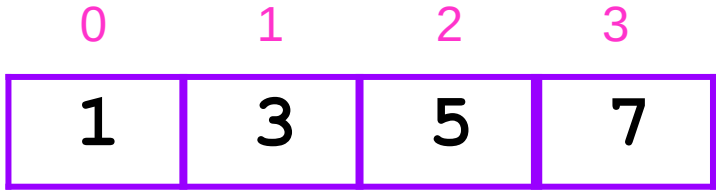
List Methods: sort

Before:




```
x.sort()
```

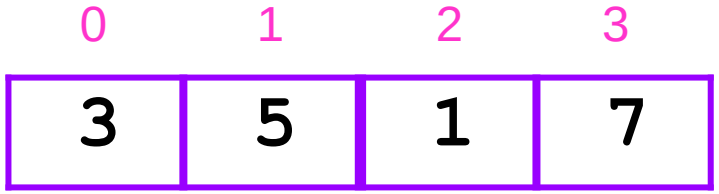
After:



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

List Methods: sort

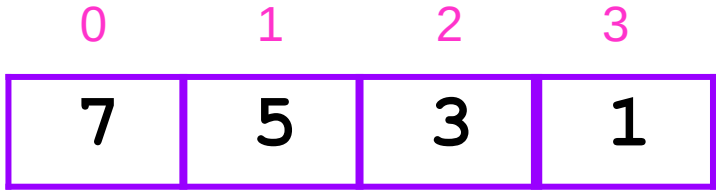
Before:




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

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

After:



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.

(Fruitful) List Methods

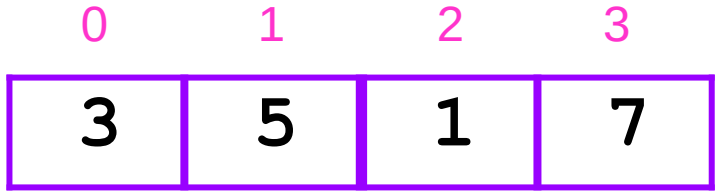
When these methods are applied to a list, they actually return something:

`pop`
`count`

Let's see what they do through examples…

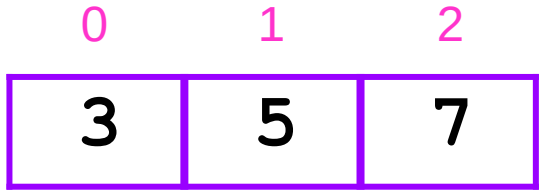
The List Method pop

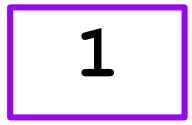
Before:




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

After:

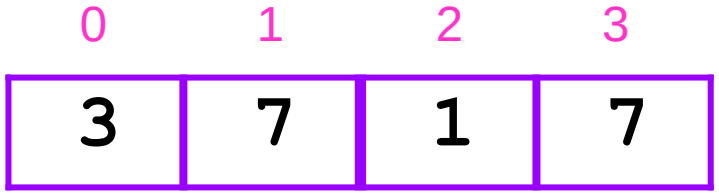




Use pop when you want to remove an element and assign it to a variable.

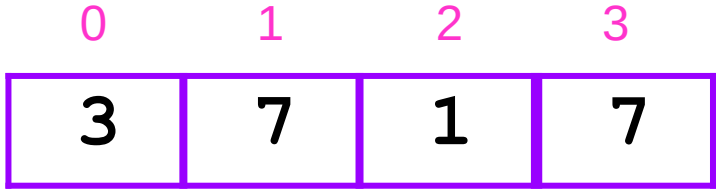
The List Method count

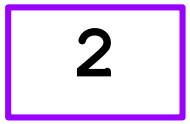
Before:




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

After:





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

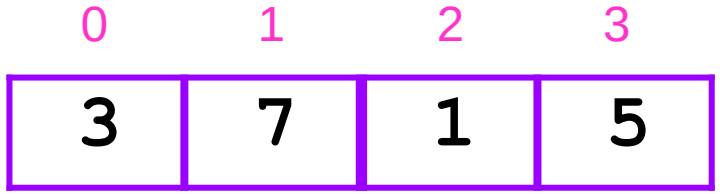
Two Built-In Functions that Can be Applied to Lists

`len` returns the length of a list

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

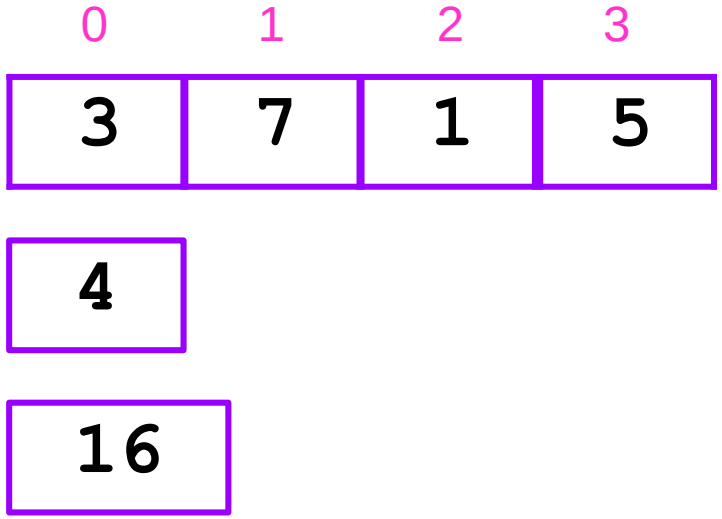
len and sum

Before




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

After



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 thecommas!

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.

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.