Python Guide

About Python

Python Interpreter

Command Line

Keyboard Shortcuts

Style1

Style Readable

Style Decomp

Variables

Math

Functions

Debugging

Doctests

For Loop

While Loop

If and Comparisons

Boolean and or not

Range

Strings

print() Standard Out

input()

File Read Write

Lists

main() Command Line Args

Dicts

Python No Copy / is

Tuples

Map Lambda

Comprehens

Sorting

Python Lists

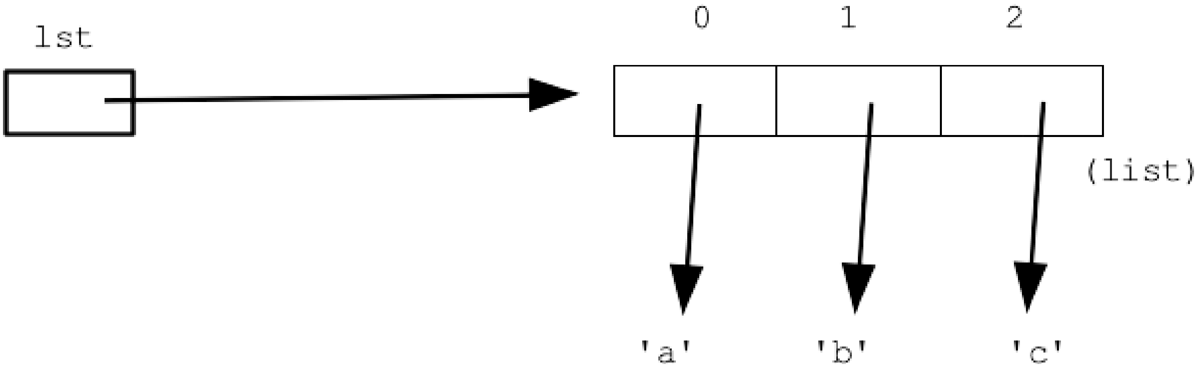
A list contains series of any data type: strings, ints, other lists. The things inside a list are generically called "elements". Unlike strings, lists are "mutable" - they can be changed.

Using the standard indexing scheme, the first element is at index 0, the next at index 1, and so on up to index length-1. As elements are added and removed, Python maintains the elements as contiguous block alwayx indexed by the numbers 0..length-1 inclusive.

Here is the code to create a list of the three strings 'a' 'b' and 'c'. The list is written within square brackets, and the elements are separated by commas.

```
>>> lst = ['a', 'b', 'c']
```

Here is what that list looks like in memory



Basic Operations

`lst = []` - create empty list

`lst = [1, 2, 3]` - create list with data in it. As a convenience, it's allowable to have an extra comma at the end of the series of elements like this: `[1, 2, 3,]`

`len(lst)` - access length of string

`lst[0]` - access individual elements with square brackets

`for x in lst:` - loop over contents, do not modify lst during loop

`x in lst` - boolean test if x is in lst (just like for string)

Python Guide

About Python

Python Interpreter

Command Line

Keyboard Shortcuts

Style1

Style Readable

Style Decomp

Variables

Math

Functions

Debugging

Doctests

For Loop

While Loop

If and Comparisons

Boolean and or not

Range

Strings

print() Standard Out

input()

File Read Write

Lists

main() Command Line Args

Dicts

Python No Copy / is

Tuples

Map Lambda

Comprehens

Sorting

`lst.append(x)` - add x to the end of lst, increasing its length by 1. The easiest way to add to a list. Does not return anything. Changes lst in place.

```
>>> lst = []             # Start with empty list
>>> lst.append('a')      # Append() elements
>>> lst.append('b')
>>> lst.append('c')
>>> lst
['a', 'b', 'c']
>>> len(lst)
3
>>> lst[0]
'a'
>>> lst[2]
'c'
>>> lst[3]
IndexError: list index out of range
>>>
>>> lst
['a', 'b', 'c']
>>> lst[0] = 'apple'     # Change data at index 0
>>>
>>> lst
['apple', 'b', 'c']
>>>
>>> 'b' in lst           # "in" check
True
```

List pop()

`lst.pop()` - remove the element from the end of the list and return it, decreasing the length of the list by 1. Mnemonic: the exact opposite of `append()`.

`lst.pop(`*`index`*`)` - alternate version with the index to remove is given, e.g. `lst.pop(0)` removes the element at index 0. Raises an error if the index is not valid.

```
>>> lst = ['a', 'b', 'c', 'd']
>>> lst.pop()    # default = remove from end
'd'
>>> lst
['a', 'b', 'c']
>>> lst.pop(0)   # can specify index to pop
'a'
>>> lst
['b', 'c']
```

List remove()

Python Guide

About Python

Python Interpreter

Command Line

Keyboard Shortcuts

Style1

Style Readable

Style Decomp

Variables

Math

Functions

Debugging

Doctests

For Loop

While Loop

If and Comparisons

Boolean and or not

Range

Strings

print() Standard Out

input()

File Read Write

Lists

main() Command Line Args

Dicts

Python No Copy / is

Tuples

Map Lambda

Comprehens

Sorting

`lst.remove(`*`elem`*`)` - search the list for the first instance of *elem* and remove it. It's an error to remove() an elem not in the list - could use `in` to check first. Note that pop() uses index numbers, but remove() uses the value, e.g. 'b', to search for and remove.

```
>>> lst = ['a', 'b', 'c', 'd']
>>> lst.remove('b')
>>> lst
['a', 'c', 'd']
>>> lst.remove('b')
ValueError: list.remove(x): x not in list
```

List extend()

`lst.extend(lst2)` - add all the elements of lst2 on to the end of lst.

```
>>> lst = [1, 2, 3]
>>> x = [4, 5]
>>> lst.extend(x)    # extend = add all
>>> lst
[1, 2, 3, 4, 5]
```

Append vs. Extend

Append vs. extend example:

```
>>> lst = [1, 2, 3]
>>> x = [4, 5]
>>> # what happens .append() vs. .extend() ?
>>>
>>> # 1. append:
>>> lst.append(x)
>>> # x is added as an *element* so lst is [1, 2, 3,
[4, 5]]
>>>
>>> # 2. extend:
>>> lst.extend(x)
>>> # all elements of x are added at end, so lst is [1,
2, 3, 4, 5]
```

List +

The + operation is an alternative to extend(), combining lists to make a bigger list (very analogous to + with strings)

```
>>> lst = [1, 2, 3]
>>> x = [4, 5]
>>> lst + x       # put lists together
[1, 2, 3, 4, 5]
>>> lst           # original is unchanged
[1, 2, 3]
```

Python Guide

About Python

Python Interpreter

Command Line

Keyboard Shortcuts

Style1

Style Readable

Style Decomp

Variables

Math

Functions

Debugging

Doctests

For Loop

While Loop

If and Comparisons

Boolean and or not

Range

Strings

print() Standard Out

input()

File Read Write

Lists

main() Command Line Args

Dicts

Python No Copy / is

Tuples

Map Lambda

Comprehens

Sorting

List index()

`lst.index(x)` - Look for first instance of x in lst and return its index. Raises an error if x is not in there - this is rather inconvenient. Therefore check with `in` first, and only if x is in there call index(). In other words, there is nothing as simple as str.find() for lists which IMHO seems like a real omission.

```
>>> lst = ['a', 'b', 'c']
>>> lst.index('c')
2
>>> lst.index('x')      # Error if not in
ValueError: 'x' is not in list
>>> 'x' in lst          # Therefore, check before
calling .index()
False
>>>
```

List min(), max()

`min(lst) max(lst)` - Return the smallest or largest element in lst. Uses the same underlying < foundation as sorted(), but much faster than sorting the whole list. Raises an error if the list is empty. Note that some functions, like these and len(), are regular functions, not noun.verb. That is because these functions work on many data types, not just lists.

```
>>> min([2, 5, 1, 6])
1
```

List insert(), copy()

`lst.insert(index, x)` - insert the element x so it is at the given index, shifting elements towards the end of the list as needed. Use index=len(lst) to insert at the end. Append() is simpler since it just goes on the end without any shifting and you don't have to think about index numbers.

`lst.copy()` - returns a copy of lst. You could use this to loop over a list and also modify it - loop over a copy, modify the original. (mentioned for completeness, I don't think we will ever need this function in CS106A.)

More details at official Python List Docs

List Slices

Slices work to pull out parts of list just as with strings.

```
lst = ['a', 'b', 'c']
lst[:2] -> ['a', 'b']
lst[2:] -> ['c']
```

Python Guide

About Python

Python Interpreter

Command Line

Keyboard Shortcuts

Style1

Style Readable

Style Decomp

Variables

Math

Functions

Debugging

Doctests

For Loop

While Loop

If and Comparisons

Boolean and or not

Range

Strings

print() Standard Out

input()

File Read Write

Lists

main() Command Line Args

Dicts

Python No Copy / is

Tuples

Map Lambda

Comprehens

Sorting

The original list is not modified, this creates a new list populated with elements from the original. Omitting both start and end indexes yields a copy of the whole list - `lst[:]`

Foreach loop - for elem in list

It's very easy to "foreach" loop over all the elements in a list, seeing each element once. Do not modify the list during iteration.

```
urls = ['https://....', ...]
for url in urls:
    # use url in here
    print(url)
```

Style: it's a nice pattern to name the list variable with the letter "s" like "urls". Then the loop variable can use the singular form "url" - confirming as you type that the loop variable and what's in the collection match up. Many Python bugs amount to mixing up what type of data is in a variable, so this convention can help you keep in mind what is in the collection.

Index loop - for i in range

The standard for/i/range loop works on lists too, using square brackets to access each element. Use this form if you want to know the index number each element during iteration.

```
lst = [...]
for i in range(len(lst)):
    # use lst[i]
```

Load a list with data

A common pattern to load up a list is to start with the empty list, and then in a loop of some sort, perhaps reading lines from a file, use .append() to load elements into the list.

```
lst = []
for i in range(10):
  lst.append(i)
# lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Del

The `del` feature is Python deletes items out of a list or dict, modifying the structure in place. For its syntax, basically write a square bracket expression to refer to an element, and `del` can delete that element. Like this:

```
>>> lst = ['a', 'b', 'c']
>>> lst[0]
```

Python Guide

About Python

Python Interpreter

Command Line

Keyboard Shortcuts

Style1

Style Readable

Style Decomp

Variables

Math

Functions

Debugging

Doctests

For Loop

While Loop

If and Comparisons

Boolean and or not

Range

Strings

print() Standard Out

input()

File Read Write

Lists

main() Command Line Args

Dicts

Python No Copy / is

Tuples

Map Lambda

Comprehens

Sorting

```
'a'
>>> del lst[0]  # delete the [0] element, lst is
modified
>>> lst
['b', 'c']
>>>
>>> # Elements shift over to stay in 0..len-1, so now
[0] is 'b'
>>> lst[0]
'b'
```

Python list elements are are kept in a contiguous block, with index numbers 0..len-1. Therefore, deleting an element from a list, Python will automatically shift over the elements to its right to keep the elements contiguous.

Del works with slices too, deleting a range deletes that sub-part of the list:

```
>>> lst = ['a', 'b', 'c', 'd']
>>> del lst[1:3]
>>> lst
['a', 'd']
```

Del works with dicts too.

Iterable

Many Python functions, such as range(), return an "iterable" which is list-like, but is not a list exactly. Fortunately, most Python features that work with lists, work with iterables too:

- Suppose we have *iterable*, all of these like-like forms work:
- `for elem in iterable:`
- `len(iterable)`
- `iterable[0]`
- `sorted(iterable)`

Look, for example, at the familiar loop to go through a series of numbers

```
>>> for i in range(10):
...   print(i)
...
0
1
2
3
4
5
6
```

Python Guide

About Python

Python Interpreter

Command Line

Keyboard Shortcuts

Style1

Style Readable

Style Decomp

Variables

Math

Functions

Debugging

Doctests

For Loop

While Loop

If and Comparisons

Boolean and or not

Range

Strings

print() Standard Out

input()

File Read Write

Lists

main() Command Line Args

Dicts

Python No Copy / is

Tuples

Map Lambda

Comprehens

Sorting

```
7
8
9
```

How does that for loop work? The call to `range(10)` is not returning a list. It returns an iterable representing the series of numbers, and fortunately the for-loop works fine with an iterable.

However, list-specific functions like .append() do not work on iterables. If you have an iterable and need a list, it's easy to construct a list from the iterable like this:

```
>>> lst = list(range(10))
>>> lst.append(99)
```

Why do Python functions return an iterable instead of a full list? Because the iterable is more lightweight and efficient compared to a list. In particular, the iterable does not allocate memory for all its elements the way a list does. Therefore, it's generally a little more efficient to do a computation with an iterable.

Behind the scenes: how does the iterable work? The Python iterator strategy uses a special function, __next__(). Each call to __next__() returns the next element of the sequence. Your code does not need to call the __next__() function explicitly. Behind the scenes, the for-loop calls __next__() again and again to get all the elements needed for the loop.