
Lists



Lists

- A list is an ordered collection of objects

```
>>> linguists = ["Admanda", "Claire", "Holly", "Luis",  
"Nick", "Sophia"]
```

- Variable length, heterogeneous, and arbitrarily nestable

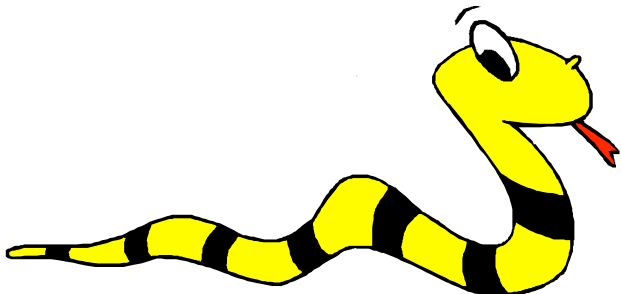
```
>>> shortlist = ["a"]
```

```
>>> mixedlist = ["a", "1", "b", 2, "c", 3]
```

```
>>> listinlist = [["a", "b", "c"], [1, 2, 3]]
```

- Lists are mutable (unlike strings)

Tuples, Lists, and Strings: Similarities



Similar Syntax

- Tuples and lists are sequential containers that share much of the same syntax and functionality.
 - For conciseness, they will be introduced together.
 - The operations shown in this section can be applied to both tuples and lists, but most examples will just show the operation performed on one or the other.
- While strings aren't exactly a container data type, they also happen to share a lot of their syntax with lists and tuples; so, the operations you see in this section can apply to them as well.

Tuples, Lists, and Strings 1

- Tuples are defined using parentheses (and commas).

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Lists are defined using square brackets (and commas).

```
>>> li = ["abc", 34, 4.34, 23]
```

- Strings are defined using quotes (" , ' , or """).

```
>>> st = "Hello World"
```

```
>>> st = 'Hello World'
```

```
>>> st = """This is a multi-line  
string that uses triple quotes."""
```

Tuples, Lists, and Strings 2

- We can access individual members of a tuple, list, or string using square bracket "array" notation.

```
>>> tu[1]          # Second item in the tuple.
```

```
'abc'
```

```
>>> li[1]          # Second item in the list.
```

```
34
```

```
>>> st[1]         # Second character in string.
```

```
'e'
```

Looking up an Item

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Positive index: count from the left, starting with 0.

```
>>> t[1]
'abc'
```

Negative lookup: count from right, starting with -1.

```
>>> t[-3]
4.56
```

Slicing: Return Copy of a Subset 1

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Return a copy of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> t[1:4]
('abc', 4.56, (2,3))
```

You can also use negative indices when slicing.

```
>>> t[1:-1]
('abc', 4.56, (2,3))
```

Slicing: Return Copy of a Subset 2

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
```

Omit the first index to make a copy starting from the beginning of the container.

```
>>> t[:2]
(23, 'abc')
```

Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> t[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Container

You can make a copy of the whole tuple using
[:].

```
>>> t[:]  
(23, 'abc', 4.56, (2,3), 'def')
```

So, there's a difference between these two
lines:

```
>>> list2 = list1      # 2 names refer to 1 ref  
                        # Changing one affects both
```

```
>>> list2 = list1[:]   # Two copies, two refs  
                        # They're independent
```

The 'in' Operator

- Boolean test whether a value is inside a container:

```
>>> t = [1, 2, 4, 5]
```

```
>>> 3 in t
```

```
False
```

```
>>> 4 in t
```

```
True
```

```
>>> 4 not in t
```

```
False
```

- Be careful: the 'in' keyword is also used in the syntax of other unrelated Python constructions: "for loops" and "list comprehensions."

The + Operator

- The + operator produces a new tuple, list, or string whose value is the concatenation of its arguments.

```
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
```

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

```
>>> "Hello" + " " + "World"
'Hello World'
```

The * Operator

- The * operator produces a new tuple, list, or string that "repeats" the original content.

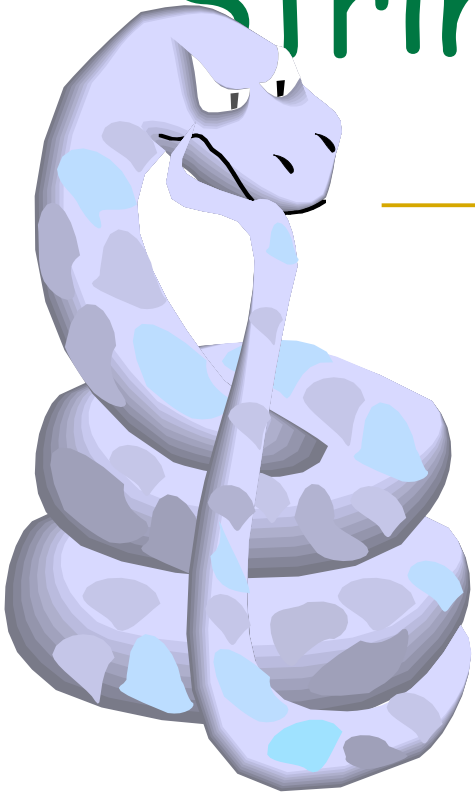
```
>>> (1, 2, 3) * 3  
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```
>>> [1, 2, 3] * 3  
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> "Hello" * 3  
'HelloHelloHello'
```

Mutability:

Strings, Tuples vs. Lists



Strings: Immutable

```
>>> str = "spam"  
>>> str[1] = 'l'
```

module

```
Traceback (most recent call last):
```

```
  File "<pyshell#75>", line 1, in -toplevel-  
    tu[2] = 3.14
```

```
TypeError: object doesn't support item assignment
```

```
>>> str.replace('p', 'l')  
>>> print str
```

If you really want to change its value, you have to make a copy:

```
>>> newstr = str.replace('p', 'l')  
>>> print newstr
```

Tuples: Immutable

```
>>> t = (23, 'abc', 4.56, (2,3), 'def')
>>> t[2] = 3.14
```

Traceback (most recent call last):

```
File "<pyshell#75>", line 1, in -toplevel-
    tu[2] = 3.14
```

TypeError: object doesn't support item assignment

You're not allowed to change a tuple *in place* in memory; so, you can't just change one element of it.

But it's always OK to make a fresh tuple and assign its reference to a previously used name.

```
>>> t = (1, 2, 3, 4, 5)
```

Lists: Mutable

```
>>> li = ['abc', 23, 4.34, 23]
>>> li2 = li
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

Has the value of li2 changed?

We can change lists *in place*. So, it's ok to change just one element of a list. Name li still points to the same memory reference when we're done.

Slicing: with mutable lists

- `>>> L = ['spam', 'Spam', 'SPAM']`

- `>>> L[1] = 'eggs'`

- `>>> L`

- `['spam', 'eggs', 'SPAM']`

- `>>> L[0:2] = ['eat', 'more']`

- `>>> L`

- `['eat', 'more', 'SPAM']`

Operations on Lists Only 1

- Since lists are mutable (they can be changed in place in memory), there are many more operations we can perform on lists than on tuples.
- The mutability of lists also makes managing them in memory more complicated... So, they aren't as fast as tuples. It's a tradeoff.

Operations on Lists Only 2

```
>>> li = [1, 2, 3, 4, 5]
```

```
>>> li.append('a')
```

```
>>> li
```

```
[1, 2, 3, 4, 5, 'a']
```

```
>>> li.insert(2, 'i')
```

```
>>> li
```

```
[1, 2, 'i', 3, 4, 5, 'a']
```

NOTE: li = li.insert(2, 'I'), what happens?

Operations on Lists Only 3

The 'extend' operation is similar to concatenation with the + operator. But while the + creates a fresh list (with a new memory reference) containing copies of the members from the two inputs, the extend operates on list `li` in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]  li + 12
```

Extend takes a list as an argument. Append takes a singleton.

```
>>> li.append([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [9, 8, 7]]
```

Operations on Lists Only 4

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

```
>>> li.index('b')      # index of first occurrence  
1
```

```
>>> li.count('b')     # number of occurrences  
2
```

```
>>> li.remove('b')    # remove first occurrence  
>>> li  
['a', 'c', 'b']
```

Operations on Lists Only 5

```
>>> li = [5, 2, 6, 8]
```

```
>>> li.reverse()      # reverse the list *in place*
```

```
>>> li
[8, 6, 2, 5]
```

```
>>> li.sort()        # sort the list *in place*
```

```
>>> li
[2, 5, 6, 8]
```

```
>>> li.sort(some_function)
# sort in place using user-defined comparison
```

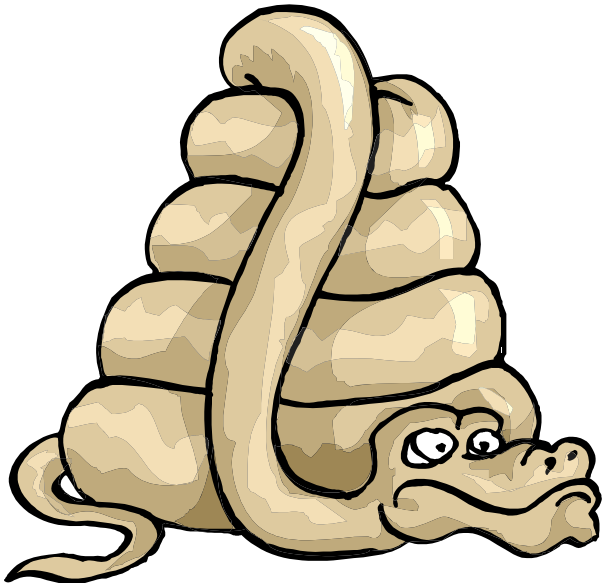
Tuples vs. Lists

- Lists slower but more powerful than tuples.
 - Lists can be modified, and they have lots of handy operations we can perform on them.
 - Tuples are immutable and have fewer features.
- We can always convert between tuples and lists using the `list()` and `tuple()` functions.

```
li = list(tu)
```

```
tu = tuple(li)
```

String Conversions



String to List to String

- Join turns a list of strings into one string.

```
<separator_string>.join( <some_list> )
```

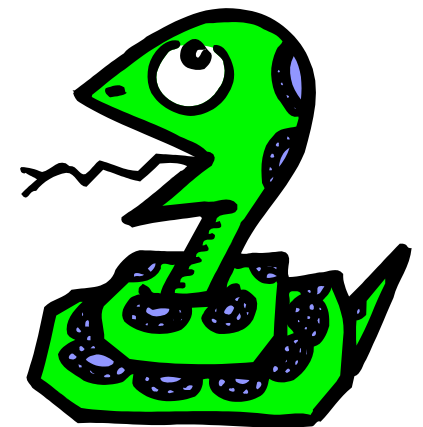
```
>>> ";" .join( ["abc", "def", "ghi"] )  
"abc;def;ghi"
```

- Split turns one string into a list of strings.

```
<some_string>.split( <separator_string> )
```

```
>>> "abc;def;ghi".split( ";" )  
["abc", "def", "ghi"]  
>>> "I love New York".split()  
["I", "love", "New", "York"]
```

For Loops



Motivating problem

- Given the following list

- `>>> linguists = ["Amanda", "Claire", "Holly", "Luis", "Nick", "Sophia"]`

- How do I print each name on a separate line?

- `>>> print linguists[0] + '\n'`
 - `>>> print linguists[1] + '\n'`
 - `>>> print linguists[2] + '\n'`
 - `>>> print linguists[3] + '\n'`
 - `>>> print linguists[4] + '\n'`
 - `>>> print linguists[5] + '\n'`

For Loops 1

- A for-loop steps through each of the items in a list, tuple, string, or any other type of object which the language considers an "iterator."

```
for <item> in <collection>:  
    <statements>
```

- When <collection> is a list or a tuple, then the loop steps through each element of the container.
- When <collection> is a string, then the loop steps through each character of the string.

```
for someChar in "Hello World":  
    print someChar
```

For Loops 2

- The `<item>` part of the for loop can also be more complex than a single variable name.
 - When the elements of a container `<collection>` are also containers, then the `<item>` part of the for loop can match the structure of the elements.
 - This multiple assignment can make it easier to access the individual parts of each element.

```
for (x, y) in [('a',1), ('b',2), ('c',3), ('d',4)]:  
    print x
```

Solution to our problem

- ```
>>> linguists = ["Amanda", "Claire", "Holly", "Luis",
"Nick", "Sophia"]
```
- ```
>>> for linguist in linguists:  
    print linguist
```

Exercise

- How do we take the sentence "Python is a great text processing language" and print one word on each line?