Lecture 11: Tuples and Lists

Comp 102

Fakhir Shaheen

Forman Christian University

Tuples

Tuples: A new data type

- Have seen scalar types: `int`,`float`,`bool`
- Have seen one compound type: `string`

Tuples: A new data type

- Have seen scalar types: `int`,`float`,`bool`
- Have seen one compound type: `string`
- Want to introduce more general **compound data types**
  - indexed sequences of elements, which could themselves be **compound structures**

Tuples: A new data type

- Have seen scalar types: `int`,`float`,`bool`
- Have seen one compound type: `string`
- Want to introduce more general **compound data types**
  - indexed sequences of elements, which could themselves be **compound structures**
  - **Tuples** - immutable
  - **Lists** - mutable

Tuples

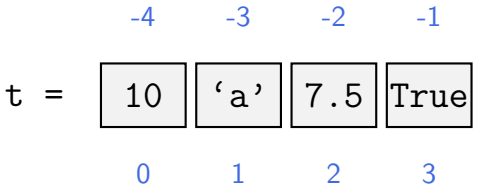
- **Indexable ordered sequence** of objects
  - Objects can be **any type** - int, string, tuple, tuple of tuples, ⋯

Tuples

- **Indexable ordered sequence** of objects
  - Objects can be **any type** - int, string, tuple, tuple of tuples, ⋯
- Cannot change element values, **immutable**

Tuple Structure & Indexing

```
# Creating a tuple
t = (10, 'a', 7.5, True)
len(t)       → 4
t[0]         → 10
t[2]         → 7.5
t[-1]        → True
t[0] = 5     → ERROR!
```



Returning Multiple Values

```
1 def get_stats(numbers):
2     total = sum(numbers)
3     count = len(numbers)
4     average = total / count
5     return (total, count, average)
6
7 # Using the function
8 result = get_stats([10, 20, 30])
```

Returning Multiple Values

```
1  def get_stats(numbers):
2      total = sum(numbers)
3      count = len(numbers)
4      average = total / count
5      return (total, count, average)
6
7  # Using the function
8  result = get_stats([10, 20, 30])
9  print(result)  → (60, 3, 20.0)
10 # Unpacking the tuple
11 s, c, a = get_stats([10, 20, 30])
12 print(s)        → 60
13 print(c)        → 3
14 print(a)        → 20.0
```

Big Idea

Returning **one** **object** *(a tuple)* allows you to return multiple **values** *(tuple elements)*

You Try!

Write a function that meets these specs:

```
1 def char_counts(s):
2     """
3     - s is a string of lowercase chars
4     - Return a tuple where the first element is the
        number of vowels in s and the second element is
        the number of consonants in s
5     """
6     pass
```

You Try! — Solution

```
1  def char_counts(s):
2      vowels = 0
3      consonants = 0
4      for ch in s:
5          if ch in 'aeiou':
6              vowels += 1
7          else:
8              consonants += 1
9      return (vowels, consonants)
10
11 print(char_counts('forman'))    → (2, 4)
12 v, c = char_counts('forman')    → v is 2, c is 4
```

You Try! — Solution

```
1  def char_counts(s):
2      vowels = 0
3      consonants = 0
4      for ch in s:
5          if ch in 'aeiou':
6              vowels += 1
7          else:
8              consonants += 1
9      return (vowels, consonants)
10
11 print(char_counts('forman'))    → (2, 4)
12 v, c = char_counts('forman')    → v is 2, c is 4
```

✓ two counters, one loop: every character adds 1 to exactly one of them

✓ `return` gives back *one* object, a tuple, that carries both counts

Lists

Lists

- **Indexable ordered sequence** of objects
  - Usually homogeneous *(i.e., all integers, all strings, all lists)*
  - But can contain mixed types *(not common)*

Lists

- **Indexable ordered sequence** of objects
  - Usually homogeneous *(i.e., all integers, all strings, all lists)*
  - But can contain mixed types *(not common)*
- Denoted by **square brackets**: [ ] Tuples are ()

Lists

- **Indexable ordered sequence** of objects
  - Usually homogeneous *(i.e., all integers, all strings, all lists)*
  - But can contain mixed types *(not common)*
- Denoted by **square brackets**: [ ] Tuples are ()
- **Mutable**, this means you can change values of specific elements of list *(Tuples are immutable)*

Tuples vs Lists

Tuples

Syntax: `( )`

Lists

Syntax: `[ ]`

Tuples vs Lists

Tuples

Syntax: ( )

```
t = (1, 2, 3)
```

Lists

Syntax: [ ]

```
L = [1, 2, 3]
```

Tuples vs Lists

Tuples

Syntax: ( )

```
t = (1, 2, 3)
```

✓ **Immutable**
✗ Cannot change
✓ Faster
✓ Less memory

Lists

Syntax: [ ]

```
L = [1, 2, 3]
```

✓ **Mutable**
✓ Can change
✓ More flexible
✓ Dynamic size

Tuples vs Lists

Tuples

Syntax: `( )`

```
t = (1, 2, 3)
```

✓ **Immutable**
✗ Cannot change
✓ Faster
✓ Less memory

```
t[0] = 5
```

ERROR!

Lists

Syntax: `[ ]`

```
L = [1, 2, 3]
```

✓ **Mutable**
✓ Can change
✓ More flexible
✓ Dynamic size

```
L[0] = 5
```

Works! `L = [5, 2, 3]`

Tuples vs Lists

Tuples

Syntax: ( )

```
t = (1, 2, 3)
```

✓ **Immutable**
✗ Cannot change
✓ Faster
✓ Less memory

```
t[0] = 5
```

ERROR!

Choose based on mutability needs

Lists

Syntax: [ ]

```
L = [1, 2, 3]
```

✓ **Mutable**
✓ Can change
✓ More flexible
✓ Dynamic size

```
L[0] = 5
```

Works! `L = [5, 2, 3]`

You Try!

Write a function that meets these specs:

```
1  def sum_and_prod(L):
2      """
3      - L is a list of numbers
4      - Return a tuple where the first value is the sum of
          all elements in L and the second value is the product
           of all elements in L
5      """
6      pass
```

For example:

```
sum_and_prod([2, 3, 4])    → (9, 24)
sum_and_prod([10, 5])      → (15, 50)
```

You Try! — Solution

```
1  def sum_and_prod(L):
2      total = 0
3      prod = 1
4      for e in L:
5          total += e
6          prod *= e
7      return (total, prod)
8
9  print(sum_and_prod([2, 3, 4]))    → (9, 24)
10 s, p = sum_and_prod([2, 3, 4])    → s is 9, p is 24
```

You Try! — Solution

```
1  def sum_and_prod(L):
2      total = 0
3      prod = 1
4      for e in L:
5          total += e
6          prod *= e
7      return (total, prod)
8
9  print(sum_and_prod([2, 3, 4]))    → (9, 24)
10 s, p = sum_and_prod([2, 3, 4])    → s is 9, p is 24
```

✓ `for e in L` visits every element of the list, as it visits every character of a string

✓ the sum starts at 0, the product at 1: a product that starts at 0 stays 0

✓ one loop builds both answers, and `return` hands them back as one tuple

Indices and Ordering in Lists

Indices and Ordering in Lists

```
1 a_list = []                    # empty list
```

Indices and Ordering in Lists

```
1 a_list = []                  # empty list
2 L = [2, 'a', 4, [1,2]]
```

Indices and Ordering in Lists

```
1 a_list = []                  # empty list
2 L = [2, 'a', 4, [1,2]]
3 len(L)                       # evaluates to 4
```

Indices and Ordering in Lists

```
1 a_list = []                  # empty list
2 L = [2, 'a', 4, [1,2]]
3 len(L)                       # evaluates to 4
4 L[0]                         # evaluates to 2
```

Indices and Ordering in Lists

```
1 a_list = []                  # empty list
2 L = [2, 'a', 4, [1,2]]
3 len(L)                       # evaluates to 4
4 L[0]                         # evaluates to 2
5 L[3]                         # evaluates to [1,2], another list
```

Indices and Ordering in Lists

```
1 a_list = []                  # empty list
2 L = [2, 'a', 4, [1,2]]
3 len(L)                       # evaluates to 4
4 L[0]                         # evaluates to 2
5 L[3]                         # evaluates to [1,2], another list
6 [2,'a'] + [5,6]              # evaluates to [2,'a',5,6]
```

Indices and Ordering in Lists

```
1 a_list = []                  # empty list
2 L = [2, 'a', 4, [1,2]]
3 len(L)                       # evaluates to 4
4 L[0]                         # evaluates to 2
5 L[3]                         # evaluates to [1,2], another list
6 [2,'a'] + [5,6]              # evaluates to [2,'a',5,6]
7 max([3,5,0])                 # evaluates to 5
```

Indices and Ordering in Lists

```
a_list = []                 # empty list
L = [2, 'a', 4, [1,2]]
len(L)                      # evaluates to 4
L[0]                        # evaluates to 2
L[3]                        # evaluates to [1,2], another list
[2,'a'] + [5,6]             # evaluates to [2,'a',5,6]
max([3,5,0])                # evaluates to 5
L[1:3]                      # evaluates to ['a', 4]
```

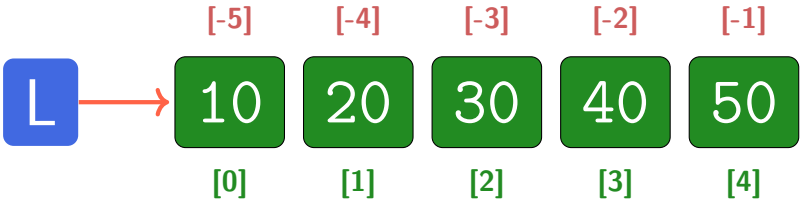
Indices and Ordering in Lists

```
a_list = []                    # empty list
L = [2, 'a', 4, [1,2]]
len(L)                         # evaluates to 4
L[0]                           # evaluates to 2
L[3]                           # evaluates to [1,2], another list
[2,'a'] + [5,6]                # evaluates to [2,'a',5,6]
max([3,5,0])                   # evaluates to 5
L[1:3]                         # evaluates to ['a', 4]
for e in L:          # loop variable becomes each element in L
    print(e)
```

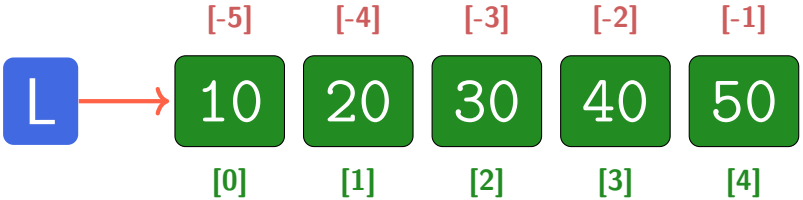
Indices and Ordering in Lists

```
1  a_list = []                    # empty list
2  L = [2, 'a', 4, [1,2]]
3  len(L)                         # evaluates to 4
4  L[0]                           # evaluates to 2
5  L[3]                           # evaluates to [1,2], another list
6  [2,'a'] + [5,6]                # evaluates to [2,'a',5,6]
7  max([3,5,0])                   # evaluates to 5
8  L[1:3]                         # evaluates to ['a', 4]
9  for e in L:          # loop variable becomes each element in L
10     print(e)
11 L[3] = 10                      # mutates L to [2,'a',4,10]
```

List Indexing and Slicing



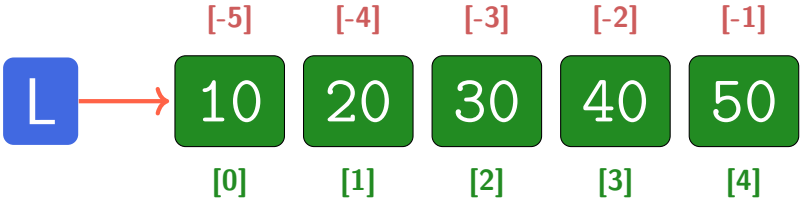
List Indexing and Slicing



Positive indexing:

```
L[0] → 10
L[2] → 30
L[4] → 50
```

List Indexing and Slicing



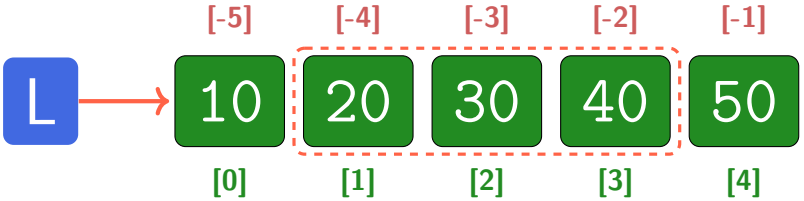
Positive indexing:

```
L[0] → 10
L[2] → 30
L[4] → 50
```

Negative indexing:

```
L[-1] → 50
L[-3] → 30
L[-5] → 10
```

List Indexing and Slicing



Slicing: `L[1:4]`

Positive indexing:

`L[0]` → 10
`L[2]` → 30
`L[4]` → 50

Negative indexing:

`L[-1]` → 50
`L[-3]` → 30
`L[-5]` → 10

`L[1:4]` → `[20, 30, 40]` *(returns a new list)*

Big Idea

Some functions **mutate** the list and don't return anything.

These are the functions that have something called **side effect**.

Operations on Lists: append()

```
L = [2, 1, 3]
```



Operations on Lists: append()

```
L = [2, 1, 3]
L.append(5)
```



Operations on Lists: append()

```
L = [2, 1, 3]
L.append(5)
```

New element



List mutated! New element added AT THE END.

Operations on Lists: append()

```
L = [2, 1, 3]
L.append(5)
```



List mutated! New element added AT THE END.

- What is the dot?
  - Lists are Python objects, **everything** in Python is an **object**
  - Objects have **data**, **operations**
  - Access this information by `object_name.do_something()`

You Try!

What is the value of L1, L2, L3 and L at the end?

```
1 L1 = ['re']
2 L2 = ['mi']
3 L3 = ['do']
4 L4 = L1 + L2
5 L3.append(L4)
6 L = L1.append(L3)
```

You Try!

Write a function that meets these specs:

```
1 def make_ordered_list(n):
2     """
3     - n is a positive int
4     - Returns a LIST containing all ints in
        order from 0 to n (inclusive)
5     """
6     pass
```

Solution (1)

```
1 def make_ordered_list(n):
2     L = []
3     for i in range(n + 1):
4         L.append(i)
5     return L
6
7 L = make_ordered_list(5)
8 print(L)     # prints [0, 1, 2, 3, 4, 5]
```

Solution (2)

```
1 def make_ordered_list(n):
2     """
3     - n is a positive int
4     - Returns a LIST containing all ints in
        order from 0 to n (inclusive)
5     """
6     return list(range(n + 1))
```

You Try!

Write a function that meets these specs:

```
1 def remove_elem(L, e):
2     """
3     - L is a list
4     - Returns a NEW list with elements in the same
        order as L but without any elements equal to e.
5     """
6     pass
7
8 L = [1,2,2,2]
9 print(remove_elem(L, 2))      # prints [1]
```

Solution

```
1 def remove_elem(L, e):
2     new_list = []
3     for item in L:
4         if item != e:
5             new_list.append(item)
6     return new_list
7
8 L = [1 ,2 ,2 ,2, 3, 2]
9 print(remove_elem(L, 2))      # prints [1, 3]
```

Strings to Lists

- Convert **string to list** with `list(s)`
  - Every character from `s` is an element in a list

Strings to Lists

- Convert **string to list** with `list(s)`
  - Every character from `s` is an element in a list
- Use `s.split()`, to split a string on a character parameter, splits on spaces if called without a parameter

Strings to Lists

- Convert **string to list** with `list(s)`
  - Every character from `s` is an element in a list
- Use `s.split()`, to split a string on a character parameter, splits on spaces if called without a parameter
- `s.split(char)`, splits on `char` if given a parameter
  e.g: `s.split(',')`, splits on commas

String to List: `split()`

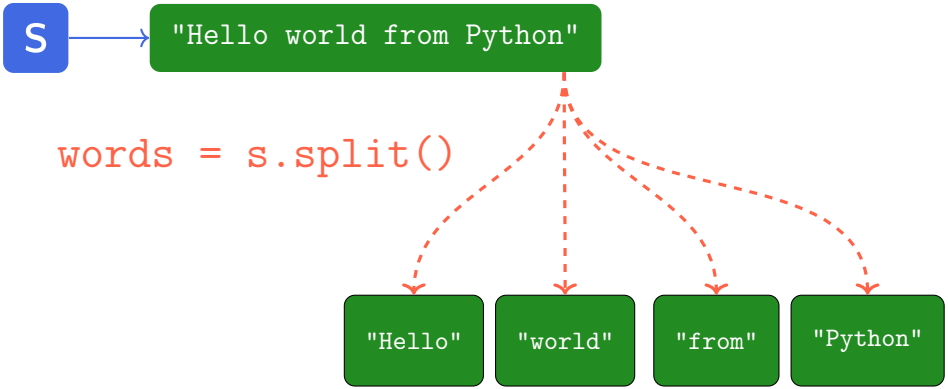


String to List: `split()`



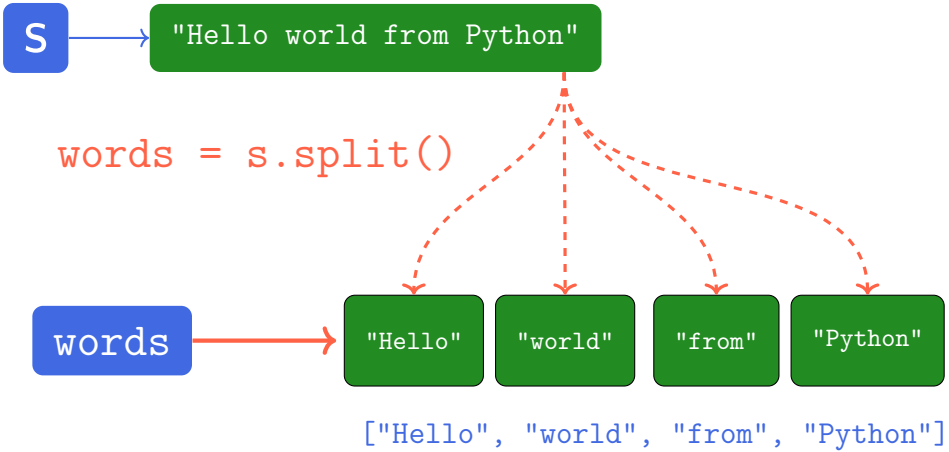

```
words = s.split()
```

String to List: split()



Splitting on spaces...

String to List: split()



Lists to Strings

- Convert **a list of strings back to string**

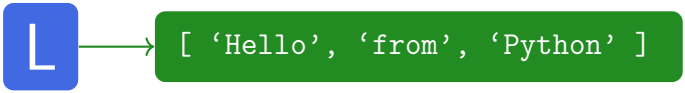
Lists to Strings

- Convert **a list of strings back to string**
- Use `''.join(L)`, to turn a list of strings into a bigger string

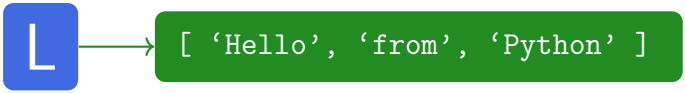
Lists to Strings

- Convert **a list of strings back to string**
- Use `''.join(L)`, to turn a list of strings into a bigger string
- Can give a character in quotes to add char between every element

List to String: `join()`



List to String: `join()`




```
result = ' '.join(L)
```

Joining with **space** character

List to String: join()




```
result = ' '.join(L)
```

Joining with **space** character



List to String: `join()`




```
result = ' '.join(L)
```

Joining with **space** character



Single string created from list of elements!

You Try!

Write a function that meets these specs:

```
1 def count_words(sen):
2     '''
3     - sen is a string representing a sentence
4     - Returns how many words are in s (i.e. a word is a
          sequence of characters between spaces).
5     '''
6     pass
7
8 print(count_words("Hello it's me"))
```

Solution

```
1 def count_words(sen):
2     words = sen.split()
3     return len(words)
4
5 print(count_words("Hi it's me"))  # prints 3
```

Other List Operations

Add an element to end of list with `L.append(element)`

mutates the list

Other List Operations

Add an element to end of list with `L.append(element)`

mutates the list

`sort()`

```
L = [4,2,7]
L.sort()
```

mutates L

Other List Operations

Add an element to end of list with `L.append(element)`
mutates the list

`sort()`

```
L = [4,2,7]
L.sort()
```

mutates L

`reverse()`

```
L = [4,2,7]
L.reverse()
```

mutates L

Other List Operations

Add an element to end of list with `L.append(element)`

mutates the list

`sort()`

```
L = [4,2,7]
L.sort()
```

mutates L

`reverse()`

```
L = [4,2,7]
L.reverse()
```

mutates L

`sorted()`

```
L = [4,2,7]
L_new = sorted(L)
```

Returns a sorted version of L (**no mutation!**)

You Try!

Write a function that meets these specs:

```
def sort_words(sen):
    '''
    - sen is a string representing a sentence
    - Returns a NEW list containing all the words in sen
       but sorted in alphabetical order.
    '''
    pass

print(sort_words("look at this photograph"))
```

Solution

```
1 def sort_words(sen):
2     words = sen.split()
3     return sorted(words)
4
5 print(sort_words("look at this photograph"))
6 # output: ['at', 'look', 'photograph', 'this']
```

Big Idea

Functions with side effects mutate inputs.

You can write your own!

List Mutation in Functions

Lists are passed by reference - mutations affect the original!

```
1 def double_elements(lst):
2     for i in range(len(lst)):
3         lst[i] = lst[i] * 2
4
5 numbers = [1, 2, 3]
6 double_elements(numbers)
7
8 print(numbers)  # [2,4,6]
```

List Mutation in Functions

Lists are passed by reference - mutations affect the original!

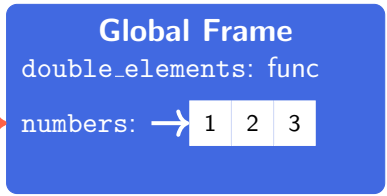
```
1  def double_elements(lst):
2      for i in range(len(lst)):
3          lst[i] = lst[i] * 2
4
5  numbers = [1, 2, 3]
6  double_elements(numbers)
7
8  print(numbers)  # [2,4,6]
```



List Mutation in Functions

Lists are passed by reference - mutations affect the original!

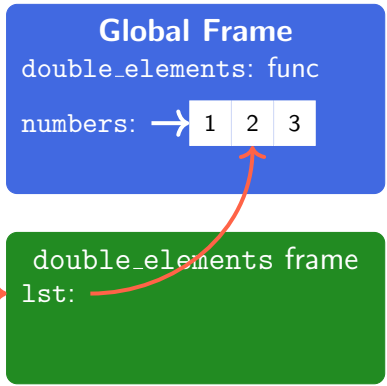
```
1  def double_elements(lst):
2      for i in range(len(lst)):
3          lst[i] = lst[i] * 2
4
5  numbers = [1, 2, 3]
6  double_elements(numbers)
7
8  print(numbers)  # [2,4,6]
```



List Mutation in Functions

Lists are passed by reference - mutations affect the original!

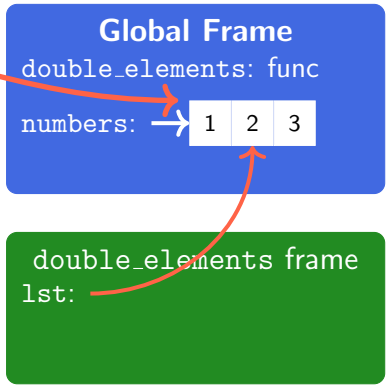
```
1  def double_elements(lst):
2      for i in range(len(lst)):
3          lst[i] = lst[i] * 2
4
5  numbers = [1, 2, 3]
6  double_elements(numbers)
7
8  print(numbers)  # [2,4,6]
```



List Mutation in Functions

Lists are passed by reference - mutations affect the original!

```
1  def double_elements(lst):
2      for i in range(len(lst)):
3          lst[i] = lst[i] * 2
4
5  numbers = [1, 2, 3]
6  double_elements(numbers)
7
8  print(numbers)  # [2,4,6]
```



List Mutation in Functions

Lists are passed by reference - mutations affect the original!

```
1  def double_elements(lst):
2      for i in range(len(lst)):
3          lst[i] = lst[i] * 2
4
5  numbers = [1, 2, 3]
6  double_elements(numbers)
7
8  print(numbers)  # [2,4,6]
```



You Try!

Write a function that meets these specs:

```
1  def square_list(L):
2      '''
3      - L is a list of numbers
4          - Mutates L by squaring each element
5      - Returns None
6      '''
7      pass
8
9  L = [1,2,3,4]
10 square_list(L)
11 print(L)     # prints [1,4,9,16]
```

You Try! Solution

Write a function that meets these specs:

```
1 def square_list(L):
2     for i in range(len(L)):
3         L[i] = L[i] ** 2
4
5
6 L = [1,2,3,4]
7 square_list(L)
8 print(L)     # prints [1,4,9,16]
```

Tricky Example 1:

- A loop iterates over **indices of L** and **mutates L** each time *(adds more elements)*

Tricky Example 1:

- A loop iterates over **indices of L** and **mutates L** each time *(adds more elements)*
- **Range returns something that behaves like a tuple** *(but isn't - it returns an iterable)*
  - Returns the first element, and a function to get the next element

Tricky Example 1:

- A loop iterates over **indices of L** and **mutates L** each time *(adds more elements)*
- **Range returns something that behaves like a tuple** *(but isn't - it returns an iterable)*
  - Returns the first element, and a function to get the next element

```
range(4)       # kind of like tuple (0,1,2,3)
range(2,9,2)   # kind of like tuple (2,4,6,8)
```

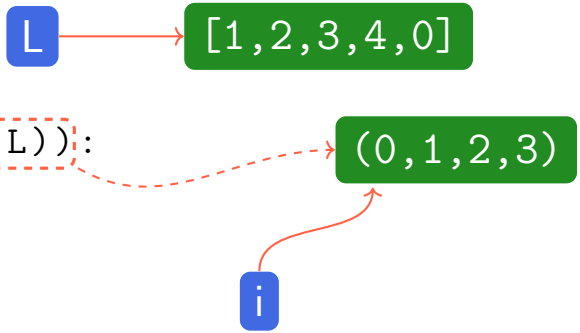
Tricky Example 1:

```
1 L = [1,2,3,4]
2
3 for i in range(len(L)):
4     L.append(i)
5     print(L)
```



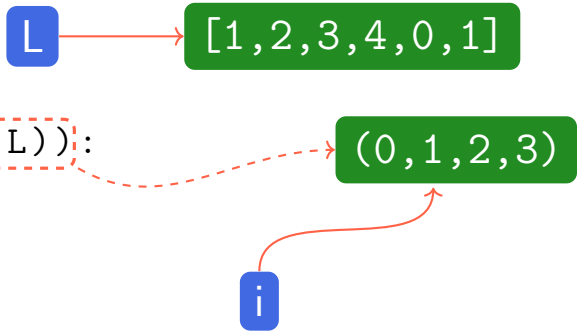
Tricky Example 1:

```
1 L = [1,2,3,4]
2
3 for i in range(len(L)):
4     L.append(i)
5     print(L)
```



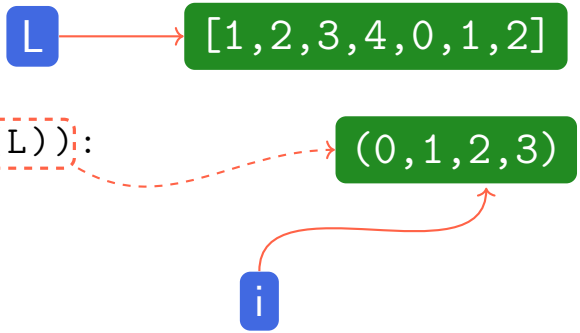
Tricky Example 1:

```
1 L = [1,2,3,4]
2
3 for i in range(len(L)):
4     L.append(i)
5     print(L)
```



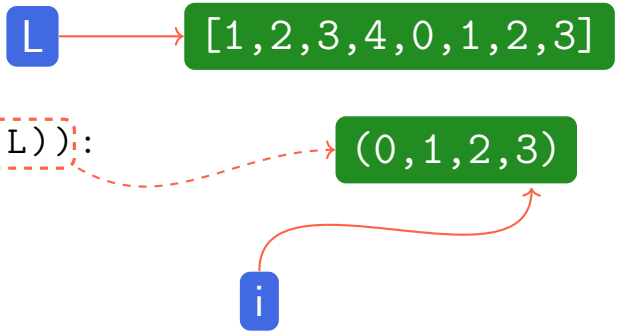
Tricky Example 1:

```
1 L = [1,2,3,4]
2
3 for i in range(len(L)):
4     L.append(i)
5     print(L)
```



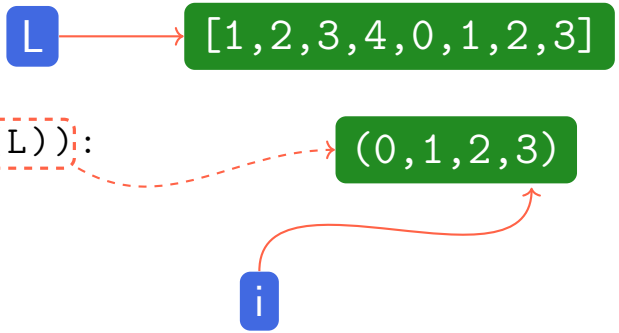
Tricky Example 1:

```
1 L = [1,2,3,4]
2
3 for i in range(len(L)):
4     L.append(i)
5     print(L)
```



Tricky Example 1:

```
1 L = [1,2,3,4]
2
3 for i in range(len(L)):
4     L.append(i)
5     print(L)
```



End of iteration

`i` iterates over a "tuple" created by range. Mutation of `L` does not affect this "tuple"

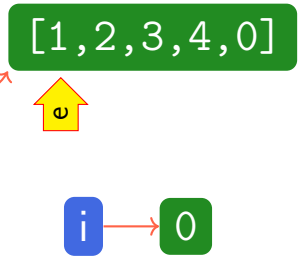
Tricky Example 2:

```
1 L = [1,2,3,4]
2 i = 0
3 for e in L:
4     L.append(i)
5     i += 1
6     print(L)
```



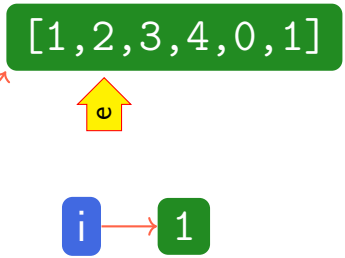
Tricky Example 2:

```
1 L = [1,2,3,4]
2 i = 0
3 for e in L:
4     L.append(i)
5     i += 1
6     print(L)
```



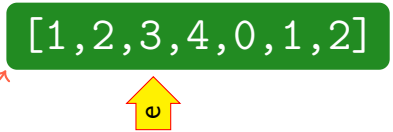
Tricky Example 2:

```
1 L = [1,2,3,4]
2 i = 0
3 for e in L:
4     L.append(i)
5     i += 1
6     print(L)
```



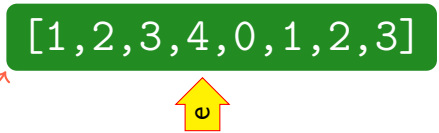
Tricky Example 2:

```
1 L = [1,2,3,4]
2 i = 0
3 for e in L:
4     L.append(i)
5     i += 1
6     print(L)
```



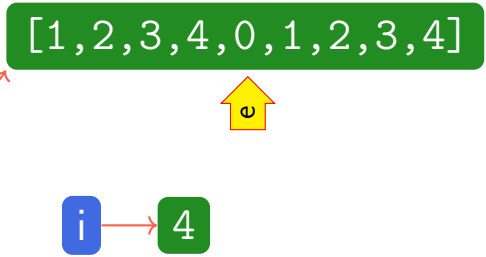
Tricky Example 2:

```
1 L = [1,2,3,4]
2 i = 0
3 for e in L:
4     L.append(i)
5     i += 1
6     print(L)
```



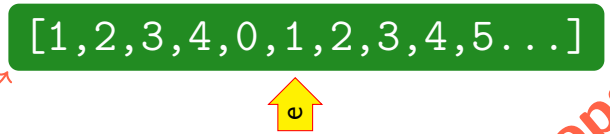
Tricky Example 2:

```
1 L = [1,2,3,4]
2 i = 0
3 for e in L:
4     L.append(i)
5     i += 1
6     print(L)
```



Tricky Example 2:

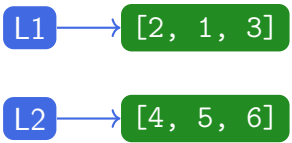
```
1 L = [1,2,3,4]
2 i = 0
3 for e in L:
4     L.append(i)
5     i += 1
6     print(L)
```



Combining Lists: Concatenation

- **Concatenation**, + operator, creates a **new** list, with copies

```
1 L1 = [2,1,3]
2 L2 = [4,5,6]
3 L3 = L1 + L2          # L3 is [2,1,3,4,5,6]
```



Combining Lists: Concatenation

- **Concatenation**, + operator, creates a **new** list, with copies

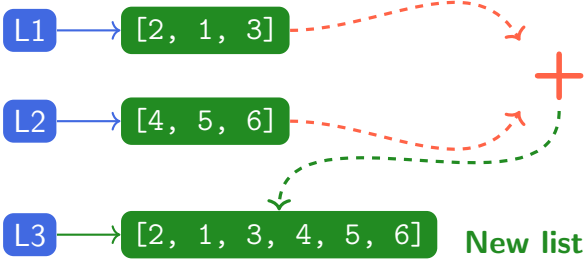
```
1 L1 = [2,1,3]
2 L2 = [4,5,6]
3 L3 = L1 + L2        # L3 is [2,1,3,4,5,6]
```



Combining Lists: Concatenation

- **Concatenation**, + operator, creates a **new** list, with copies

```
1 L1 = [2,1,3]
2 L2 = [4,5,6]
3 L3 = L1 + L2        # L3 is [2,1,3,4,5,6]
```

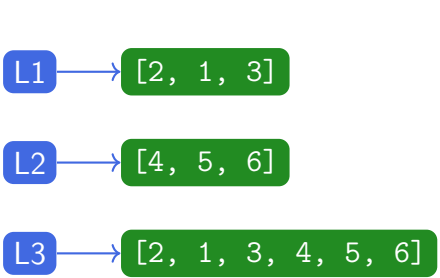


New list created! L1 and L2 unchanged.

Combining Lists: extend()

- **Mutate** list with `L.extend(some_list)`

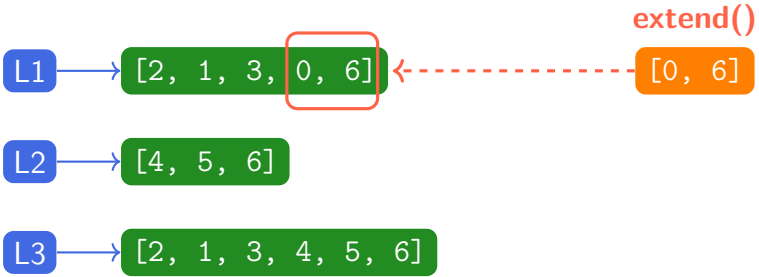
```
1 L1 = [2,1,3]
2 L2 = [4,5,6]
3 L3 = L1 + L2        # L3 is [2,1,3,4,5,6]
4 L1.extend([0,6])    # mutates L1 to [2,1,3,0,6]
```



Combining Lists: extend()

- **Mutate** list with `L.extend(some_list)`

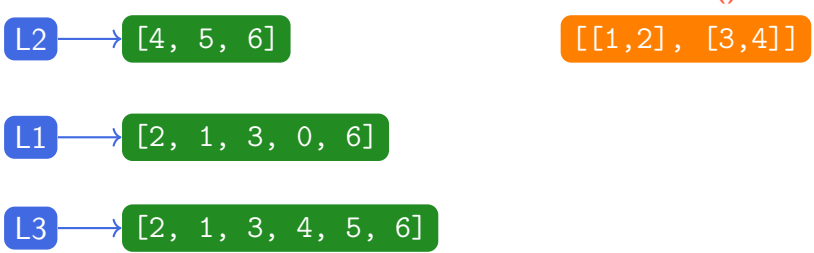
```
1 L1 = [2,1,3]
2 L2 = [4,5,6]
3 L3 = L1 + L2        # L3 is [2,1,3,4,5,6]
4 L1.extend([0,6])    # mutates L1 to [2,1,3,0,6]
```



Combining Lists: extend() with Nested Lists

- `extend()` adds elements from the list, not the list itself

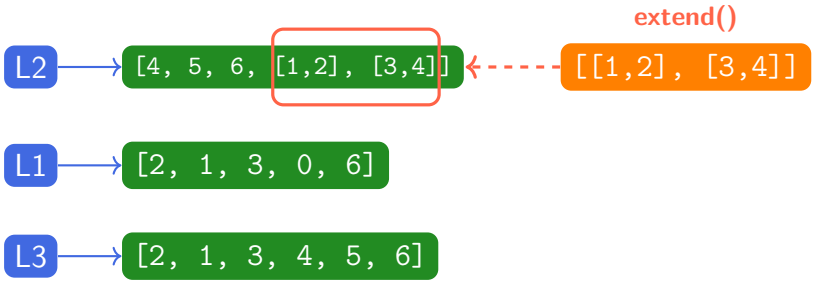
```
1 L1 = [2,1,3]
2 L2 = [4,5,6]
3 L3 = L1 + L2        # L3 is [2,1,3,4,5,6]
4 L1.extend([0,6])    # mutates L1 to [2,1,3,0,6]
5 L2.extend([[1,2],[3,4]])   # mutates L2 to [4,5,6,[1,2],[3,4]]
```



Combining Lists: extend() with Nested Lists

- `extend()` adds elements from the list, not the list itself

```
1 L1 = [2,1,3]
2 L2 = [4,5,6]
3 L3 = L1 + L2        # L3 is [2,1,3,4,5,6]
4 L1.extend([0,6])    # mutates L1 to [2,1,3,0,6]
5 L2.extend([[1,2],[3,4]])  # mutates L2 to [4,5,6,[1,2],[3,4]]
```

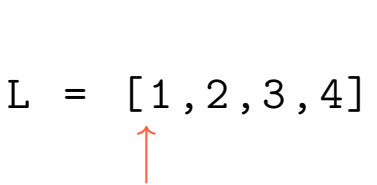


Tricky Example 3:

```
1 L = [1,2,3,4]
2
3 for e in L:
4     L = L + L
5     print(L)
```

Tricky Example 3:

1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]



```
1 L = [1,2,3,4]
2
3 for e in L:
4     L = L + L
5     print(L)
```

Tricky Example 3:

1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]



```
1 L = [1,2,3,4]
2
3 for e in L:
4     L = L + L
5     print(L)
```

Tricky Example 3:

```
1 L = [1,2,3,4]
2
3 for e in L:
4     L = L + L
5     print(L)
```



1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

3rd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

Tricky Example 3:

```
1 L = [1,2,3,4]
2
3 for e in L:
4     L = L + L
5     print(L)
```

- Loop runs **exactly** 4 times
- L is **bount to a new object** each iteration.

But **e** points to the **original** [1,2,3,4]

1st time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4]

2nd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

3rd time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

4th time: **new** L is [1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4
1, 2, 3, 4, 1, 2, 3, 4,
1, 2, 3, 4, 1, 2, 3, 4]

Object Identity

- You can **mutate a list to remove all its elements**
  - This **does not make a new empty list**!
- Use `L.clear()`

Object Identity

- You can **mutate a list to remove all its elements**
  - This **does not make a new empty list**!
- Use `L.clear()`
- How to check that it's the **same object in memory?**
  - Use the `id()` function

Mutation vs Reassignment in Memory

Mutation (Same Object)



Mutation vs Reassignment in Memory

Mutation (Same Object)



`L.append(4)`

Mutation vs Reassignment in Memory



Mutation vs Reassignment in Memory

Mutation (Same Object)



Reassignment (New Object)



Mutation vs Reassignment in Memory

Mutation (Same Object)



Reassignment (New Object)




```
L = [5, 6]
```

Mutation vs Reassignment in Memory



You Try!

Try this in the console:

```
>>> L = [4,5,6]
>>> id(L)
>>> L.append(8)
>>> id(L)
>>> L.clear()
>>> id(L)
```

```
>>> L = [4,5,6]
>>> id(L)
>>> L.append(8)
>>> id(L)
>>> L = []
>>> id(L)
```

Summary

- **Tuples** - Immutable sequences
  - Syntax: `t = (1, 'a', 3.5)`
  - Cannot change elements, great for returning multiple values
- **Lists** - Mutable sequences
  - Syntax: `L = [1, 2, 3]`
  - Methods: `append()`, `sort()`, `reverse()`
  - Conversions: `split()`, `join()`
  - Functions can mutate lists (side effects)

Questions?

Exit Ticket

Today's question *(2 minutes)*

Let `t = (10, 20, 30)` and `L = [10, 20, 30]`

(a) What value is `t[1]`?

(b) Exactly one of `t[0] = 5` and `L[0] = 5` gives an error. Which one?

COMP 102 — Exit Ticket #11

Name: _____

Roll #: _____ Date: _____

Answer:

Hand it to me on your way out.
No name, no attendance.