Lecture Roadmap

- Intro to Programming
- Variables
- Functions
  - The stack
  - Scope
- Flow control
  - `if`
  - `while`
  - `for`
- Strings
- Lists
- Tuples
- Dictionaries

Lists (part II)

COMP1730/COMP6730

Reading: Textbook chapter 10 : Alex Downey, *Think Python*, 2nd Edition (2016)

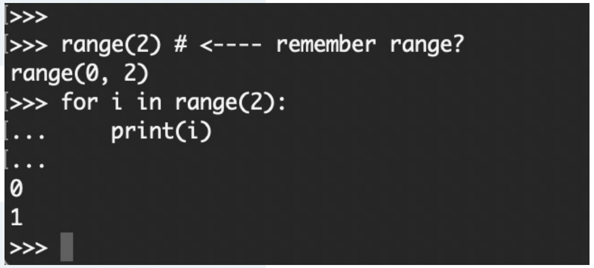
Australian
National
University

List traversal

- Like strings, lists can be traversed with a `for` loop:

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
```

```
for cheese in cheeses:
    print(cheese)
```



```
>>>
>>> range(2) # <---- remember range?
range(0, 2)
>>> for i in range(2):
...     print(i)
...
0
1
>>> 
```

- And modified in the process, if desired:

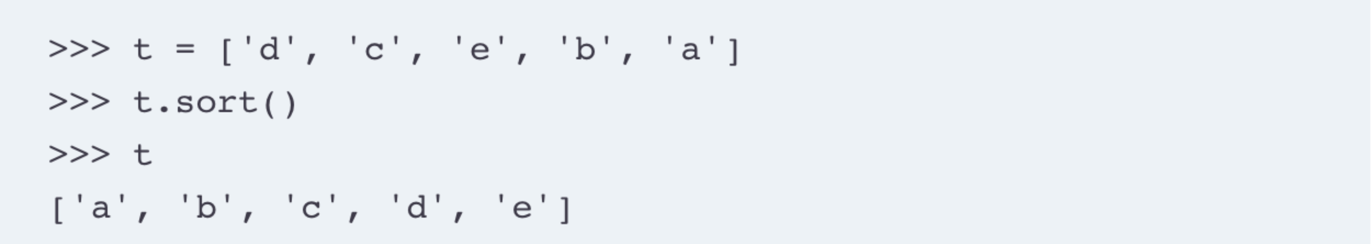
```
>>> numbers = [42, 123]
```

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Downey (2015) Think Python, 2nd Ed.

List methods: `sort()`

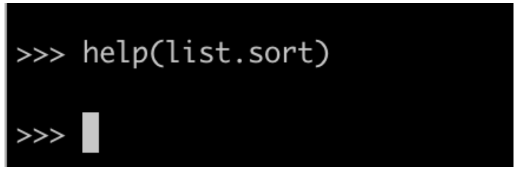
- Sort a list with `sort()`



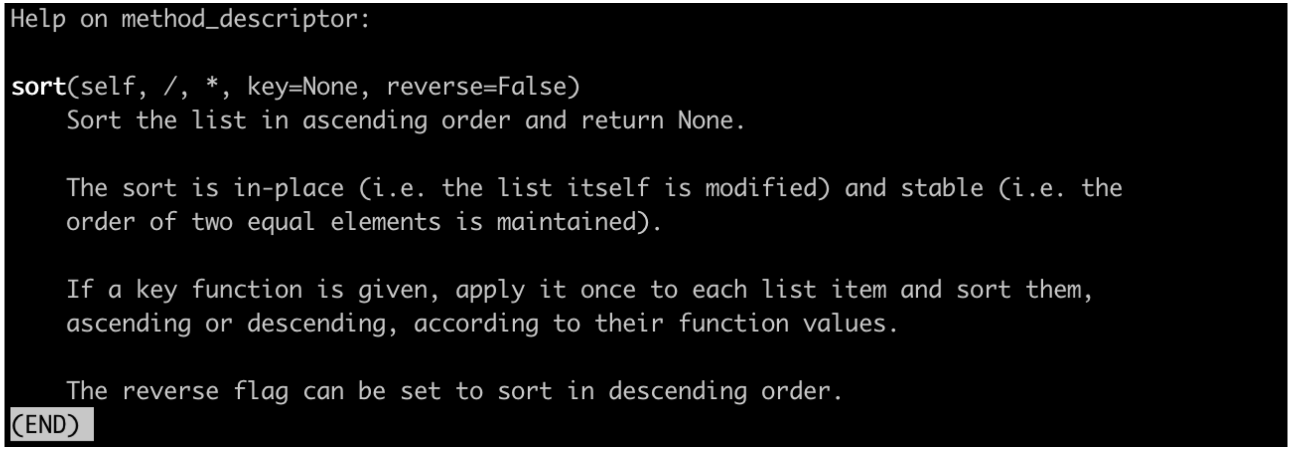
```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

Downey (2015) Think Python, 2nd Ed. (chapter 10)

- Note how the sort is performed on the original list. The result is that the original list is sorted – and does not create a new list.



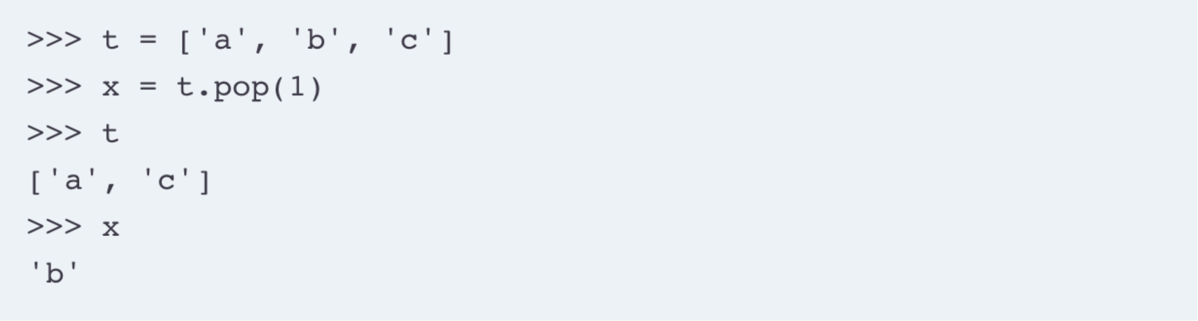
```
>>> help(list.sort)

>>> ▌
```



```
Help on method_descriptor:

sort(self, /, *, key=None, reverse=False)
    Sort the list in ascending order and return None.

    The sort is in-place (i.e. the list itself is modified) and stable (i.e. the
    order of two equal elements is maintained).

    If a key function is given, apply it once to each list item and sort them,
    ascending or descending, according to their function values.

    The reverse flag can be set to sort in descending order.
(END)
```

Deleting list elements: `pop()`

- Lists are mutable, but how to delete an element? With `pop()`.



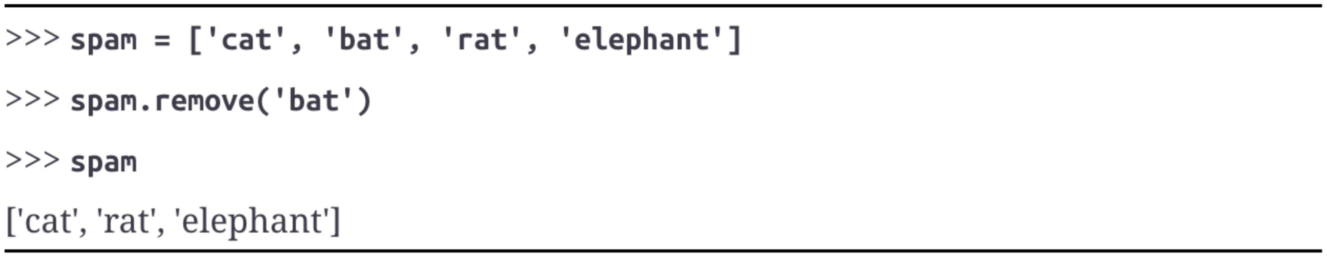
```
>>> t = ['a', 'b', 'c']
>>> x = t.pop(1)
>>> t
['a', 'c']
>>> x
'b'
```

Downey (2015) Think Python, 2nd Ed. (chapter 10)

- The elements with higher indices all shuffle down one, to fill the gap left by the deleted element.
- There are other ways to delete elements, too: the `del` and `remove()` methods. Each with useful features.

Delete by value with `remove()`

- `pop()` deletes whatever value is present at the index specified.
- `remove()` deletes the first occurrence of a particular value:



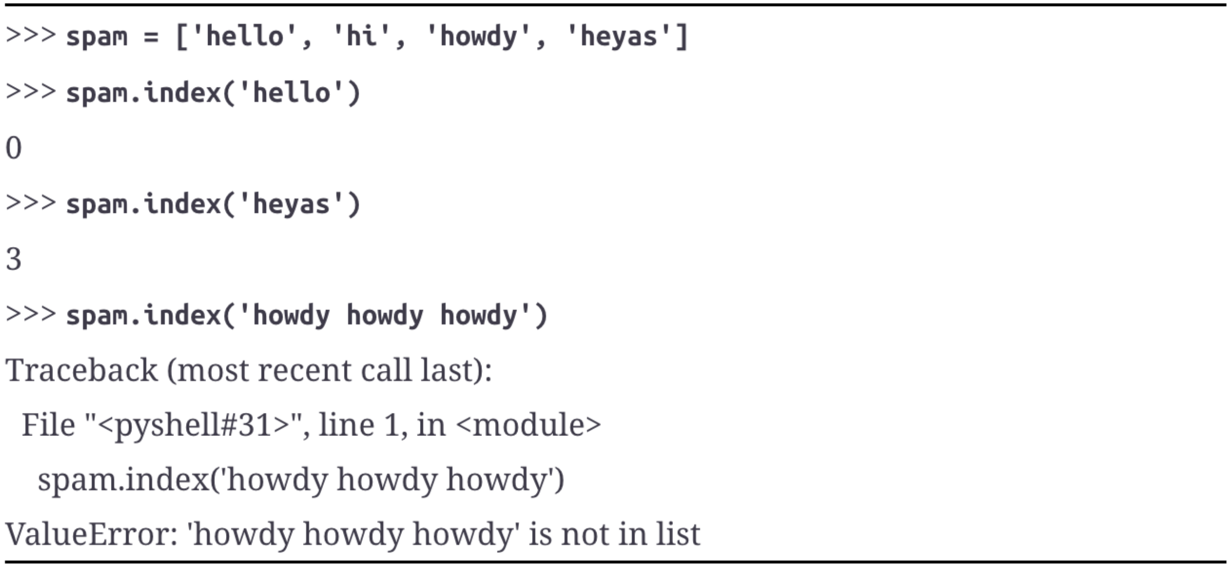
```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

Sweigart (2019) *Automate the boring stuff with python (Chapter 4)*

- It won’t remove further occurrences of the value from the list
- You will also get a `ValueError` error if the list doesn’t contain the value specified

Searching a list with `index()`

- When you pass a value to the list method index(), it will return the index value of that value in the list:



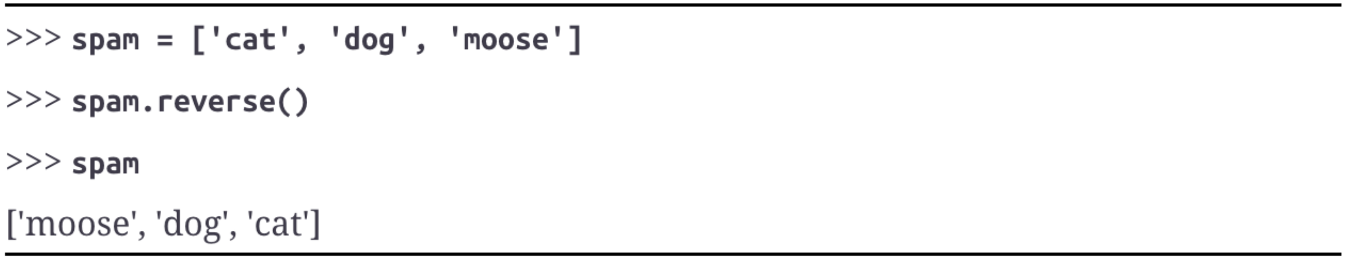
```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<pyshell#31>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

Sweigart (2019) *Automate the boring stuff with python (Chapter 4)*

- Though, if the value isn't present you will get a `ValueError` error

reverse()

- Seemingly trivial, but `reverse()` is useful:



```
>>> spam = ['cat', 'dog', 'moose']
>>> spam.reverse()
>>> spam
['moose', 'dog', 'cat']
```

Sweigart (2019) *Automate the boring stuff with python (Chapter 4)*

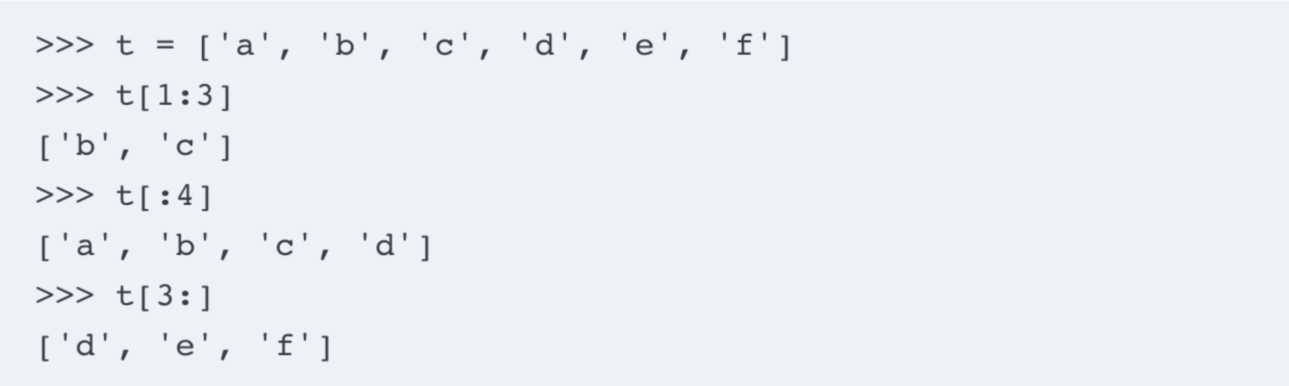
More list methods

- Full list at https://docs.python.org/3/tutorial/datastructures.html

Method	Description
list.**append**(*x*)	Add an item to the end of the list.
list.**extend**(*iterable*)	Extend the list by appending all the items from the iterable.
list.**insert**(*i*, *x*)	Insert an item at a given position.
list.**remove**(*x*)	Remove the first item from the list whose value is equal to *x*.
list.**pop**([*i*])	Remove the item at the given position in the list,
list.**clear**()	Remove all items from the list.
list.**index**(*x*[, *start*[, *end*]])	Return zero-based index in the list of the first item whose value is equal to *x*.
list.**count**(*x*)	Return the number of times *x* appears in the list.
list.**sort**(*, *key=None*, *reverse=False*)	Sort the items of the list in place
list.**copy**()	Return a shallow copy of the list.

List slices

- Use a colon ‘:’ with brackets to specify the range of elements to include – [start:end]
- The start element is included with the returned elements. The end element is not. Remember, this is the ‘half-open’ range.



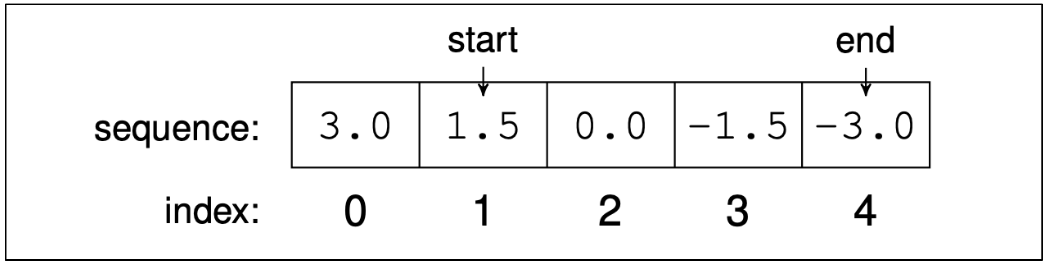
```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

Downey (2015) Think Python, 2nd Ed.

Slices

- Slice syntax: `example_string[start:end]`
  - `start` is the index of the first element
  - `end` the index of the next element past the last (half-open range)
- Slicing works of all built-in sequence types (`str`, `list`, `tuple`) and returns the same type
- If `start` or `end` are left out, they default to the beginning and end

```
>>> x = [3,1.5,0,-1.5,-3]
>>> x[1:4]
[ 1.5, 0, -1.5]
```



Indexes and list length

- Say we have a list:

```
decimal_values = [3.0, 1.5, 0.0, -1.5, -3.0]
```

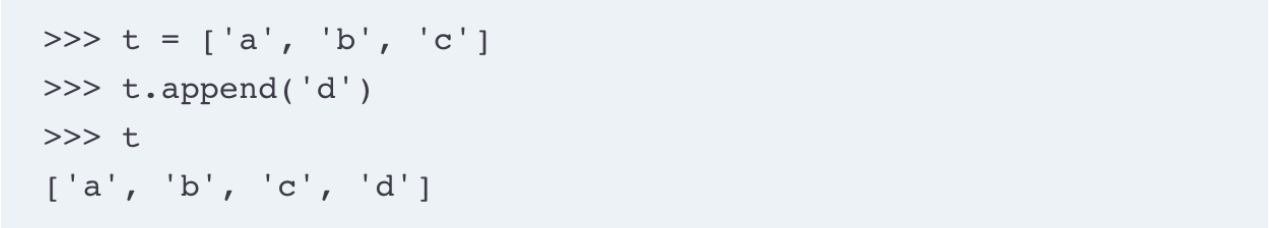
decimal_values:	3.0	1.5	0.0	-1.5	-3.0
Index:	0	1	2	3	4
	-5	-4	-3	-2	-1

- Index starts from 0
- Index numbers must be integers
- Negative integers allow wrap-around of index numbers:

```
decimal_values[0] -> 3.0
decimal_values[-2] -> -1.5
decimal_values[-1] -> -3.0
```

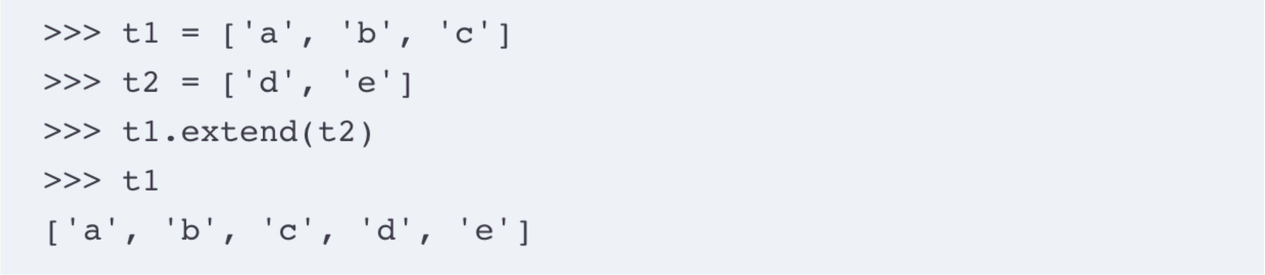
List methods: `append()` and `extend()`

- Add elements to a list with `append()`



```
>>> t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
['a', 'b', 'c', 'd']
```

- Add a list to a list with `extend()`



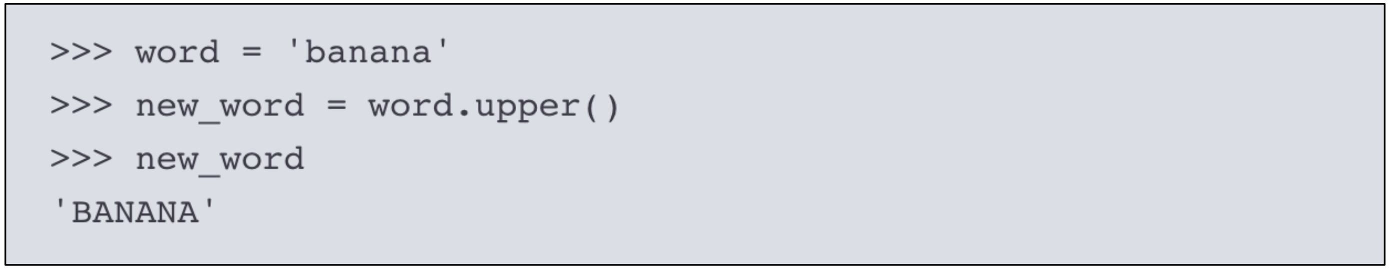
```
>>> t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
['a', 'b', 'c', 'd', 'e']
```

Downey (2015) Think Python, 2nd Ed. (chapter 10)

- `insert()` too

List and string methods – what is different?

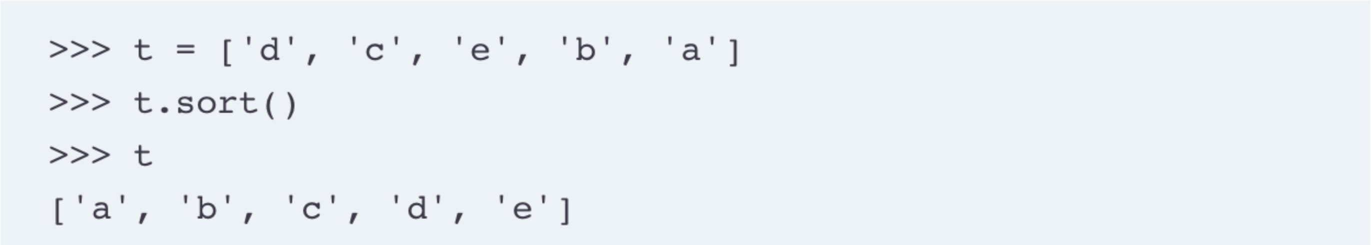
- Remember, Strings are immutable. Lists are mutable.
- A method on a string can’t change that string – so methods create a new string:



```
>>> word = 'banana'
>>> new_word = word.upper()
>>> new_word
'BANANA'
```

Downey (2015) Think Python, 2nd Ed.

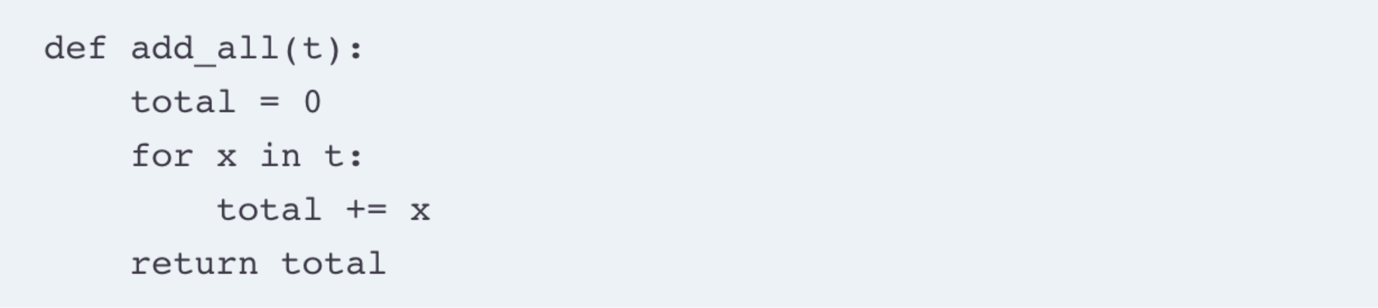
- A method on a list can CHANGE THE LIST. It makes sense, but can catch you out when you are starting to program:



```
>>> t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
['a', 'b', 'c', 'd', 'e']
```

list example

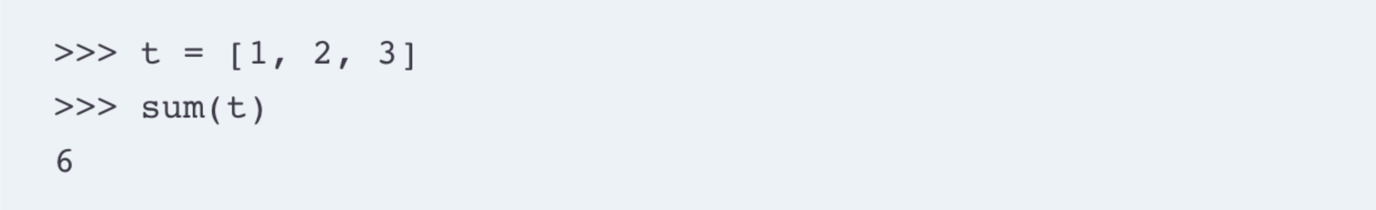
- Say you needed to add the numbers in a list:



```
def add_all(t):
    total = 0
    for x in t:
        total += x
    return total
```

Downey (2015) Think Python, 2nd Ed. (chapter 10)

- Though, in reality, this is redundant because of the `sum()` function:



```
>>> t = [1, 2, 3]
>>> sum(t)
6
```

Downey (2015) Think Python, 2nd Ed. (chapter 10)

Creating lists

- You can use different ways to create a list:

```
my_list = list()   # creates an empty list
my_list = list([1,2,3,4])  # creates a list with the list argument supplied
my_list = [1,2,3,4]  # the same thing
```

- Say, you want to perform an operation on the list at the same time:

```
precise = [1.23, 1.99, 2.01, 2.51, 3.45]
rounded = []

for number in precise:
       rounded_number = round(number)
       rounded.append(rounded_number)
```

- `rounded` **becomes** `[1, 2, 2, 3, 3]`

Creating lists with comprehensions

- Alternatively, you can use a python short-hand called a **list comprehension**
- This:

```
precise = [1.23, 1.99, 2.01, 2.51, 3.45]
rounded = []

for number in precise:
       rounded_number = round(number)
       rounded.append(rounded_number)
```

- Becomes this:

```
precise = [1.23, 1.99, 2.01, 2.51, 3.45]
rounded = [round(number) for number in precise]
```

Unpacking a List Comprehension

- This is the syntax of a list comprehension:

```
new_list = [expression for item in list]
```

```
precise = [1.23, 1.99, 2.01, 2.51, 3.45]
rounded = [round(number) for number in precise]
```

```
precise = [1.23, 1.99, 2.01, 2.51, 3.45]
rounded = []

for number in precise:
        rounded_number = round(number)
        rounded.append(rounded_number)
```

Another example

- Creating a new list, containing the values transformed from another list:

```
raw_text = [' and ', ' is ', ' however ']
cleaned_text = []

for word in raw_text:
       word_no_spaces = word.strip()
       cleaned_text.append(word_no_spaces)
```

- Remember: `new_list = [expression for item in list]`

```
raw_text = [' and ', ' is ', ' however ']
cleaned_text = [word.strip() for word in raw_text]
```

- The new list `cleaned_text` is `['and', 'is', 'however']`

List comprehensions with added `if`

- It is possible to also filter with an `if` at the same time:

```
new_list = [expression for item in list if condition]
```

```
small_integers = [1,2,3,4,5,6,7,8,9]
even_integers = []

for number in small_integers:
       if number % 2 == 0:
              even_integers.append(number)
```

```
small_integers = [1,2,3,4,5,6,7,8,9]
even_integers = [number for number in small_integers if number % 2 == 0]
```

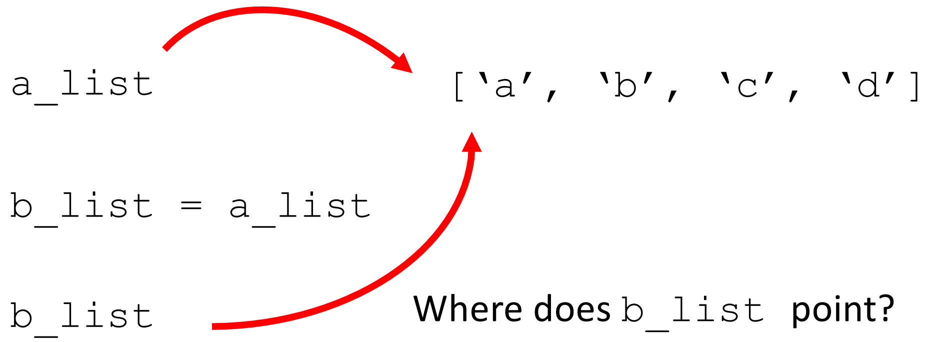
Specific reading for list comprehension

- If you are lost:
  - Lubanovic (2019) *Introducing python* –2nd Ed.
    - Chapter 7: Create a List with a Comprehension
    - This is clear and about two pages long

References and Lists - a trap for the unwary

- In python, the value held in the list variable is **a reference**
  - not the actual values
- References are a new concept
  - References can be thought of as addresses. With a street address, you should be able to find a house
  - References are memory addresses. With a reference, python knows where to find the value of a variable
- The value stored in the list name variable is the reference

```
a_list = [‘a’, ‘b’, ‘c’, ‘d’]
```



```
a_list                [‘a’, ‘b’, ‘c’, ‘d’]

b_list = a_list

b_list
```

Where does b_list point?

A List is an address

- If you forget that your list variable is a reference, you might get a surprise:

```
>>> a_list = [‘zero’, ‘one’, ‘two’]
>>> print(a_list)
[‘zero’, ‘one’, ‘two’]
>>>
>>> b_list = a_list
>>> b_list[1] = ‘four’
>>>
>>> print(a_list)
[‘zero’, ‘four’, ‘two’]
>>>
>>> id(a_list)
140384948070336
>>> id(b_list)
140384948070336
>>>
```

Same address!

When in doubt, make copies

```
copy_list = original_list[:]
another_copy = orginal_list.copy()
```

```
>>> a_list = [‘zero’, ‘one’, ‘two’]
>>> b_list = a_list[:]
>>> c_list = a_list.copy()
>>>
>>> id(a_list)
140384948070336
>>> id(b_list)
140384944782144
>>> id(c_list)
140384948106240
>>>
>>> b_list[1] = ‘four’
>>> c_list[2] = ‘five’
>>> print(b_list)
[‘zero’, ‘four’, ‘two’]
```

Graphically:

```
>>> a_list = [1,2,3]
>>> b_list = a_list
```

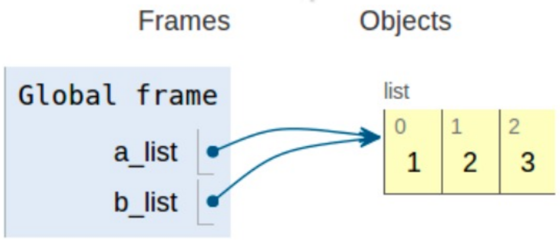


Image from `pythontutor.com`

```
>>> a_list = [1,2,3]
>>> b_list = a_list[:]
```

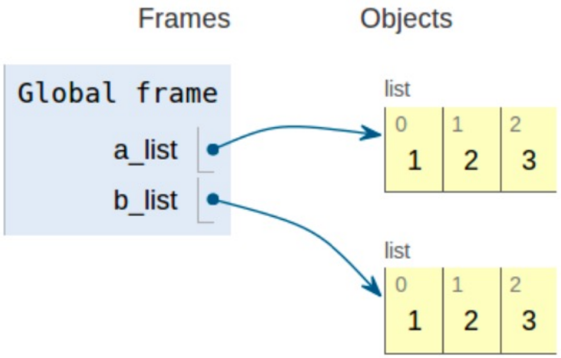


Image from `pythontutor.com`

Multi-dimension lists

- Remember, that lists may contain other lists:

```
>>> a_list = [‘zero’, ‘one’, ‘two’]
>>> b_list = [0.11, 1.03, 2.01]
>>> c_list = [0, 1, 2]
>>>
>>> list_of_lists = [a_list, b_list, c_list]
>>> list_of_lists
[['zero', 'one', 'two'], [0.11, 1.03, 2.01], [0, 1, 2]]
>>>
>>> list_of_lists[0][2]
'two'
>>> list_of_lists[1][0]
0.11
>>>
>>> another_list_of_lists = list_of_lists.copy()
>>> another_list_of_lists[0][2] = ‘three’
>>> list_of_lists
[['zero', 'one', ‘three'], [0.11, 1.03, 2.01], [0, 1, 2]]
```

deepcopy() of multi-dimensional lists

```
>>> import copy
>>>
>>> a_list = [‘zero’, ‘one’, ‘two’]
>>> b_list = [0.11, 1.03, 2.01]
>>> c_list = [0, 1, 2]
>>>
>>> list_of_lists = [a_list, b_list, c_list]
>>> list_of_lists
[['zero', 'one', 'two'], [0.11, 1.03, 2.01], [0, 1, 2]]
>>>
>>> another_list_of_lists = copy.deepcopy(list_of_lists)
>>>
>>> id(list_of_lists[0])
140384948106432
>>> id(another_list_of_lists[0])
140384948110784
```

- `deepcopy()` will copy very deep multi-dimensional lists

Passing lists to functions as arguments

Australian
National
University

- Be aware that when you pass a list to a function, you are just passing the address:

```
def bad_sort(input_list):
       input_list.sort()
       return input_list

original_list = [4, 2, 1, 0, 3]

new_list = bad_sort(original_list)

print(original_list)
print(new_list)
```

- Output:

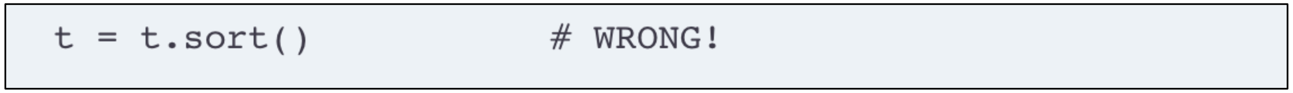
```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4]
```

Advice for using lists (*Think Python* Ch 10)

Australian
National
University

(or spend hours debugging your code)

1. Most list methods modify the argument (the list itself) and return `None`. Watch out that you aren't doing this:



```
t = t.sort()                # WRONG!
```

2. There are so many ways to manipulate lists – choose your style and don't worry about what you don't use.
    - For example, `pop()`, `del` and `remove()` all kind-of do the same thing – but the different 'features' of each specific method might catch you by surprise.
3. Make copies of most of your lists (especially if they are small) to avoid inadvertently modifying other lists via references

Exercises

- Exercises 10-1, 10-3 and 10-4, *Think Python* Ch. 10

Reading

- *Think Python* Ch 10
- But do have a look at Lubanovic (2019) *Introducing python* (ch. 7) if list comprehensions were a little bit incomprehensible to you.

Tuples

COMP1730/COMP6730

Reading: Textbook chapter 12 : Alex Downey, *Think Python*, 2nd Edition (2016)

Sections: *Tuples are immutable, Tuple assignment, Tuples as return values*

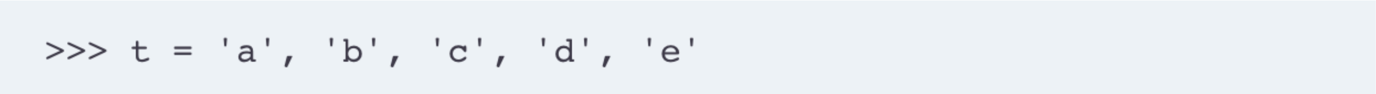
Australian
National
University

Lists versus Tuples

- Both a sequences.
- Lists are mutable. Tuples are immutable. Otherwise, they are very similar.
- There are good reasons for using tuples in certain circumstances:
    - Performance – if a list won’t change, the python interpreter can make optimisations
    - Hands off – sometimes it is better to not be able to change (or have something else change) the values in your sequence.

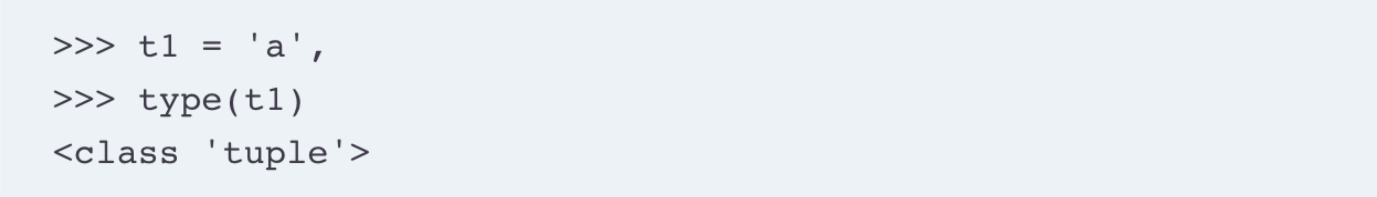
Tuples?

- Tuples are immutable. So think of them like lists that can't be changed.
- A comma-separated sequence of values (with or without parentheses):



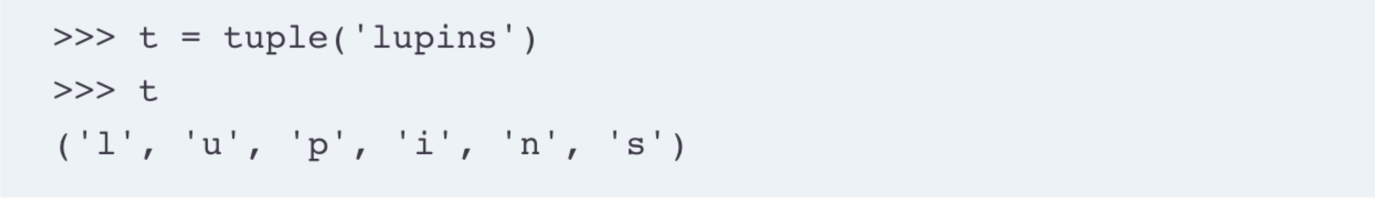
```
>>> t = 'a', 'b', 'c', 'd', 'e'
```

- Create with a trailing comma:



```
>>> t1 = 'a',
>>> type(t1)
<class 'tuple'>
```

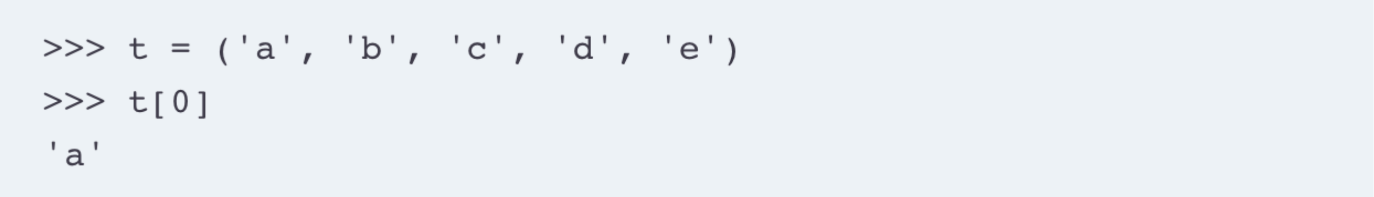
- Or with the the `tuple()` function:



```
>>> t = tuple('lupins')
>>> t
('l', 'u', 'p', 'i', 'n', 's')
```

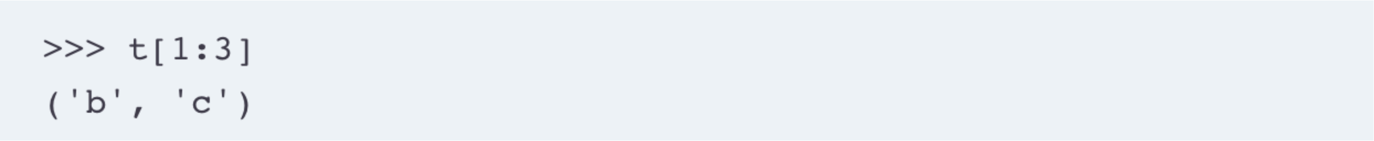
Tuples work mostly like lists

- Elements in a tuple can be accessed by indexes:



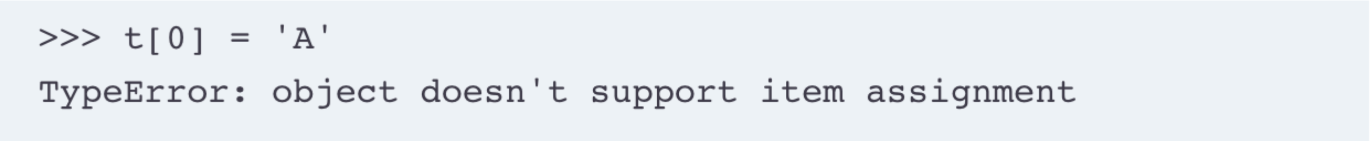
```
>>> t = ('a', 'b', 'c', 'd', 'e')
>>> t[0]
'a'
```

- And slices can be made from tuples:



```
>>> t[1:3]
('b', 'c')
```

- But they can’t be changed:



```
>>> t[0] = 'A'
TypeError: object doesn't support item assignment
```

Why, tuples?

- They make excellent `return` values from a function

- And are good protection from unintended side-effects of functions on your data structures

Exercises

- Only if you want – try a few at the end of *Think Python* Ch. 11

Reading

- *Think Python* Ch 11