Lecture 2.1 Lists

Introduction

- Python's built-in list type is a *collection type* (meaning it contains a number of objects that can be treated as one object). It is also a *sequence type* meaning each object in the collection occupies a specific numbered location within it i.e. elements are *ordered*. The list is an *iterable type* meaning we can use a loop to inspect each of its elements in turn.

- So far a list sounds exactly like a string. However a list differs in two significant ways:

    1. A list can contain objects of *differing* and *arbitrary* types. (A string is made up entirely of objects of the same type, namely, characters.)
    2. A list is a *mutable* type. This means it can be modified after initialisation. (A string is an *immutable* type. It cannot be modified after creation.)
- As with strings, to select a particular element in a list we index into it using square brackets. The first element of the list is located at index zero. The last element is at index N-1 in a list of length N.

- Below we demonstrate the contrasting and common properties of lists and strings.

```
>>> l = [1, 2.2, 3, 'oranges', [4, 5.5, 6] ] # Lists can contain objects of arb
>>> l
[1, 2.2, 3, 'oranges', [4, 5.5, 6]]
>>> for i in l: # Lists are iterable
...     print('{} is of type {}'.format(i, type(i)))
...
1 is of type <class 'int'>
2.2 is of type <class 'float'>
3 is of type <class 'int'>
oranges is of type <class 'str'>
[4, 5.5, 6] is of type <class 'list'>
>>> l = ['a', 'p', 'p', 'l', 'e']
>>> l[0] = 'A' # Lists are mutable
>>> l
['A', 'p', 'p', 'l', 'e']
>>> l.append('s') # Lists are mutable
>>> l
['A', 'p', 'p', 'l', 'e', 's']
>>> l[0] # We select particular elements with square brackets
'A'
>>> l[1]
'p'
>>> l[-1]
's'
>>> l[-2]
'e'
>>> s = 'apple' # Strings contain only characters
>>> s[0] = 'A' # Strings are immutable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- We see above that a list can contain objects of any type, even other lists. Lists of lists are useful for representing many data types e.g. spreadsheets, matrices, images, etc. To select a particular

element in a nested (i.e. embedded) list we first select the embedded list and then select the element. Each of these selection operations requires the use of square brackets:

```
>>> l = [ [1,2,3], [4,5,6] ]
>>> l[0]
[1, 2, 3]
>>> l[1]
[4, 5, 6]
>>> l[0][0]
1
>>> l[0][1]
2
>>> l[0][2]
3
>>> l[1][0]
4
>>> l[1][1]
5
>>> l[1][2]
6
```

- A list with no elements is the empty list, `[]`, and being interpreted as `False` means we can write code like this:

```
>>> l = ['A', 'p', 'p', 'l', 'e', 's']
>>> while l:
...     print(l.pop())
...
s
e
l
p
p
A
```

List slicing and extended slicing

- Slicing and extended slicing work exactly as they do for strings. (This makes sense as both lists and strings are sequence types.)

List operators

- The `+`, `*`, `in` and == operators function identically for lists and strings, Lists may be concatenated and replicated with the `+` and `*` operators. We use the `in` operator to test for membership of a list. We use == to test for list equality.

List functions

- `len(L)` returns the number of elements in L.
- `min(L)` returns the minimum element in L.
- `max(L)` returns the maximum element in L.
- `sum(L)` returns the sum of the elements in L (L must be a list of numbers).

List methods

- Python comes with built-in support for a set of common list operations. These operations are called `methods` and they define the things we can do with lists. Calling `help(list)` or `pydoc list` outputs a list of these methods. We see the methods we can invoke on a list `l` include `append()` (adds an object to the end of `l`), `clear()` (removes all elements from `l`), `pop()` (removes and returns the last element in `l`), `reverse()` (reverses the elements of `l` in place), `sort()` (sorts the elements of `l` in place), etc.
- Note that because lists are *mutable* calling a method on a list may alter the list itself. (Contrast this behaviour with that of string methods.)
- Whenever you find it necessary to carry out some list processing first look up the available built-in list methods. There may be one that will help you with your task. There is no point writing your own code that duplicates what a built-in list method can do for you already.

From strings to lists and back again

- We can use the `split()` method to convert a string to a list. We can then modify the list (remember lists are mutable while strings are immutable) before converting back to a string with the `join()` method. Suppose every student's list of marks is available as a string such as `"Mary Rose O'Reilly 40 45 60 70 55"` and we want to replace each student's set of marks with a single average mark. How might we go about it?

```
>>> s = "Mary Rose O'Reilly 40 45 60 70 55"
>>> s.split()
['Mary', 'Rose', "O'Reilly", '40', '45', '60', '70', '55']
>>> tokens = s.split()
>>> tokens[-5:]
['40', '45', '60', '70', '55']
>>> total = 0
>>> for m in tokens[-5:]:
...     total += m
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: unsupported operand type(s) for +=: 'int' and 'str'
>>> for m in tokens[-5:]:
...     total += int(m)
...
>>> total
270
>>> 40+45+60+70+55
270
>>> total/5
54.0
>>> total//5
54
>>> average = total//5
>>> s
"Mary Rose O'Reilly 40 45 60 70 55"
>>> tokens
['Mary', 'Rose', "O'Reilly", '40', '45', '60', '70', '55']
>>> tokens[:-5]
['Mary', 'Rose', "O'Reilly"]
>>> l = tokens[:-5]
>>> l.append(average)
>>> l
['Mary', 'Rose', "O'Reilly", 54]
>>> ' '.join(l)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 3: expected str instance, int found
>>> l.pop()
54
>>> l
['Mary', 'Rose', "O'Reilly"]
>>> l.append(str(average))
>>> l
['Mary', 'Rose', "O'Reilly", '54']
>>> ' '.join(l)
"Mary Rose O'Reilly 54"
```

```
# string2list.py
def append_average(s):
    tokens = s.split()
    total = 0
    for mark in tokens[-5:]:
        total += int(mark)
    total //= 5
    l = tokens[:-5]
    l.append(str(total))
    return ' '.join(l)
```

The `sorted()` function

- The `sort()` method works only with lists. How can we sort the characters in a string? We could convert the string to a list, invoke the `sort()` method on the list and then convert the sorted list back to a string with the `join()` method. That's quite a bit of work. Is there a handier way? Yes. The `sorted()` function converts a collection to a list and returns the sorted list:

```
>>> s = 'gfedcba'
>>> sorted(s)
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> ''.join(sorted(s))
'abcdefg'
```