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 seems similar to a string. However a list differs in two significant ways:

    1. A list can contain objects of *differing* and *arbitrary* type. (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.)

Indexing and slicing lists

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

Lists in action

- Below we explore some list properties and demonstrate associated methods:

```
# Create a list with []
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

```
# Reverse a list with extended slicing
print(days[::-1])
```

```
['Friday', 'Thursday', 'Wednesday', 'Tuesday', 'Monday']
```

```
# Add to a list
days.append('Saturday')
days.append('Sunday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
# Check list membership
print('Friday' in days)
print('April' in days)
```

```
True
False
```

```
# Find the index of a list element
print(days.index('Tuesday'))
```

```
1
```

```
# Remove and return an item by index
i = days.index('Tuesday')
print(days.pop(i))
print(days)
```

```
Tuesday
['Monday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
# Remove first occurrence of an item by value (without returning it)
days.remove('Wednesday')
print(days)
```

```
['Monday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
# Insert an item at a particular index (bumping elements to the right)
days.insert(1, 'Tuesday')
days.insert(2, 'Wednesday')
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
# Remove and return the last item in the list
print(days.pop())
print(days)
```

```
Sunday
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday']
```

```
# Delete an item by index from the list (without returning it)
del(days[-1])
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

```
# Combine two lists with extend
days.extend(['Saturday', 'Sunday'])
print(days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
# Combine a list of strings into a single string
print(' '.join(days))
```

```
Monday Tuesday Wednesday Thursday Friday Saturday Sunday
```

```
# Update each element (proving lists are mutable)
i = 0
N = len(days)
while i < N:
    days[i] = days[i].lower()
    i += 1
print(days)
```

```
['monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday', 'sunday']
```

```
# Iterate over each element with a for loop to build a new list
capdays = list()
for day in days:
    capdays.append(day.capitalize())
print(capdays)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

```
# Sort a list in place
days.sort()
print(days)
```

```
['friday', 'monday', 'saturday', 'sunday', 'thursday', 'tuesday', 'wednesday']
```

Lists are `True` or `False`

- The empty list `[]` is interpreted as `False`.
- Any non-empty list is `True`.
- Because lists have truth values we can write code like this:

```
while (capdays):
    print(capdays.pop())
```

```
Sunday
Saturday
Friday
Thursday
Wednesday
Tuesday
Monday
```

List concatenation, replication, copying, comparison

```
# Create two lists
alist = [1, 2, 3]
blist = [4, 5, 6]

# Extend alist (with the contents of blist)
alist += blist
print(alist)

# Replicate blist
blist *= 3
print(blist)

# Make clist a copy of alist and compare
clist = alist[:]
print(clist == alist)
```

```
[1, 2, 3, 4, 5, 6]
[4, 5, 6, 4, 5, 6, 4, 5, 6]
True
```

Lists of lists

- 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:

```
lol = [ [1, 2, 3], ['a', 'b', 'c'] ]
print(lol[0])
print(lol[1])
print(lol[0][-1])
print(lol[1][1])
```

```
[1, 2, 3]
['a', 'b', 'c']
3
b
```

From strings to lists and back again

- Suppose every student's list of marks is available as a string e.g. `"Mary Rose O'Reilly-McCann 40 45 60 70 55"`
- We want to replace each student's set of marks with their min, max and average mark. How might we go about it?

- The length of each student's name is variable but the number of marks is fixed. We can take advantage of that.

```
line_in = "Mary Rose O'Reilly-McCann 40 45 60 70 55"
NUM_MARKS = 5

# Split the line into its constituent tokens
tokens = line_in.strip().split()
print(tokens)

# Extract the list of marks
marks_as_strings = tokens[-NUM_MARKS:]
print(marks_as_strings)

# Convert marks to integers
marks_as_ints = list()
for mark in marks_as_strings:
   marks_as_ints.append(int(mark))
print(marks_as_ints)

# Extract min, max and calculate average
min_mark = min(marks_as_ints)
max_mark = max(marks_as_ints)
avg_mark = sum(marks_as_ints) // NUM_MARKS

# Convert new marks to strings and put in a list
marks_as_strings = [str(min_mark), str(max_mark), str(avg_mark)]
print(marks_as_strings)

# Combine our two lists to rebuild tokens
tokens = tokens[:-NUM_MARKS] + marks_as_strings
print(tokens)

# Turn our list of strings into one string
print(' '.join(tokens))

# Phew!
```

```
['Mary', 'Rose', "O'Reilly-McCann", '40', '45', '60', '70', '55']
['40', '45', '60', '70', '55']
[40, 45, 60, 70, 55]
['40', '70', '54']
['Mary', 'Rose', "O'Reilly-McCann", '40', '70', '54']
Mary Rose O'Reilly-McCann 40 70 54
```

Multiple assignment

- Lists provide us with the opportunity to highlight a handy python feature called multiple assignment.

```
# Here is a list with three elements
print(marks_as_strings)

# We can assign a name to each list element in a single line of code
[min_mark, max_mark, avg_mark] = marks_as_strings

# Check it
print(f'min_mark: {min_mark}')
```

```
print(f'max_mark: {max_mark}')
print(f'avg_mark: {avg_mark}')
```

```
['40', '70', '54']
min_mark: 40
max_mark: 70
avg_mark: 54
```

- This is also sometimes referred to as list unpacking.

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.
- 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.

```
help(list)
```

```
Help on class list in module builtins:

class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
 |
 |  __gt__(self, value, /)
```

```
|      Return self>value.
|
|  __iadd__(self, value, /)
|      Implement self+=value.
|
|  __imul__(self, value, /)
|      Implement self*=value.
|
|  __init__(self, /, *args, **kwargs)
|      Initialize self.  See help(type(self)) for accurate signature.
|
|  __iter__(self, /)
|      Implement iter(self).
|
|  __le__(self, value, /)
|      Return self<=value.
|
|  __len__(self, /)
|      Return len(self).
|
|  __lt__(self, value, /)
|      Return self<value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __reversed__(self, /)
|      Return a reverse iterator over the list.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __setitem__(self, key, value, /)
|      Set self[key] to value.
|
|  __sizeof__(self, /)
|      Return the size of the list in memory, in bytes.
|
|  append(self, object, /)
|      Append object to the end of the list.
|
|  clear(self, /)
|      Remove all items from list.
|
|  copy(self, /)
|      Return a shallow copy of the list.
|
|  count(self, value, /)
|      Return number of occurrences of value.
|
|  extend(self, iterable, /)
|      Extend list by appending elements from the iterable.
|
|  index(self, value, start=0, stop=9223372036854775807, /)
|      Return first index of value.
|
|      Raises ValueError if the value is not present.
|
|  insert(self, index, object, /)
|      Insert object before index.
|
```

```
 |  pop(self, index=-1, /)
 |      Remove and return item at index (default last).
 |
 |      Raises IndexError if list is empty or index is out of range.
 |
 |  remove(self, value, /)
 |      Remove first occurrence of value.
 |
 |      Raises ValueError if the value is not present.
 |
 |  reverse(self, /)
 |      Reverse *IN PLACE*.
 |
 |  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.
 |
 |  ----------------------------------------------------------------------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __hash__ = None
```