Custom Sequences

Part 2

Concatenation and In-Place Concatenation

When dealing with the + and += operators in the context of sequences

we usually expect them to mean concatenation

But essentially, it is just an overloaded definition of these operators

We can overload the definition of these operators in our custom classes by using the methods:

`__add__` `__iadd__`

In general (but not necessarily), we expect:

`obj1 + obj2` → `obj1` and `obj2` are of the same type

→ result is a new object also of the same type

`obj1 += obj2` → `obj2` is any iterable

→ result is the original `obj1` memory reference

(i.e. `obj1` was mutated)

Repetition and In-Place Repetition

When dealing with the `*` and `*=` operators in the context of sequences

we usually expect them to mean repetition

But essentially, it is just an overloaded definition of these operators

We can overload the definition of these operators in our custom classes by using the methods:

`__mul__`   `__imul__`

In general (but not necessarily), we expect:

`obj1 * n` → `n` is a non-negative integer

→ result is a new object of the same type as `obj1`

`obj1 *= n` → `n` is a non-negative integer

→ result is the original `obj1` memory reference

(i.e. `obj1` was mutated)

Assignment

We saw in an earlier lecture how we can implement accessing elements in a custom sequence type

`__getitem__` → `seq[n]`

→ `seq[i:j]`

→ `seq[i:j:k]`

We can handle assignments in a very similar way, by implementing `__setitem__`

There a few restrictions with assigning to slices that we have already seen (at least with lists):

For any slice we could only assign an iterable

For extended slices only, both the slice and the iterable must have the same length

Of course, since we are implementing `__setitem__` ourselves, we could technically make it do whatever we want!

Additional Sequence Functions and Operators

There are other operators and functions we can support:

`__contains__`	`in`
`__delitem__`	`del`
`__rmul__`	`n * seq`

The way Python works is that when it encounters an expression such as:

	`a + b`	`a * b`
it first tries	`a.__add__(b)`	`a.__mul__(b)`

if a does not support the operation (`TypeError`), it then tries:

`b.__radd__(a)`   `b.__rmul__(a)`

Implementing `append`, `extend`, `pop`

Actually there's nothing special going here.

If we want to, we can just implement methods of the same name (not special methods)

and they can just behave the same way as we have seen for lists for example

Code Exercises