Sequences

Ranges and strings are types of sequence we have seen already.

Lists

A list is a type of sequence that allows you to group bunches of numbers together.

Example: `[21, 46, 84, 17]`, a list of four ints.

You can also have `[]`, the empty list.

List Comprehensions

An efficient way of generating large lists. Consists of an expression and a generator:

```
[n for n in range(1, 6)] => [1, 2, 3, 4, 5]
```

- In this code, `n` is the expression, and `for n in range(1, 6)` is the generator.

The generator cranks out numbers, which are fed into the expression to give the output for each step of the generator, and these outputs are stored in a list.

The expression can be as simple or complex as you like, and doesn't have to be related to the generator:

```
['hello' for _ in range(5)] => ['hello', 'hello', 'hello', 'hello', 'hello']
```

You can also include function calls in the expression:

```
[SmallestFactor(n) for n in range(2, 11)]
```

A list comprehension will return `[]` when the range is empty in the generator.

Filter

If you want to generate a bunch of numbers but only include some of them, you can use the optional filter parameter:

```
[n * n for n in range(20) if n * n > 60]
```

Concatentation

The `+` operator can be used for concatenation of lists as well as for strings:

```
[1, 2] + [3, 4, 5] => [1, 2, 3, 4, 5]
```

This can be useful for generating lists:

```
def Reverse(lst):

    reverse = []

    for item in lst:
        reverse = [item] + reverse

    return reverse
```

Strings

A type of sequence that groups characters together.

Example: `'abc def'`, a string with seven characters (including the space).

You can also have `''`, the empty string.

Concatenation

The reverse function for lists defined above won't work exactly for strings, it needs adjusting:

```
def Reverse(string):

    reverse = ''

    for char in string:
        reverse = char + reverse

    return reverse
```

- Note that you don't need quote marks around `char` in the loop, as it is already a character, unlike in the version for lists, where you needed square brackets.

'sum' and 'max'

Two inbuilt functions that work on sequences.

`sum(sequence)` gives the sum of all the elements in a sequence.

`max(sequence)` gives the maximal element in a sequence.

Indexing of Sequences

You can access position n of sequence s with `s[n]`.

- Counting is `0`-based, so the first position is at index `0`, and the last position is at index len(s) - 1.
- Accessing elements this way takes the same amount of time regardless of the value of n.
- You can put any expression inside the square brackets that evaluates to an integer value.

Indexing can be used for some problems that are difficult otherwise:

```
def PrintReverse(s):
    for i in range(len(s)):
        print(s[len(s) - 1 - i])

def IsSorted(s):
    for i in range(len(s) - 1):
        if s[i + 1] > s[i]:
            return False
    return True

def AddLists(numbers1, numbers2):
    addlists = []

    for i in range(len(numbers1)):
        addlists += [numbers1[i] + numbers2[i]]

    return addlists

def Rotate(lst, n):
    rotate = []

    for i in range(n, n + len(lst)):
        rotate += lst[i % len(lst)]

    return rotate
```

Updating Values

Using indexing, you can update values at specific positions in a sequence:

```
s = 'Bond'

s[3] = 'o'
print(s)    => 'Bono'
```

Handouts & Assignments

- Handout 6 - Inspecting Lists (1)
- Handout 7 - Inspecting Lists (2)
- Handout 8 - List Comprehensions
- Handout 9 - Generating Lists
- Assignment 5 - Inspecting Sequences
- Assignment 6 - Inspecting and Generating Sequences
- Assignment 7 - Generating Sequences