Introduction:

- In Haskell, lists are a **homogenous data structure**. It stores several elements of the same type. That means that we can have a list of integers or a list of characters but we can't have a list that has a few integers and then a few characters.
- Lists are denoted by square brackets and the values in the lists are separated by commas.
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> print(a)
[1,2,3,4,5]
```

List Operations:

Note: I will use L1 and L2 to denote lists in the examples below, and e1 and e2 to denote elements.

1. **Adding to the beginning of a list:**

- To add to the beginning of a list, use "**:**". This is called **consing**. In fact, Haskell builds all lists this way by consing all elements to the empty list, []. The commas-and-brackets notation are just syntactic sugar. So [1,2,3,4,5] is exactly equivalent to 1:2:3:4:5:[].
- Syntax: **e1 : L1**
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> let b = 0:a
Prelude> print(b)
[0,1,2,3,4,5]
```

2. **Adding to the end of a list:**

- To add to the end of a list, use "**++**".
- Syntax: **L1 ++ [e1]**
- E.g.

```
Prelude> let c = b ++ [6]
Prelude> print(c)
[0,1,2,3,4,5,6]
```

3. **Joining 2 lists:**

- To join L2 to L1, do **L1 ++ L2**.
- E.g.

```
Prelude> let c = b ++ [6]
Prelude> print(c)
[0,1,2,3,4,5,6]
Prelude> let d = [7,8,9]
Prelude> let e = c ++ d
Prelude> print(e)
[0,1,2,3,4,5,6,7,8,9]
```

4. **Comparing Lists:**

    - Lists can be compared if the stuff they contain can be compared. When using **<**, **<=**, **>**, **>=**, **==** to compare lists, they are compared in lexicographical order.

- Syntax:
  L1 > L2
  L1 >= L2
  L1 < L2
  L1 <= L2
  L1 == L2
- E.g.

```
Prelude> [1,2,3] > [1,2,4]
False
Prelude> [1,2,3] >= [1,2,4]
False
Prelude> [1,2,3] < [1,2,4]
True
Prelude> [1,2,3] <= [1,2,4]
True
Prelude> [1,2,3] == [1,2,3]
True
```

5. **Head:**
   - head takes a list and returns its first element.
   - Syntax: **head L1**
   - E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> head a
1
```

6. **Last:**
   - last takes a list and returns its last element.
   - Syntax: **last L1**
   - E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> last a
5
```

7. **Init:**
   - init takes a list and returns everything except its last element.
   - Syntax: **init L1**
   - E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> init a
[1,2,3,4]
```

8. **Tail:**
   - tail takes a list and returns everything except for the first element.
   - Syntax: **tail L1**

- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> tail a
[2,3,4,5]
```

9. Length:

- length takes a list and returns its length.
- Syntax: **length L1**
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> length a
5
```

10. Reverse:

- reverse takes a list and returns its reverse.
- Syntax: reverse L1
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> reverse a
[5,4,3,2,1]
```

11. Take:

- take takes a number and a list. It extracts that many elements from the beginning of the list.
- Syntax: **take num L1**
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> take (-1) a
[]
Prelude> take 0 a
[]
Prelude> take 1 a
[1]
Prelude> take 2 a
[1,2]
Prelude> take 3 a
[1,2,3]
Prelude> take 4 a
[1,2,3,4]
Prelude> take 5 a
[1,2,3,4,5]
Prelude> take 6 a
[1,2,3,4,5]
Prelude> take 10000 a
[1,2,3,4,5]
```

12. Drop:

- drop takes a number and a list and it removes that many elements from the list.

- Syntax: **drop num L1**
- E.g.

```
Prelude> let a = [1,2,3,4,5]
Prelude> drop (-1) a
[1,2,3,4,5]
Prelude> drop 0 a
[1,2,3,4,5]
Prelude> drop 1 a
[2,3,4,5]
Prelude> drop 2 a
[3,4,5]
Prelude> drop 3 a
[4,5]
Prelude> drop 4 a
[5]
Prelude> drop 5 a
[]
Prelude> drop 100 a
[]
```

13. Range/Sequence:

- Denoted by "**..**"
- Syntax: **[starting_element .. end_element]**
- E.g.

```
Prelude> print([1..10])
[1,2,3,4,5,6,7,8,9,10]
Prelude> print(['a'..'z'])
"abcdefghijklmnopqrstuvwxyz"
Prelude> print(['A' .. 'Z'])
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

14. Other facts:

- To show that a list only has 0 elements, you can do this: [].
- To show that a list only has 1 element, you can do this: [a] or (a:[]).
- To show that a list has 1 or more elements, you can do this: (a:_). The "_" means 0 or more items.
- There's also a thing called **as patterns**. Those are a handy way of breaking something up according to a pattern and binding it to names whilst still keeping a reference to the whole thing. You do that by putting a name and an **@** in front of a pattern. For instance, the pattern **xs@(x:y:ys)**. This pattern will match exactly the same thing as **x:y:ys** but you can easily get the whole list via **xs** instead of repeating yourself by typing out **x:y:ys** in the function body again.

- E.g.

```
getListLength1 :: [Integer] -> Integer
getListLength1 [] = 0
getListLength1 (a:[]) = 1 -- Another way of saying there's only 1 element. Equivalent to [a].
getListLength1 (a:b:[]) = 2 -- Another way of saying there's only 2 elements. Equivalent to [a,b].
getListLength1 (a:b:c) = 2 + x
    where x = getListLength1 c
```

```
*Main> getListLength1 []
0
*Main> getListLength1 [1]
1
*Main> getListLength1 [1,2,3,4,5,6,7,8,9]
9
```

- E.g.

```
newGetListLength1 :: [Integer] -> Integer
newGetListLength1 [] = 0
newGetListLength1 (_:a) = 1 + newGetListLength1 a
```

```
*Main> newGetListLength1 []
0
*Main> newGetListLength1 [1]
1
*Main> newGetListLength1 [1,2,3,4,5]
5
```

- E.g.

```
getListLength2 :: [Integer] -> String
getListLength2 [] = "0 elements"
getListLength2 (a:[]) = "1 elements" -- Another way of saying there's only 1 element. Equivalent to [a].
getListLength2 (a:b:_) = "2 or more elements."
```

```
*Main> getListLength2 []
"0 elements"
*Main> getListLength2 [1]
"1 elements"
*Main> getListLength2 [1,2]
"2 or more elements."
*Main> getListLength2 [1,2,3]
"2 or more elements."
```

- E.g.

```
capital :: String -> String
capital "" = "Empty string, whoops!"
capital all@(x:xs) = "The first letter of " ++ all ++ " is " ++ [x]
```

```
*Main> capital "ABCD"
"The first letter of ABCD is A"
```