Lesson 5, Week 3:
(anonymous functions for interation)

AIM

— To explain and illustrate the use of anonymous functions in iterations

After this lesson, you will be able to

* Use the `->` syntax to write an anonymous function
* Use the `map` function to iterate an anonymous function over an array or a range
* Use the `filter` function to iterate an anonymous function over an array or a range

Anonymous functions in general

An anonymous function is simply a function without a name. The main purpose is to define a function *as briefly as possible* for local use, after which it is discarded. The functions we have so far seen, by contrast, all have names that stay in the namespace, so that the functions remain available.

We will discuss the operator syntax[1], which uses the operator `->`

For example, `x -> x^2`. This simply says "For input `x`, return `x^2`". The exact same function is created from `z -> z^2`

Note the absence of any type specification—this function will make string input into a string by repeating it, and numerical input into a number by squaring it. Type specification is however possible—see below

[1] There is a way to use the `function` keyword to make an anonymous function, but it is not part of this course.

Case 1: iterations using the `map` function

The function `map` iterates over an array or a range, but not a string.
[DEMO: `x -> x^2 for x in 1:4`, `x -> x^2 for x in ["abc", 2.0]`, `x -> x^2 for x in "abc"`]

To use `map` on the characters in a string, we have to extract them to an array first:
[DEMO: `y = [x for x in "abc"]; map(x -> x^3, y)`]

Here's a slightly more interesting example: `map(x -> x^2 + 2x + 1, -5:3)`

Case 2: iterations using the `filter` function

Whereas `map` applies a function to an array, `filter` applies a test, retaining the elements that satisfy it.

This is a good time to mention that the comparison operators also work on characters. For example, `'Z' < 'a'` is `true` and `'p' < '+'`.

Thus we have that `y = [x for x in "My A1 Jag is XJ6"]; filter(x -> x < 'a', y)` shortens the 20-character string to a 12-element array.

The relationship between `filter` and `map` can be illustrated by simply replacing the one with the other in this example. [DEMO]

It is always possible to go from `filter` to `map` in this way, but usually not the other way round.

`filter!` replaces the input array with the filtered result. [DEMO]

Case 3: comprehensions

This case is a almost trivial. Consider the comprehension `[string(x) for x in -5:3]`. Clearly, here we apply a function to the values in the range. [DEMO]

Now consider `[x^2 + 2x - 1 for x in -5:3]`. The result is identical to our earlier example `map(x -> x^2 + 2x + 1, -5:3)`. In fact, both of them evaluate the same formula over the same values, and return the same array.

Again we ask: why have `map` if we can do the same thing with a comprehension? And again the answer is: convenience. The `map` structure has possibilities that comprehensions do not have. We could achieve the same effect with a `for` loop, but the code would not be as compact nor as easy to read. Something similar is true for `filter`, although in that case there is the added capacity to

return an array smaller than the one you started with, a topic which is beyond the scope of this course.

Review and summary of Lesson

* The syntax `<variable> -> <expression>` is used for an anonymous function with input `<variable>` and function body `<expression>`
* The syntax `map(<function>, <iterable>)` returns an array of values made by calling `<function>` for every element of `<iterable>`
* The syntax `filter(<test function>, <iterable>)` returns an array of only those values in `<iterable>` for which `<test function>` returns `true`
* An array or a range is acceptable as the iterable in a call to `map` and to `filter`, but a string is not
* The function `filter!` is available to replace the array on which it is called, rather than creating a new one