Lesson 3, Week 3: Structures II
(`while ... end`)

AIM

— To describe and illustrate iteration with a `while` code block

After this lesson, you will be able to

* Motivate for (or if it is better, against) using a `while` block for a particular iteration
* Explain what kind of scope a `while` block has
* Explain how to write a `while` block, with particular reference to the stopping condition and the need for at least on global variable
* Use Ctrl-C to interrupt an infinite loop
* Use `push!` to extend an array
* Discuss the nesting of code blocks

Why use `while` code blocks?

As you will see below, a `while` code block repeats until some condition becomes false. Such iterations occur quite often with input/output streams, such as streaming music or a reading a file of text. They are also quite common in numerical work—such examples can be quite simple, which is while we use one below.

`while` code blocks have local scope

The keyword `while` creates a local scope which ends with the keyword `end` that is paired with it. Thus if any global variable is to change inside the `while` code block, it must be qualified with the `global` keyword. As you'll see very soon, this is crucially important.

`while` is a simple structure for repeated operations

Let's start with an example:

```
while loopvar < 4
    println("loopvar is now $loopvar")
    global loopvar = loopvar + 1
end
```

Initially, `loopvar < 4` is `true`, so the body of the `while` code block is executed repeatedly eventually causing `loopvar` to equal 4 at which point the `while` code block is ignored

The structure therefore is `while <test is true>` ... `end`. The logical expression `<test is true>` must return either `true` or `false`. The stopping condition is therefore that the result of the test is false, and when that happens the iteration ends. If the stopping condition is false when the keyword `while` is reached, the code skips over the whole `while` block without executing any of it.

It is important that the `<test>` depends on global variables, at least one of which must be change in the body of the code block, so that the result of the stopping condition test can change from `false` to `true` at least one `global` is required. If not, the next section shows what happens.

But perhaps `while` is *too* simple

First, make sure you can enter Ctrl-C on your keyboard. You will need it!

```
while true
    println("But it's true, I tell you!")
end
```

Hit Ctrl-C as soon as you can after the `end` there.

Here, the test is always `true`, so Julia enters an infinite loop, which will not end until you interrupt it with Ctrl-C or your computer does (by running out of battery, or being switched off, for example).

If you use `while` loops in your code, and we have suggested above that it is occasionally useful, be sure to remember that Ctrl-C will interrupt a Julia computation.

Using `push!` in a `while` block to make a list of numbers

Let us consider a list of numbers which grows quite fast: the Fibanacci numbers. They form the sequence 1, 1, 2, 3, 5, 8, 13, 21, ... The rule for continuing the sequence is that you determine the next number by adding the last two together[1].

We will make an array called `fibnumbers`. Initially it has the value `[1, 1]`. So we can calculate the next number with the formulat `fibnumbers[end-1] + fibnumbers[end]` and this will work for arrays with more of the Fibonacci numbers.

[1] So here the next number would be 13+21 which is 34.

There is a very useful built-in function `push!`, where the exclamation mark indicates that this function modifies its argument. Let's look up the help with `?`. We see that the line `push!(fibnumbers, fibnumbers[end-1] + fibnumbers[end])` changes the value from `[1, 1]` to `[1, 1, 2]` and repeated entering puts more and more Fibonacci numbers in the array.

Now suppose we want a list of these numbers up to at least 1000. We generate them with a `while` code block:

```
fibnumbers = [1, 1]
while fibnumbers[end] < 10000
    push!(fibnumbers, fibnumbers[end-1] + fibnumbers[end])
end
println(fibnumbers)
```

Nested code blocks

What does it mean to say two lines form a `while ... end` pair? Well, if a keyword such as `if` occurs after `while`, it must have its `end` before the end of the `while` code block. That is, the whole of the `if` block must be inside the `while` block. Putting `...` for all except the most essential parts, we see this implies the following:

```
while ...
    ...
    if ...
        ...
    else ...
        ...
    end
    ...
end
```

That is, the `if` block must be nested inside the `while` block[2]. Another way of saying this is note the the first `end` in this example closes the `if` block and cannot do otherwise. That is, partial nesting is not possible[3].

You can also see that the nesting can be reversed, and that one can nest `while` blocks inside other `while` blocks and `if` blocks inside `if` blocks and other blocks inside those and so on, many layers deep. Most computer languages have a limit in how deep the nesting can go, but that does not matter on this course, as we will not be nesting code. In fact, I recommend that you avoid deep nesting when writing code.

[2]Of course, the `if` block need not start inside the `while` block, but then they're simply two separate blocks and there is not chance of trouble.

[3]This comes directly from formal logic, by the way. The incomplete logical expressions of everyday speech are a nightmare for formal logic!

Review and summary of Lesson

* `while` block is useful when the number of steps in an interation is not known, but its stopping condition is known
* A `while` block has local scope
* The syntax is `while <test is true> ...end` where the `...` indicate the code in the body of the code block
* At least one variable in a `while` block must be made `global` and it must be used in the stopping condition
* Use Ctrl-C to interrupt a `while` iteration that will never reach its stopping condition.
* The function `push!` adds one or more elements to the end of an array, and also illustrates the Julian naming convention of input-modifying functions
* Code blocks can be nested, but they cannot otherwise overlap
* You should avoid writing deeply nested code