Lesson 1, Week 3:
(comparison operators, logical operators)

AIM

—

After this lesson, you will be able to

* Name and type the five comparison operators used on this course
* Name and type the three logical operators used on this course
* Discuss which kinds of values can be compared with the comparison operators
* Write long logical tests which combine names with comparison as well as logical operators

Looking back, looking forward

This week, we study things that are often done much earlier in a programming course. But remember our slogan "Small steps, leave nothing out, everything makes sense". What you have learnt so far is about creating and modifying values, often values that are stored by binding them to the names of variables.

The code that created these values consisted of a sequence of valid expressions, carried out in the order they appear in the code. We will call such a sequence a *code path*. Let's recap what you've learnt about creating code paths. We'll use as headings the four components of valid Julia code: values, names, operators and delimiters.

Values String values (of type `String`) which consist of characters (of type `Char`), number values (types `Int64` and `Float64`) and the logical constants `true` and `false` (the possible values of type `Bool`).

Names These include the valid variable names, which are also valid names for functions, as well as the reserved keywords which cannot be used for names[1] and the names in Julia's type system.

[1]So far you've seen that the reserved keywords include `false`, `true`, `for`, `function`, `return`, `end`, `global` and `local`. We'll do just a few more, so about half of the reserved keywords in Julia are not part of this course.

Operators So far, you've seen the assignment operator `=`, the string operators `*` and `^`, the number operators `+`, `-`, `*`, `\` and `^`, the `in` operator used to make comprehensions, and the type operator `::`.

Delimiters You've seen `" "` (for strings), `' '` (for characters), `[ ]` (for arrays and for indexing into arrays and strings), `( )` (for a function's argument, and also to specify the order of arithmetic operations), and `,` (as separators for the elements of an array and for the variables in an argument list).

What you haven't seen much of is the applied formal logic that I spoke about such a lot in week 1. It is time to remedy that. Recall that formal logic is based on the two logical constants `true` and `false`[2]. In what follows, we look first at comparison operators, which generate those values, and then at the logical operators `!`, `&&` and `||`.

This will enable you create logical tests, which Julia uses to create branching code[3], in which the calculation could proceed in one of two ways, and the way chosen depends on whether the result of a logical test is true or false.

Comparison operators

In this course, we use the following comparson operators[4]:

Symbol	Result
`==`	equality test
`>`	true if left hand side greater than right
`<`	true if left hand side smaller than right
`>=`	ditto for greater than or equal to
`<=`	ditto for less than or equal to

These are, of course, taken from standard mathematics, with the exception of the equality test. In Julia, as in all computer languaes, a strict separation is made between assignment and equality testing. Since assignment already uses a single equals sign, a different symbol is needed for equality testing, and Julia's designers chose the double equals sign.

All of these operators are written as they would be in maths: between a left hand side value and a right side value. The result of such a comparison is always one of the constants `true` or `false`. Let's see this in action.

[DEMO: examples of comparison tests using number, character and string values.]

[2]It is true that some formal logic use a third and even a fourth constant, and to a very limited extent that also happens in Julia, but on this course we avoid that possibility.
[3]And so do all other programming languagges.
[4]Julia recognises many more.

Logical operators

In week 1, we used truth tables to discuss the operators NOT, AND, OR. Let's now see how they work in Julia:

[DEMO: illustrate NOT, it is `!` via `!true` and `!false`]

[illustrate AND `&&` via `true && true`, `false && true`, `true && false` and `false && false`]

[illustrate OR `||` via `true ||true`, `false || true`, `true || false` and `false || false`]

Let's also illustrate the use of `( )` to make logical expressions unambiguous.

DEMO: `!true && false`

`!(true && false)`

In logical expressions, even more so than in arithmetical expressions, use brackets to make sure the expression is not ambigous.

Most often, we use the logical and comparison operators together, to make *logical tests*. These are expressions like $a > b$ OR $a < -b$, $x < 4$ AND $x > 0$, and NOT $x = 4$ (note the ambiguity with the equals sign here). Let's illustrate them all and more with a [DEMO:]

The truth values of these logical tests depend, of course, on the numerical values of the variables in them. When reading code, it is often necessary to answer questions like: for which values of `a` is the expression `a > -2 || a < 2` equal to `true` ?

Review and summary of Lesson

* The comparison operators on this course are `==`, `>`, `<`, `>=` and `<=`
* Using a comparison operator results in one of two values: `true` and `false`
* One can compare values of like type: numbers with numbers[5], characters with characters, and strings with strings
* The logical operators on this course are `!` for NOT, `&&` for AND and `||` for OR
* Logical tests often consist of names, comparison operators, logical operators and delimiters, and can become long[6]

[5]Note that this allows comparison across number types.
[6]Even on this course—see Week 4.

Lesson 2, Week 3: Structures I (`if ... elseif ... else ... end`)

AIM

— To learn to use `if` blocks to write code branches

After this lesson, you will be able to

* Use a simple `if` block to branch between code that executes or not depending on a logical test
* Use `elseif` and `else` blocks inside an `if` block to choose among several code branches

The reserved keyword `if`

Like `function`, the keyword `if` opens a code block which must be closed with the keyword `end`.

However, they differ as to scope: the scope of an `if ... end` block is global; the keyword `if` does not create a local scope.

`if` must be followed by a truth value—normally, the result of a logical test. The code body of the `if` block consists of the lines after the logical test and before the `end`.

An `if` block may contain only one code path. In that case, `if true ... end` will ensure that the code in the code path is executed, and `if false ... end` will simply do nothing. This is the simplest kind of branch you can introduce in your code.

Simple examples

Let's start with `if (a > 1) println("a is greater than 1") end` The code body is a single line, so it is convenient to write the entire `if` block on on line. We need to assign a value to `a` before the code will run, of course. We can also play with the actual code body[1], but we'll resist doing too much.

[1]It is quite tempting to add lots of lines ...

If we wrap the :

```
function ifeg1(a)
    if (a > 1) println("a is greater than 1") end
end
```

DEMO: try a few values for `a` and in a comprehension

The comprehension `[ifeg1(x) for x in [0, 1, 2, 3]]` shows on screen the two lines that correspond to calling `println` twice—and the value `nothing` four times! We can clarify things with the lines

```
x=[(if x > 1 println("yes") end) for x in [2,3,51] ];
x
```

which separates out the creation of `x` and its display. [DEMO]

The value `nothing` goes into `x` comes from `println`—it formats its input and prints it on the screen, but it does not create any other value that can be used to make an element of `x`. If we add something that does, for example some value like a number or a string or a logical constant, that makes things even clearer. Compare

`y=[(if x > 1 println("yes"); true end) for x in [2,3,51] ];` to

`z=[(if x > 1 true; println("yes") end) for x in [2,3,51] ];`

Choosing between two code blocks

In the previous example, the `if` block chose between doing something or doing nothing. Often you want to execute one of two code blocks. The keyword `else` allows you to do that:

```
function ifeg2(a, b)
     if a > b
         println("a is bigger than b")
     else
         println("a is not bigger than b")
     end
end
```

DEMO:

Note that code block starting with `else` is the if-all-else-fails option: it runs whenever the logical test is results in `false`. This will become clearer in the next section.

Choosing among three or more options

Let's write a function that specifies Goldilocks' dialogue:

```
function goldisays(porrdegreesC)
   if porrdegreesC > 55
      println("The porridge is too hot!")
   elseif porrdegreesC < 45
      println("The porridge is too cold!")
   else
      println("Hmm!  The porridge is just right.")
   end
end
```

DEMO:

You should be aware that there are usually lots of logical expressions that will give the required results. For example, we could replace the code body of the function with

```
if ((porrdegreesC > 45) && (porrdegreesC < 55)) println("Hmmm!  Just right!")
elseif (porrdegreesC < 45) println("Ugh!!  Too cold!")
else println("Ouch!  Too hot!")
end
```

This also shows that the whole `if` block is just one statement, as in
`if true print("true") else print("false") end` and
`if false print("first true") elseif false print("second true") else print("none true") end`

A word of advice

If you read about coding style, you are likely to get warnings about "elseif ladders" and how they're bad style. Such ladders can happen when your logical tests must pick one of many options (that is, quite a few more than three). You test for one, then you test for the next, and so on, until all of the options are tested. Every test is an `elseif` rung on the ladder.

That kind of coding problem is undoubtedly hard. The logic for choosing where and what to eat when you go out may mean choosing among several eateries, and several menu options, and how you travel depends on where you go and what (or indeed whether) you have dessert depends on what you had before, and so on. It can be a long job to code, a very hard job to test properly, and a nightmare for anybody else to read[2].

My advice is, don't worry about style. Get your `if` blocks to work properly with 4 or 6 options, no matter how ugly they are. Always use only one `if` block for choosing one among many options. Avoid especially trying to use a sequence of separate `if` blocks, you can never be sure that only one of them will be executed. Once you have mastered getting it right you can start working on making it look good and making it readable.

[2]Including you yourself a months or sometimes a few days later.

Review and summary of Lesson

* An `if ...end` code block has global scope
* Syntax is `if (<logical test> ... elseif (<logical test>) ... else ...end)`
* Each logical test is an expression that evaluates either to `true` or `false`
* You may repeat `elseif` code blocks inside the `if` block
* Beginners shouldn't worry about the style of their `if`, but should instead focus on making sure the (formal) logic is correct

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

Lesson 4, Week 3: Structures III (`for ... end`)

AIM

— To learn to describe and use `for` loops

After this lesson, you will be able to

* Use a `for` code block to iterate over an iterable container
* Discuss the the local scope of a `for` with particular reference to the loop variable

`for` blocks have local scope

A `for` block introduces local scope: global variables can be read inside the block, but if their values are to change, they must be qualified with the keyword `global`. However, remember that you can replace a value in an array with another value of the same type[1].

Syntax of a `for` block by example

Our example is

```
for loopvar in 1:4
    println(('α' + loopvar)^loopvar)
end
```

The `println` function is called once for each value in the range.
The iteration requires the reserved keyword `for`, the operator `in` and an iterable value[a]

[a]In this case, the range `1:4`

You will notice that this is very similar to the comprehension `[('α'+ loc1)^loc1 for loc1 in 1:4]`, which creates exactly the same string values. This is because comprehension can be understood as a specialised kind of `for` loop: it uses the same keywords (except that a comprehension doesn't use `end`), it always produces an array, and it is one line of code only.

[1]For completeness: actually, any value that can be converted to the correct type will do, but we do not pursue that on this course.

Of course, the iterable supplying the values does not need to be a range. On this course, the iterable containers are strings, arrays and ranges, but Julia has many more. [DEMO]

You can change the value of the loop variable inside the loop, but this does not affect what value it takes on the next pass through the loop body. After all, it is local to the loop. But note that if you try to declare it global, Julia throws an error. [DEMO]

Pros and cons of `for` blocks vs `while` blocks vs comprehensions

Anything one does with a comprehension can be done with a `for` loop, anything that is done with a `for` loop can also be done with a `while` loop.

Question: if everything can be done with `while` code blocks, why the other two structures?

Answer: human convenience. High-level languages like Julia have one purpose only: to make things easier for those humans (like us!) who want to tell computers what we want them to do. You've seen that `while` loops can be infinite; they also take more lines to code than a `for` loop. For those two reasons, a `for` loop is better[2] if it can be used. Comprehension on the other hand is not necessarily better: it can for instance be harder to read when coming upon a piece of Julia code for the first time. The main reason for using a comprehension is that it takes up only one line, which can make code more compact. Overall, code is easier to read when it takes up less screen space.

Contrasting the code structures on this course

The structures we have seen are comprehensions, functions (two ways: with the keyword `function` and inline), logical tests using `if`, and the loops introduced by `while` and `for`.

- `if` blocks have global scope, the others have local scope.
- Except for comprehensions and inline functions, the structures start and end with a reserved keyword.
- Structures can be nested, but this possibility should not be over-used

Review and summary of Lesson

* `for` blocks have local scope
* The syntax is `for <loop variable name> in <iterable value> ...end`, where the `...` stands for the body of the loop
* The the loop variable takes the values in the iterable one by one and for each such value the body of the loop is executed
* The loop variable cannot be made visible globally

[2]From the human point of view, not the computer's, of course.

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