Lesson 2, Week 2: Number Types

AIM

— To describe the main number types of Julia

After this lesson, you will be able to

* Motivate why we do number types very early in this course
* Describe the types `Int64` and `Float64`
* Use `bitstring` to display the bits pattern of a number
* Show that a floating point number equal to an integer has a different bits pattern
* Briefly describe integer and floating point types that use fewer than 64 bits per number

Why mention number types so early in the course?

Reason 1: many error messages mention number types.
Efficient debugging relies on understanding them well enough to decide whether the problem really is with the type a numerical value has.
[DEMO: indexing with the value `1.0`; the decimal point makes all the difference!]

Reason 2: they make a good introduction to the Julia type system.

The `Int64` type

You've seen that a character has variable width of one or more code units. By contrast, number values of a given type have the same width. It is measured in bits[1] and an `Int64` is an integer value that occupies 64 bits.

If you simply enter an integer you get a 64-bit value.

DEMO: `[1, 2]`, `bitstring(1)`
We explain the function `bitstring` in more detail after the examples.

[1] A bit has the value 0 or the value 1.

Let's show the raw bits of a few more 64-bit integers.

[DEMO: 0, 7 and 13. And -0, -7 and -13]

`Int64` represents all integers from -9223372036854775808 to 9223372036854775807 but none outside that range[2].

The function `bitstring` returns, if possible, the bits representation of its argument. Note that this includes character values[3].

[DEMO: explain the result of `[bitstring(x) for x in "aα±"]`.]

Again, we emphasise that these examples rely crucially on the absence of decimal points, because that is how Julia knows to use the `Int64` type.

Floating point numbers: `Float64`

These numbers are very different from the signed integers above: we use them to approximate all numbers in a range, whether whole numbers, rational numbers, or irrational numbers. They can go much larger than `Int64` values, and they can represent fractions as well as integers[4]. Note the presence of the decimal in all the floating point values we create below.

[DEMO bitstrings of 1.0, 0.1, 1.1 and their negative counterparts]

As you can see, `Float64` also has a width of 64 bits, but uses a much more complicated arrangement of bits than one sees in `Int64`. The details of this difference do not matter on this course, only the implication at machine level. Without going into electronic detail, it is reasonably obvious that the actual manipulation of bits for adding two `Int64` numbers will be very different from adding two `Float64` numbers. Moreover, it is simply not possible to add a `Float64` to a `Int64`, as the result must be one of the two, it can't be a hybrid.

[DEMO: arithmetic with mixed types]

We see that in all cases, Julia gives the result with type `Float64`. This is because all `Int64` values can be fairly accurately represented by `Float64` values, but not the other way round.

Besides representing quite a large range of numbers, `Float64` helps in other ways. The biggest actual positive number it can represent is near $1.7 * 10^{308}$, much larger than the largest possible `Int64`. Something perhaps unexpected happens for numbers that go bigger:

[DEMO: `1.7*(10.0^308)`, `1.8*(10.0^308)`, `(1.7*(10.0^308)) * 2` — note ALL decimal points!]

The value `Inf` is mathematically very interesting, but we don't pursue it. A similarly interesting

[2] An easier way to think of this interval is that it is approximately $[-9 * 10^{18}, 9 * 10^{18}]$

[3] Note that the bitstring of every character contains 32 bits. Does this mean fixed width? No, not when characters combine to form a string. This is because the bitstring of a character contains information about how many code units it contains. Everything outside these code units are zeros anyway and are discarded when the characters are joined to form a string.

[4] Not all of them exactly, though

value is `NaN`, which represents the indeterminate result of attempted calculations like `0.0/0.0` and `Inf * 0.0` . These values will occasionally crop up not only in your results but also in your error messages.

There is a very convenient shorthand for for numbers of the form `1.7*(10.0^308)`: we can simply write `1.7e308`. The error message for `1.8e308` is new, and note that the number following the `e` need not be an integer[5].

Number types with reduced width: faster computation but less accurate

On most modern systems, numbers by default use 64 bits, and so if you enter an integer it is by default of type `Int64` and a float by default is of type `Int64`. Using so many bits makes them very powerful in many ways, but it can be more efficient to use shorter bitstrings. Julia makes available types like `Int8` and `Float32`, which use 8 and 32 bits, respectively. The tradeoff works as follows: using 8 bits instead of 64 means you need only 1/8th of the capacity. However, and this is the reason modern systems have gone on to 64 bit defaults, you have many fewer numbers available. For example, the type `Int8` consists of the integers from -128 to 127, and no others.

To make sure a number is of a given type, you can call the type as if it is a function: `Int32(13)`, `Int16(13)`, `Int8(13)`.
But you won't see any difference until you use them, `bitstring` being a reliable indicator:

[DEMO of this fact and a few more illustrations]

Other number types

Julia has a great many more number types, including complex and exact rational numbers. It is not necessary for a beginner to master them.

Review and summary

* Julia's main number types are `Int64` integers and `Float64` floating point numbers
* The `Int64` integers range from approximately $-9 * 10^{18}$ to $9 * 10^{18}$
* The numerical values of `Float64` range from approximately $-1.7 * 10^{308}$ to $1.7 * 10^{308}$
* `Float64` numbers can approximate fractions
* Arithmetic that uses both `Float64` and `Int64`numbers always result in `Float64` number
* `Float64` values include `Inf` and `NaN`, which do not represent actual numbers

[5] And it can be negative, of course.