2c. Numbers

Martin Alfaro
PhD in Economics

INTRODUCTION

The previous section introduced the concept of variables, distinguishing between those containing a single element (scalars) and collections. This section expands on scalars, exclusively focusing on those holding numeric values.

NUMBERS

Computers store numbers in various formats, treating integers and decimal numbers as separate entities. Even within each category of numbers, multiple representations emerge depending on the intended level of precision. This precision is determined by the number of bits allocated to store values in memory, which in turn defines the maximum range of values that a data type supports. [1] The representation just described extends well beyond Julia, and is intrinsic to how computers operate at a fundamental level.

In modern computers, numbers typically have a default size of 64 bits, and Julia's default types for numbers are:

- `Int64` for integers.
- `Float64` for decimal numbers. [2]

> **Remark**
> Julia provides the type `Int` as a more versatile option than `Int64`, which adapts to your computer's architecture: `Int` defaults to `Int64` on 64-bit systems and `Int32` on 32-bit systems. Since most modern machines operate on a 64-bit architecture, `Int` typically defaults to `Int64`. Note that there's no equivalent type `Float` for floating-point numbers, with Julia always defaulting to `Float64`.

It's worth emphasizing that `Int64` and `Float64` are two different data types. Thus, while `1` is a value with type `Int64`, the same value becomes `1.0` as a `Float64` type.

NUMBERS

```
x = 1       # `Int64`

y = 1.0     # `Float64`
z = 1.      # alternative notation for `1.0`
```

> **Remark**
>
> To enhance code readability, you can break up long numbers by inserting underscores `_`.

NOTATION FOR NUMBERS

```
x = 1000000
y = 1_000_000             # equivalent to `x` and more readable

x = 1000000.24
y = 1_000_000.24              # '_' can be used with decimal
numbers
```

The type `Float64` encompasses not only decimal numbers, but also two special values: `Inf` for infinity and `NaN` for indeterminate expressions such as 0/0 (`NaN` stands for "not a number"). Considering this, all the following variables have type `Float64`.

FLOAT64

```
x = 2.5

y = 10/0

z = 0/0
```

```
julia> x
2.5
julia> y
Inf
julia> z
NaN
```

BOOLEAN VARIABLES

A distinct numeric type is `Bool`, which facilitates the representation of **Boolean variables**. These variables can only take on the values `true` and `false`. Internally, they're implemented as integers, with `true` corresponding to `1` and `false` to `0`. Because of this implementation, Julia accepts `1` and `0` interchangeably with `true` and `false`.

Boolean expressions come into play when evaluating conditions, such as checking whether a number exceeds a certain value or whether a string matches a specific pattern. These conditional evaluations yield Boolean values, and can then be employed to control the flow of the program. Some examples of Boolean values are presented below.

NOTATION FOR BOOLEAN

```
x = 2
y = 1

z = (x > y)       # is `x` greater than `y` ?
z = x > y         # same operation (don't interpreted it as 'z = x')
```

```
julia> z
true
```

ARITHMETIC OPERATORS

Numbers can be manipulated through a variety of **arithmetic operators**. These operators are represented by symbols akin to those in other programming languages.

Julia's Arithmetic Operator	Meaning
`x + y`	addition
`x - y`	subtraction
`x * y`	product
`x / y`	division
`x^y`	power (x^y)

It's worth noting that all the operators presented above are *binary*. Consequently, they follow the syntax `x <symbol> y`, as indicated in our **discussion** about the syntax of operators.

FOOTNOTES

[1.] For instance, 8-bit integers can only represent values from -128 to 127. Likewise, 32-bit floating-point numbers, used for decimal numbers, can represent up to 7 significant digits of precision.

[2.] The term "Float" stands for "floating point" and is how computers represent decimal numbers.