Learn Julia in Y minutes (1)

Get the code: learnjulia.jl

Primitive Datatypes and Operators

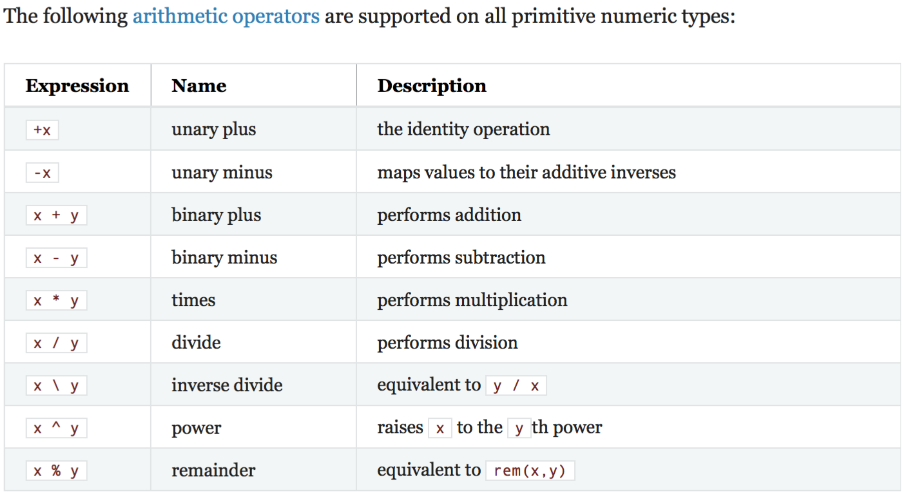
There are several basic types of numbers.

```
3; # => 3 (Int64)
3.2; # => 3.2 (Float64)
2 + 1im; # => 2 + 1im (Complex{Int64})
2//3; # => 2//3 (Rational{Int64})
```

All of the normal infix operators are available.

```
1 + 1; # => 2
8 - 1; # => 7
10 * 2; # => 20
35 / 5; # => 7.0
5 / 2; # => 2.5 # dividing an Int by an Int always results in
div(5, 2); # => 2 # for a truncated result, use div
5 \ 35; # => 7.0
2 ^ 2; # => 4 # power, not bitwise xor
12 % 10; # => 2
```

Arithmetic Operators



The following arithmetic operators are supported on all primitive numeric types:

Expression	Name	Description
`+x`	unary plus	the identity operation
`-x`	unary minus	maps values to their additive inverses
`x + y`	binary plus	performs addition
`x - y`	binary minus	performs subtraction
`x * y`	times	performs multiplication
`x / y`	divide	performs division
`x \ y`	inverse divide	equivalent to `y / x`
`x ^ y`	power	raises `x` to the `y` th power
`x % y`	remainder	equivalent to `rem(x,y)`

Enforce precedence with parentheses

```
(1 + 3) * 2; # => 8
```

Bitwise Operators

```
~2; # => -3   # bitwise not
3 & 5; # => 1 # bitwise and
2 | 4; # => 6 # bitwise or
2 $ 4; # => 6 # bitwise xor
2 >>> 1; # => 1 # logical shift right
2 >> 1 ; # => 1 # arithmetic shift right
2 << 1 ; # => 4 # logical/arithmetic shift left
```

Boolean values are primitives

```
true
false
```

Boolean operators

```
!true; # => false
!false; # => true
1 == 1; # => true
2 == 1; # => false
1 != 1; # => false
2 != 1; # => true
1 < 10; # => true
1 > 10; # => false
2 <= 2; # => true
2 >= 2; # => true
```

Comparisons can be chained

```
1 < 2 < 3; # => true
2 < 3 < 2; # => false
```

Strings are created with "

```
"This is a string."
```

Julia has several types of strings, including ASCIIString and UTF8String. More on this in the Types section.

Character literals are written with '

```
'a'
```

Some strings can be indexed like an array of characters

```
"This is a string"[1]; # => 'T' # Julia indexes from 1
```

However, this is will not work well for UTF8 strings,

so iterating over strings is recommended (map, for loops, etc).

$ can be used for string interpolation:

```
"2 + 2 = $(2 + 2)"; # => "2 + 2 = 4"
```

You can put any Julia expression inside the parentheses.

Another way to format strings is the printf macro.

```
@printf "%d is less than %f" 4.5 5.3 # 5 is less than 5.30000
```

```
5 is less than 5.300000
```

Printing is easy

```
println("I'm Julia. Nice to meet you!")
```

```
I'm Julia. Nice to meet you!
```

String can be compared lexicographically

```
"good" > "bye"; # => true
"good" == "good"; # => true
"1 + 2 = 3" == "1 + 2 = $(1+2)"; # => true
```

Variables and Collections

You don’t declare variables before assigning to them.

```
some_var = 5 # => 5
some_var # => 5
```

Accessing a previously unassigned variable is an error

```
try
    some_other_var # => ERROR: some_other_var not defined
catch e
    println(e)
end
```

```
UndefVarError(:some_other_var)
```

Variable names start with a letter or underscore.

After that, you can use letters, digits, underscores, and exclamation points.

```
SomeOtherVar123! = 6 # => 6
```

You can also use certain unicode characters

```
\0x2603 = 8 # => 8
```

These are especially handy for mathematical notation

```
2 * pi # => 6.283185307179586
```

A note on naming conventions in Julia:

- Word separation can be indicated by underscores ('_'), but use of underscores is discouraged unless the name would be hard to read otherwise.

A note on naming conventions in Julia:

- Word separation can be indicated by underscores ('_'), but use of underscores is discouraged unless the name would be hard to read otherwise.
- Names of Types begin with a capital letter and word separation is shown with CamelCase instead of underscores.

A note on naming conventions in Julia:

- Word separation can be indicated by underscores ('_'), but use of underscores is discouraged unless the name would be hard to read otherwise.
- Names of Types begin with a capital letter and word separation is shown with CamelCase instead of underscores.
- Names of functions and macros are in lower case, without underscores.

A note on naming conventions in Julia:

- Word separation can be indicated by underscores ('_'), but use of underscores is discouraged unless the name would be hard to read otherwise.
- Names of Types begin with a capital letter and word separation is shown with CamelCase instead of underscores.
- Names of functions and macros are in lower case, without underscores.
- Functions that modify their inputs have names that end in !. These functions are sometimes called mutating functions or in-place functions.

Arrays store a sequence of values indexed by integers 1 through n:

```
julia> a = Int64[]
0-element Array{Int64,1}
```

1-dimensional array literals can be written with comma-separated values.

```
julia> b = [4, 5, 6]

3-element Array{Int64,1}:
 4
 5
 6
julia> b = [4; 5; 6]

3-element Array{Int64,1}:
 4
 5
 6
julia> b[1]

4
```

2-dimensional arrays use space-separated values and semicolon-separated rows.

```
julia> matrix = [1 2; 3 4]
2x2 Array{Int64,2}:
 1  2
 3  4
```

Arrays of a particular Type

```
julia> b = Int8[4, 5, 6]
3-element Array{Int8,1}:
 4
 5
 6
```

Add stuff to the end of a list with push! and append!

```
a = []
push!(a,1)      # => [1]
push!(a,2)      # => [1,2]
push!(a,4)      # => [1,2,4]
push!(a,3)      # => [1,2,4,3]
append!(a,b)    # => [1,2,4,3,4,5,6]
```

Remove from the end with pop

```
pop!(b)          # => 6 and b is now [4,5]
```

Let's put it back

```
push!(b,6)   # b is now [4,5,6] again.
a[1] # => 1 # remember that Julia indexes from 1, not 0!
```

end is a shorthand for the last index. It can be used in any indexing expression

```
a[end] # => 6
```

we also have shift and unshift

```
shift!(a) # => 1 and a is now [2,4,3,4,5,6]
unshift!(a,7) # => [7,2,4,3,4,5,6]
```

Function names that end in exclamations points

indicate that they modify their argument

```
arr = [5,4,6] # => 3-element Int64 Array: [5,4,6]
sort(arr) # => [4,5,6]; arr is still [5,4,6]
sort!(arr) # => [4,5,6]; arr is now [4,5,6]
```

Looking out of bounds is a BoundsError

```
try
    a[0] # => ERROR: BoundsError() in getindex at array.jl:27
    a[end+1] # => ERROR: BoundsError() in getindex at array.j
catch e
    println(e)
end

BoundsError(Int64[],(0,))
```

Errors list the line and file

they came from, even if it's in the standard library. If you built Julia from source, you can look in the folder base inside the julia folder to find these files.

You can initialize arrays from ranges

```
a = [1:5;] # => 5-element Int64 Array: [1,2,3,4,5]
```

You can look at ranges with slice syntax.

```
a[1:3] # => [1, 2, 3]
a[2:end] # => [2, 3, 4, 5]
```

Remove elements from an array by index with splice!

```
arr = [3,4,5]
splice!(arr,2) # => 4 ; arr is now [3,5]
```

Concatenate lists with append!

```
b = [1,2,3]
append!(a,b) # Now a is [1, 2, 3, 4, 5, 1, 2, 3]
```

Check for existence in a list with in

```
in(1, a) # => true
```

Examine the length with `length`

```
length(a) # => 8
```