Learn Julia in Y minutes

Get the code: learnjulia.jl

Primitive Datatypes and Operators

There are several basic types of numbers.

```julia
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

```julia
(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

```julia
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

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

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

String can be compared lexicographically

```julia
"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.

```julia
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:270
    a[end+1] # => ERROR: BoundsError() in getindex at array.jl
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
```

Tuples are immutable.

```
tup = (1, 2, 3) # => (1,2,3) # an (Int64,Int64,Int64) tuple.
tup[1] # => 1
try:
    tup[1] = 3 # => ERROR: no method setindex!((Int64,Int64,In
catch e
    println(e)
end
```

```
MethodError(setindex!,(:tup,3,1))
```

Many list functions also work on tuples

```
length(tup) # => 3
tup[1:2] # => (1,2)
in(2, tup) # => true
```

You can unpack tuples into variables

```
a, b, c = (1, 2, 3) # => (1,2,3)  # a is now 1, b is now 2 anc
```

Tuples are created even if you leave out the parentheses

```julia
d, e, f = 4, 5, 6 # => (4,5,6)
```

A 1-element tuple is distinct from the value it contains

```julia
(1,) == 1 # => false
(1) == 1 # => true
```

Look how easy it is to swap two values

```julia
e, d = d, e  # => (5,4) # d is now 5 and e is now 4
```

Dictionaries store mappings

```julia
empty_dict = Dict() # => Dict{Any,Any}()
```

You can create a dictionary using a literal

```
filled_dict = Dict("one"=> 1, "two"=> 2, "three"=> 3)
# => Dict{ASCIIString,Int64}
```

Look up values with []

```julia
filled_dict["one"] # => 1
```

Get all keys

```
keys(filled_dict)
# => KeyIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>
```

Note

dictionary keys are not sorted or in the order you inserted them.

Get all values

```
values(filled_dict)
# => ValueIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=
```

Note - Same as above regarding key ordering.

Check for existence of keys in a dictionary with in, haskey

```
in(("one" => 1), filled_dict) # => true
in(("two" => 3), filled_dict) # => false
haskey(filled_dict, "one") # => true
haskey(filled_dict, 1) # => false
```

Trying to look up a non-existent key will raise an error

```
try
    filled_dict["four"] # => ERROR: key not found: four in ge
catch e
    println(e)
end
```

```
UndefVarError(:filled_dict)
```

Use the get method

to avoid that error by providing a default value

```
get(dictionary,key,default_value)
get(filled_dict,"one",4) # => 1
get(filled_dict,"four",4) # => 4
```

Use Sets to represent collections of unordered, unique values

```
empty_set = Set() # => Set{Any}()
```

Initialize a set with values

```
filled_set = Set([1,2,2,3,4]) # => Set{Int64}(1,2,3,4)
```

Add more values to a set

```
push!(filled_set,5) # => Set{Int64}(5,4,2,3,1)
```

Check if the values are in the set

```
in(2, filled_set) # => true
in(10, filled_set) # => false
```

There are functions for set intersection, union, and difference.

```
other_set = Set([3, 4, 5, 6]) # => Set{Int64}(6,4,5,3)
intersect(filled_set, other_set) # => Set{Int64}(3,4,5)
union(filled_set, other_set) # => Set{Int64}(1,2,3,4,5,6)
setdiff(Set([1,2,3,4]),Set([2,3,5])) # => Set{Int64}(1,4)
```

Control Flow

Let's make a variable

```
some_var = 5
```

Here is an if statement. Indentation is not meaningful in Julia.

```julia
if some_var > 10
    println("some_var is totally bigger than 10.")
elseif some_var < 10    # This elseif clause is optional.
    println("some_var is smaller than 10.")
else                    # The else clause is optional too.
    println("some_var is indeed 10.")
end
# => prints "some var is smaller than 10"
```

For loops iterate over iterables.

Iterable types include `Range, Array, Set, Dict`, and `AbstractString`.

```
julia> for animal=["dog", "cat", "mouse"]
    println("$animal is a mammal")
    # You can use $ to interpolate variables or expression int
enddog is a mammal
cat is a mammal
mouse is a mammal
```

You can use ‘in’ instead of ‘=’.

```
julia> for animal in ["dog", "cat", "mouse"]
    println("$animal is a mammal")
enddog is a mammal
cat is a mammal
mouse is a mammal
```

Example

```
julia> for a in Dict("dog"=>"mammal","cat"=>"mammal","mouse"=>
    println("$(a[1]) is a $(a[2])")
endmouse is a mammal
cat is a mammal
dog is a mammal
```

Example

```
julia> for (k,v) in Dict("dog"=>"mammal","cat"=>"mammal","mous
    println("$k is a $v")
endmouse is a mammal
cat is a mammal
dog is a mammal
```

While loops loop while a condition is true

```
julia> x = 0

0
julia> while x < 4
    println(x)
    x += 1  # Shorthand for x = x + 1
end0
1
2
3
```

Handle exceptions with a try/catch block

```julia
try
   error("help")
catch e
   println("caught it $e")
end
# => caught it ErrorException("help")
```

Functions

The keyword ‘function’ creates new functions

```julia
function name(arglist)
  body...
end

function add(x, y)
    println("x is $x and y is $y")

    # Functions return the value of their last statement
    x + y
end

add(5, 6) # => 11 after printing out "x is 5 and y is 6"

x is 5 and y is 6
```

Compact assignment of functions

```
f_add(x, y) = x + y # => "f (generic function with 1 method)"
f_add(3, 4) # => 7
```

Function can also return multiple values as tuple

```julia
f(x, y) = x + y, x - y
f(3, 4) # => (7, -1)
```

You can define functions that take a variable number of positional arguments

```
function varargs(args...)
    return args
    # use the keyword return to return anywhere in the functi
end
# => varargs (generic function with 1 method)

varargs(1,2,3) # => (1,2,3)
```

Splat

- The ... is called a splat.

```
add([5,6]...) # this is equivalent to add(5,6)

x is 5 and y is 6


x = (5,6)     # => (5,6)
add(x...)     # this is equivalent to add(5,6)

x is 5 and y is 6
```

Splat

- The ... is called a splat.
- We just used it in a function definition.

```
add([5,6]...) # this is equivalent to add(5,6)

x is 5 and y is 6


x = (5,6)      # => (5,6)
add(x...)      # this is equivalent to add(5,6)

x is 5 and y is 6
```

Splat

- The ... is called a splat.
- We just used it in a function definition.
- It can also be used in a function call,

```
add([5,6]...) # this is equivalent to add(5,6)
```

```
x is 5 and y is 6
```

```
x = (5,6)     # => (5,6)
add(x...)     # this is equivalent to add(5,6)
```

```
x is 5 and y is 6
```

Splat

- The ... is called a splat.
- We just used it in a function definition.
- It can also be used in a function call,
- where it will splat an Array or Tuple's contents into the argument list.

```
add([5,6]...) # this is equivalent to add(5,6)
```

```
x is 5 and y is 6
```

```
x = (5,6)      # => (5,6)
add(x...)      # this is equivalent to add(5,6)
```

```
x is 5 and y is 6
```

You can define functions with optional positional arguments

```
function defaults(a,b,x=5,y=6)
    return "$a $b and $x $y"
end

defaults('h','g') # => "h g and 5 6"
defaults('h','g','j') # => "h g and j 6"
defaults('h','g','j','k') # => "h g and j k"
try
    defaults('h') # => ERROR: no method defaults(Char,)
    defaults() # => ERROR: no methods defaults()
catch e
    println(e)
end
```

```
MethodError(Weave.ReportSandBox1.defaults,('h',))
```

You can define functions that take keyword arguments

```
function keyword_args(;k1=4,name2="hello") # note the ;
    return Dict("k1"=>k1,"name2"=>name2)
end

keyword_args(name2="ness") # => ["name2"=>"ness","k1"=>4]
keyword_args(k1="mine") # => ["k1"=>"mine","name2"=>"hello"]
keyword_args() # => ["name2"=>"hello","k1"=>4]
```

You can combine all kinds of arguments in the same function

```
function all_the_args(normal_arg, optional_positional_arg=2; k
    println("normal arg: $normal_arg")
    println("optional arg: $optional_positional_arg")
    println("keyword arg: $keyword_arg")
end

all_the_args(1, 3, keyword_arg=4)
```

```
normal arg: 1
optional arg: 3
keyword arg: 4
```

prints:

```
#   normal arg: 1
#   optional arg: 3
```

Julia has first class functions

```julia
function create_adder(x)
    adder = function (y)
        return x + y
    end
    return adder
end
```

This is “stabby lambda syntax” for creating anonymous functions

```
(x -> x > 2)(3) # => true
```

This function is identical to create_adder implementation above.

```
function create_adder(x)
    y -> x + y
end
```

You can also name the internal function, if you want

```julia
function create_adder(x)
    function adder(y)
        x + y
    end
    adder
end

add_10 = create_adder(10)
add_10(3) # => 13
```

There are built-in higher order functions

```
map(add_10, [1,2,3]) # => [11, 12, 13]
filter(x -> x > 5, [3, 4, 5, 6, 7]) # => [6, 7]
```

We can use list comprehensions for nicer maps

```
[add_10(i) for i=[1, 2, 3]] # => [11, 12, 13]
[add_10(i) for i in [1, 2, 3]] # => [11, 12, 13]
```

Types

Julia has a type system.

Every value has a type; variables do not have types themselves. You can use the `typeof` function to get the type of a value.

```
typeof(5) # => Int64
```

Types are first-class values

```
typeof(Int64) # => DataType
typeof(DataType) # => DataType
```

DataType is the type that represents types, including itself.

- Types are used for documentation, optimizations, and dispatch.

DataType is the type that represents types, including itself.

- Types are used for documentation, optimizations, and dispatch.
- They are not statically checked.

DataType is the type that represents types, including itself.

- Types are used for documentation, optimizations, and dispatch.
- They are not statically checked.
- Users can define types

DataType is the type that represents types, including itself.

- Types are used for documentation, optimizations, and dispatch.
- They are not statically checked.
- Users can define types
- They are like records or structs in other languages.

DataType is the type that represents types, including itself.

- Types are used for documentation, optimizations, and dispatch.
- They are not statically checked.
- Users can define types
- They are like records or structs in other languages.
- New types are defined using the `type` keyword.

type Name

```
type Name
   field::OptionalType
   ...
end

type Tiger
  taillength::Float64
  coatcolor # not including a type annotation is the same as
end
```

The default constructor's arguments

are the properties of the type, in the order they are listed in the definition

```
tigger = Tiger(3.5,"orange") # => Tiger(3.5,"orange")
```

The type doubles as the constructor function for values of that type

```
sherekhan = typeof(tigger)(5.6,"fire") # => Tiger(5.6,"fire")
```

Remark

- These struct-style types are called concrete types

Remark

- These struct-style types are called concrete types
- They can be instantiated, but cannot have subtypes.

Remark

- These struct-style types are called concrete types
- They can be instantiated, but cannot have subtypes.
- The other kind of types is abstract types.

abstract Name

```
abstract Cat # just a name and point in the type hierarchy
```

Abstract types cannot be instantiated, but can have subtypes.

For example, `Number` is an abstract type

```
subtypes(Number) # => 2-element Array{Any,1}:
                 #     Complex{T<:Real}
                 #     Real
subtypes(Cat) # => 0-element Array{Any,1}
```

AbstractString, as the name implies, is also an abstract type

```
subtypes(AbstractString)     # 8-element Array{Any,1}:
                             #  Base.SubstitutionString{T<:Abs
                             #  DirectIndexString
                             #  RepString
                             #  RevString{T<:AbstractString}
                             #  RopeString
                             #  SubString{T<:AbstractString}
                             #  UTF16String
                             #  UTF8String
```

Every type has a super type; use the super function to get it.

```julia
typeof(5) # => Int64
super(Int64) # => Signed
super(Signed) # => Integer
super(Integer) # => Real
super(Real) # => Number
super(Number) # => Any
super(super(Signed)) # => Real
super(Any) # => Any
```

All of these type, except for Int64, are abstract

```julia
typeof("fire") # => ASCIIString
super(ASCIIString) # => DirectIndexString
super(DirectIndexString) # => AbstractString
```

Likewise here with `ASCIIString`

```julia
# <: is the subtyping operator
type Lion <: Cat # Lion is a subtype of Cat
  mane_color
  roar::AbstractString
end
```

Constructors

- You can define more constructors for your type

```
Lion(roar::AbstractString) = Lion("green",roar)
```

Constructors

- You can define more constructors for your type
- Just define a function of the same name as the type

```
Lion(roar::AbstractString) = Lion("green",roar)
```

Constructors

- You can define more constructors for your type
- Just define a function of the same name as the type
- and call an existing constructor to get a value of the correct type

```
Lion(roar::AbstractString) = Lion("green",roar)
```

This is an outer constructor because it's outside the type definition

```
type Panther <: Cat # Panther is also a subtype of Cat
  eye_color
  Panther() = new("green")
  # Panthers will only have this constructor, and no default c
end
```

Using inner constructors,

- like Panther does, gives you control over how values of the type can be created.

Using inner constructors,

- like Panther does, gives you control over how values of the type can be created.
- When possible, you should use outer constructors rather than inner ones.