Learn Julia in Y minutes (2)

Get the code: learnjulia.jl

1. Control Flow
2. Functions
3. Types

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,I
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 and
```

Tuples are created even if you leave out the parentheses

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

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

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

Look how easy it is to swap two values

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

Dictionaries store mappings

```
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 []

```
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.

```
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 in
end
dog 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")
end
dog 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])")
end
mouse 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")
end
mouse 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
end
0
1
2
3
```

Handle exceptions with a try/catch block

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

Functions

The keyword 'function' creates new functions

```
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

```
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

```
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

```
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 `::Any`
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.

```
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

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

Likewise here with ASCIIString

```
# <: 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.