

# Learn Julia in Y minutes (3)

Get the code: [learnjulia.jl](https://github.com/JuliaLang/learnjulia.jl)

- 1 Functions
- 2 Types
- 3 Multiple-Dispatch
- 4 Assignments

# 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 function
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; keyword_arg="foo")  
    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)
```

prints:

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

# 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

**Data Type** 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.

**Data**Type 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<:AbstractString}
                          #  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 constructor
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.

# Multiple-Dispatch

# Functions (generic) and methods (specific)

- 1 In Julia, all named functions are generic functions

For a non-constructor example, let's make a function meow:

```
# Definitions for Lion, Panther, Tiger
function meow(animal::Lion)
    animal.roar # access type properties using dot notation
end

function meow(animal::Panther)
    "grrr"
end

function meow(animal::Tiger)
    "rawwr"
end
```



# Functions (generic) and methods (specific)

- 1 In Julia, all named functions are generic functions
- 2 This means that they are built up from many small methods

For a non-constructor example, let's make a function meow:

```
# Definitions for Lion, Panther, Tiger
function meow(animal::Lion)
    animal.roar # access type properties using dot notation
end

function meow(animal::Panther)
    "grrr"
end

function meow(animal::Tiger)
    "rawwr"
end
```

# Functions (generic) and methods (specific)

- ❶ In Julia, all named functions are generic functions
- ❷ This means that they are built up from many small methods
- ❸ Each constructor for Lion is a method of the generic function Lion.

For a non-constructor example, let's make a function meow:

```
# Definitions for Lion, Panther, Tiger
function meow(animal::Lion)
    animal.roar # access type properties using dot notation
end

function meow(animal::Panther)
    "grrr"
end

function meow(animal::Tiger)
    "rawwr"
end
```

## Testing the meow function

```
meow(tigger) # => "rawwr"  
meow(Lion("brown","ROAAR")) # => "ROAAR"  
meow(Panther()) # => "grrr"
```

Review the local type hierarchy

```
issubtype(Tiger,Cat) # => false  
issubtype(Lion,Cat) # => true  
issubtype(Panther,Cat) # => true
```

## Defining a function that takes Cats

```
function pet_cat(cat::Cat)
    println("The cat says $(meow(cat))")
end

pet_cat(Lion("42")) # => prints "The cat says 42"

try
    pet_cat(tigger) # => ERROR: no method pet_cat(Tiger,)
catch e
    println(e)
end
```

# single vs multiple dispatch

- 1 In OO languages, single dispatch is common;

# single vs multiple dispatch

- 1 In OO languages, single dispatch is common;
- 2 this means that the method is picked based on the type of the first argument.

# single vs multiple dispatch

- ❶ In OO languages, single dispatch is common;
- ❷ this means that the method is picked based on the type of the first argument.
- ❸ In Julia, all of the argument types contribute to selecting the best method.

Let's define a function with more arguments, so we can see the difference

```
function fight(t::Tiger,c::Cat)
    println("The $(t.coatcolor) tiger wins!")
end
# => fight (generic function with 1 method)
```

```
fight(tigger,Panther()) # => prints The orange tiger wins!
fight(tigger,Lion("ROAR")) # => prints The orange tiger wins!
```

```
# Let's change the behavior when the Cat is specifically a Lion
fight(t::Tiger,l::Lion) = println("The $(l.mane_color)-maned lion wins!")
# => fight (generic function with 2 methods)
```

```
fight(tigger,Panther()) # => prints The orange tiger wins!
fight(tigger,Lion("ROAR")) # => prints The green-maned lion wins!
```



# Assignments

# 1. Reading a graph from file

- 1 write the coordinates of nodes of a graph embedded in 2D in a text file (wo word!). Save the file

# 1. Reading a graph from file

- 1 write the coordinates of nodes of a graph embedded in 2D in a text file (wo word!). Save the file
- 2 write the edges in a file as pairs of integers. Save the file

# 1. Reading a graph from file

- 1 write the coordinates of nodes of a graph embedded in 2D in a text file (wo word!). Save the file
- 2 write the edges in a file as pairs of integers. Save the file
- 3 Look for the julia's input/output from from/to files

# 1. Reading a graph from file

- 1 write the coordinates of nodes of a graph embedded in 2D in a text file (wo word!). Save the file
- 2 write the edges in a file as pairs of integers. Save the file
- 3 Look for the julia's input/output from from/to files
- 4 Open and read the first file in an 2-array named  $V$

# 1. Reading a graph from file

- 1 write the coordinates of nodes of a graph embedded in 2D in a text file (wo word!). Save the file
- 2 write the edges in a file as pairs of integers. Save the file
- 3 Look for the julia's input/output from from/to files
- 4 Open and read the first file in an 2-array named  $V$
- 5 Open and read the second file in an 2-array named  $EV$

## 2. Compute the edges incident on a node

- 1 Write a function to compute the edges incident on a given node

## 2. Compute the edges incident on a node

- 1 Write a function to compute the edges incident on a given node
- 2 Write a file putting of each row the indices of edges incident on each node