FREE eBook

LEARNING

# Julia Language

Free unaffiliated eBook created from
**Stack Overflow contributors.**

#julia-lang

# Table of Contents

# About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: julia-language

It is an unofficial and free Julia Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Julia Language.



Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

# Chapter 1: Getting started with Julia Language

## Versions

| Version | Release Date |
|---|---|
| 0.6.0-dev | 2017-06-01 |
| 0.5.0 | 2016-09-19 |
| 0.4.0 | 2015-10-08 |
| 0.3.0 | 2014-08-21 |
| 0.2.0 | 2013-11-17 |
| 0.1.0 | 2013-02-14 |

## Examples

### Hello, World!

```
println("Hello, World!")
```

To run Julia, first get the interpreter from the website's download page. The current stable release is v0.5.0, and this version is recommended for most users. Certain package developers or power users may choose to use the nightly build, which is far less stable.

When you have the interpreter, write your program in a file named `hello.jl`. It can then be run from a system terminal as:

```
$ julia hello.jl
Hello, World!
```

Julia can also be run interactively, by running the `julia` program. You should see a header and prompt, as follows:

```
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.4.2 (2015-12-06 21:47 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
```

```
|__/                   |  x86_64-w64-mingw32

julia>
```

You can run any Julia code in this REPL, so try:

```
julia> println("Hello, World!")
Hello, World!
```

This example makes use of a string, `"Hello, World!"`, and of the `println` function—one of many in the standard library. For more information or help, try the following sources:

- The REPL has an integrated help mode to access documentation.
- The official documentation is quite comprehensive.
- Stack Overflow has a small but growing collection of examples.
- Users on Gitter are happy to help with small questions.
- The primary online discussion venue for Julia is the Discourse forum at discourse.julialang.org. More involved questions should be posted here.
- A collection of tutorials and books can be found here.

Read Getting started with Julia Language online: https://riptutorial.com/julia-lang/topic/485/getting-started-with-julia-language

# Chapter 2: @goto and @label

## Syntax

- @goto label
- @label label

## Remarks

Overuse or inappropriate use of advanced control flow makes code hard to read. `@goto` or its equivalents in other languages, when used improperly, leads to unreadable spaghetti code.

Similar to languages like C, one cannot jump between functions in Julia. This also means that `@goto` is not possible at the top-level; it will only work within a function. Furthermore, one cannot jump from an inner function to its outer function, or from an outer function to an inner function.

## Examples

### Input validation

Although not traditionally considered loops, the `@goto` and `@label` macros can be used for more advanced control flow. One use case is when the failure of one part should lead to the retry of an entire function, often useful in input validation:

```
function getsequence()
    local a, b

@label start
    print("Input an integer: ")
    try
        a = parse(Int, readline())
    catch
        println("Sorry, that's not an integer.")
        @goto start
    end

    print("Input a decimal: ")
    try
        b = parse(Float64, readline())
    catch
        println("Sorry, that doesn't look numeric.")
        @goto start
    end

    a, b
end
```

However, this use case is often more clear using recursion:

```
function getsequence()
    local a, b

    print("Input an integer: ")
    try
        a = parse(Int, readline())
    catch
        println("Sorry, that's not an integer.")
        return getsequence()
    end

    print("Input a decimal: ")
    try
        b = parse(Float64, readline())
    catch
        println("Sorry, that doesn't look numeric.")
        return getsequence()
    end

    a, b
end
```

Although both examples do the same thing, the second is easier to understand. However, the first one is more performant (because it avoids the recursive call). In most cases, the cost of the call does not matter; but in limited situations, the first form is acceptable.

## Error cleanup

In languages such as C, the `@goto` statement is often used to ensure a function cleans up necessary resources, even in the event of an error. This is less important in Julia, because exceptions and `try-finally` blocks are often used instead.

However, it is possible for Julia code to interface with C code and C APIs, and so sometimes functions still need to be written like C code. The below example is contrived, but demonstrates a common use case. The Julia code will call `Libc.malloc` to allocate some memory (this simulates a C API call). If not all allocations succeed, then the function should free the resources obtained so far; otherwise, the allocated memory is returned.

```
using Base.Libc
function allocate_some_memory()
    mem1 = malloc(100)
    mem1 == C_NULL && @goto fail
    mem2 = malloc(200)
    mem2 == C_NULL && @goto fail
    mem3 = malloc(300)
    mem3 == C_NULL && @goto fail
    return mem1, mem2, mem3

@label fail
    free(mem1)
    free(mem2)
    free(mem3)
end
```

Read @goto and @label online: https://riptutorial.com/julia-lang/topic/5564/-goto-and--label

# Chapter 3: Arithmetic

## Syntax

- +x
- -x
- a + b
- a - b
- a * b
- a / b
- a ^ b
- a % b
- 4a
- sqrt(a)

## Examples

### Quadratic Formula

Julia uses similar binary operators for basic arithmetic operations as does mathematics or other programming languages. Most operators can be written in infix notation (that is, placed in between the values being computed). Julia has an order of operations that matches the common convention in mathematics.

For instance, the below code implements the quadratic formula, which demonstrates the `+`, `-`, `*`, and `/` operators for addition, subtraction, multiplication, and division respectively. Also shown is *implicit multiplication*, where a number can be placed directly before a symbol to mean multiplication; that is, `4a` means the same as `4*a`.

```
function solvequadratic(a, b, c)
    d = sqrt(b^2 - 4a*c)
    (-b - d) / 2a, (-b + d) / 2a
end
```

Usage:

```
julia> solvequadratic(1, -2, -3)
(-1.0,3.0)
```

### Sieve of Eratosthenes

The remainder operator in Julia is the `%` operator. This operator behaves similarly to the `%` in languages such as C and C++. `a % b` is the signed remainder left over after dividing `a` by `b`.

This operator is very useful for implementing certain algorithms, such as the following

implementation of the Sieve of Eratosthenes.

```
iscoprime(P, i) = !any(x -> i % x == 0, P)

function sieve(n)
    P = Int[]
    for i in 2:n
        if iscoprime(P, i)
            push!(P, i)
        end
    end
    P
end
```

Usage:

```
julia> sieve(20)
8-element Array{Int64,1}:
  2
  3
  5
  7
 11
 13
 17
 19
```

### Matrix Arithmetic

Julia uses the standard mathematical meanings of arithmetic operations when applied to matrices. Sometimes, elementwise operations are desired instead. These are marked with a full stop (.) preceding the operator to be done elementwise. (Note that elementwise operations are often not as efficient as loops.)

## Sums

The `+` operator on matrices is a matrix sum. It is similar to an elementwise sum, but it does not broadcast shape. That is, if `A` and `B` are the same shape, then `A + B` is the same as `A .+ B`; otherwise, `A + B` is an error, whereas `A .+ B` may not necessarily be.

```
julia> A = [1 2
            3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [5 6
            7 8]
2×2 Array{Int64,2}:
 5  6
 7  8

julia> A + B
2×2 Array{Int64,2}:
```

```
  6   8
 10  12

julia> A .+ B
2×2 Array{Int64,2}:
  6   8
 10  12

julia> C = [9, 10]
2-element Array{Int64,1}:
  9
 10

julia> A + C
ERROR: DimensionMismatch("dimensions must match")
 in promote_shape(::Tuple{Base.OneTo{Int64},Base.OneTo{Int64}}, ::Tuple{Base.OneTo{Int64}}) at
./operators.jl:396
 in promote_shape(::Array{Int64,2}, ::Array{Int64,1}) at ./operators.jl:382
 in _elementwise(::Base.#+, ::Array{Int64,2}, ::Array{Int64,1}, ::Type{Int64}) at
./arraymath.jl:61
 in +(::Array{Int64,2}, ::Array{Int64,1}) at ./arraymath.jl:53

julia> A .+ C
2×2 Array{Int64,2}:
 10  11
 13  14
```

Likewise, `-` computes a matrix difference. Both `+` and `-` can also be used as unary operators.

## Products

The `*` operator on matrices is the matrix product (not the elementwise product). For an elementwise product, use the `.*` operator. Compare (using the same matrices as above):

```
julia> A * B
2×2 Array{Int64,2}:
 19  22
 43  50

julia> A .* B
2×2 Array{Int64,2}:
  5  12
 21  32
```

## Powers

The `^` operator computes matrix exponentiation. Matrix exponentiation can be useful for computing values of certain recurrences quickly. For instance, the Fibonacci numbers can be generated by the matrix expression

```
fib(n) = (BigInt[1 1; 1 0]^n)[2]
```

As usual, the `.^` operator can be used where elementwise exponentiation is the desired operation.

# Chapter 4: Arrays

## Syntax

- [1,2,3]
- [1 2 3]
- [1 2 3; 4 5 6; 7 8 9]
- Array(type, dims...)
- ones(type, dims...)
- zeros(type, dims...)
- trues(type, dims...)
- falses(type, dims...)
- push!(A, x)
- pop!(A)
- unshift!(A, x)
- shift!(A)

## Parameters

| Parameters | Remarks |
|---|---|
| **For** | `push!(A, x)`, `unshift!(A, x)` |
| `A` | The array to add to. |
| `x` | The element to add to the array. |

## Examples

### Manual construction of a simple array

One can initialize a Julia array by hand, using the square-brackets syntax:

```
julia> x = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3
```

The first line after the command shows the size of the array you created. It also shows the type of its elements and its dimensionality (int this case `Int64` and `1`, repectively). For a two-dimensional array, you can use spaces and semi-colon:

```
julia> x = [1 2 3; 4 5 6]
2x3 Array{Int64,2}:
```

```
 1  2  3
 4  5  6
```

To create an uninitialized array, you can use the `Array(type, dims...)` method:

```
julia> Array(Int64, 3, 3)
3x3 Array{Int64,2}:
 0  0  0
 0  0  0
 0  0  0
```

The functions `zeros`, `ones`, `trues`, `falses` have methods that behave exactly the same way, but produce arrays full of `0.0`, `1.0`, `True` or `False`, respectively.

## Array types

In Julia, Arrays have types parametrized by two variables: a type `T` and a dimensionality `D` (`Array{T, D}`). For a 1-dimensional array of integers, the type is:

```
julia> x = [1, 2, 3];
julia> typeof(x)
Array{Int64, 1}
```

If the array is a 2-dimensional matrix, `D` equals to 2:

```
julia> x = [1 2 3; 4 5 6; 7 8 9]
julia> typeof(x)
Array{Int64, 2}
```

The element type can also be abstract types:

```
julia> x = [1 2 3; 4 5 "6"; 7 8 9]
3x3 Array{Any,2}:
 1  2  3
 4  5 "6"
 7  8  9
```

Here `Any` (an abstract type) is the type of the resulting array.

**Specifying Types when Creating Arrays**

When we create an Array in the way described above, Julia will do its best to infer the proper type that we might want. In the initial examples above, we entered inputs that looked like integers, and so Julia defaulted to the default `Int64` type. At times, however, we might want to be more specific. In the following example, we specify that we want the type to be instead `Int8`:

```
x1 = Int8[1 2 3; 4 5 6; 7 8 9]
typeof(x1)  ## Array{Int8,2}
```

We could even specify the type as something such as `Float64`, even if we write the inputs in a way

that might otherwise be interpreted as integers by default (e.g. writing `1` instead of `1.0`). e.g.

```
x2 = Float64[1 2 3; 4 5 6; 7 8 9]
```

## Arrays of Arrays - Properties and Construction

In Julia, you can have an Array that holds other Array type objects. Consider the following examples of initializing various types of Arrays:

```
A = Array{Float64}(10,10)  # A single Array, dimensions 10 by 10, of Float64 type objects

B = Array{Array}(10,10,10)  # A 10 by 10 by 10 Array.  Each element is an Array of unspecified
type and dimension.

C = Array{Array{Float64}}(10)  ## A length 10, one-dimensional Array.  Each element is an
Array of Float64 type objects but unspecified dimensions

D = Array{Array{Float64, 2}}(10)  ## A length 10, one-dimensional Array.  Each element of is
an 2 dimensional array of Float 64 objects
```

Consider for instance, the differences between C and D here:

```
julia> C[1] = rand(3)
3-element Array{Float64,1}:
 0.604771
 0.985604
 0.166444

julia> D[1] = rand(3)
ERROR: MethodError:
```

`rand(3)` produces an object of type `Array{Float64,1}`. Since the only specification for the elements of `C` are that they be Arrays with elements of type Float64, this fits within the definition of `C`. But, for `D` we specified that the elements must be 2 dimensional Arrays. Thus, since `rand(3)` does not produce a 2 dimensional array, we cannot use it to assign a value to a specific element of `D`

**Specify Specific Dimensions of Arrays within an Array**

Although we can specify that an Array will hold elements which are of type Array, and we can specify that, e.g. those elements should be 2-dimensional Arrays, we cannot directly specify the dimenions of those elements. E.g. we can't directly specify that we want an Array holding 10 Arrays, each of which being 5,5. We can see this from the syntax for the `Array()` function used to construct an Array:

> **Array{T}(dims)**
>
> constructs an uninitialized dense array with element type T. dims may be a tuple or a series of integer arguments. The syntax Array(T, dims) is also available, but deprecated.

The type of an Array in Julia encompasses the number of the dimensions but not the size of those

dimensions. Thus, there is no place in this syntax to specify the precise dimensions. Nevertheless, a similar effect could be achieved using an Array comprehension:

```
E = [Array{Float64}(5,5) for idx in 1:10]
```

Note: this documentation mirrors the following SO Answer

## Initialize an Empty Array

We can use the `[]` to create an empty Array in Julia. The simplest example would be:

```
A = [] # 0-element Array{Any,1}
```

Arrays of type `Any` will generally not perform as well as those with a specified type. Thus, for instance, we can use:

```
B = Float64[]  ## 0-element Array{Float64,1}
C = Array{Float64}[]  ## 0-element Array{Array{Float64,N},1}
D = Tuple{Int, Int}[] ## 0-element Array{Tuple{Int64,Int64},1}
```

See Initialize an Empty Array of Tuples in Julia for source of last example.

## Vectors

Vectors are one-dimensional arrays, and support mostly the same interface as their multi-dimensional counterparts. However, vectors also support additional operations.

First, note that `Vector{T}` where `T` is some type means the same as `Array{T,1}`.

```
julia> Vector{Int}
Array{Int64,1}

julia> Vector{Float64}
Array{Float64,1}
```

One reads `Array{Int64,1}` as "one-dimensional array of `Int64`".

Unlike multi-dimensional arrays, vectors can be resized. Elements can be added or removed from the front or back of the vector. These operations are all constant amortized time.

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> push!(A, 4)
4-element Array{Int64,1}:
 1
 2
 3
```

```
 4

julia> A
4-element Array{Int64,1}:
 1
 2
 3
 4

julia> pop!(A)
4

julia> A
3-element Array{Int64,1}:
 1
 2
 3

julia> unshift!(A, 0)
4-element Array{Int64,1}:
 0
 1
 2
 3

julia> A
4-element Array{Int64,1}:
 0
 1
 2
 3

julia> shift!(A)
0

julia> A
3-element Array{Int64,1}:
 1
 2
 3
```

As is convention, each of these functions `push!`, `pop!`, `unshift!`, and `shift!` ends in an exclamation mark to indicate that they are mutate their argument. The functions `push!` and `unshift!` return the array, whereas `pop!` and `shift!` return the element removed.

### Concatenation

It is often useful to build matrices out of smaller matrices.

## Horizontal Concatenation

Matrices (and vectors, which are treated as column vectors) can be horizontally concatenated using the `hcat` function.

```
julia> hcat([1 2; 3 4], [5 6 7; 8 9 10], [11, 12])
2×6 Array{Int64,2}:
```

```
 1  2  5  6   7  11
 3  4  8  9  10  12
```

There is convenience syntax available, using square bracket notation and spaces:

```
julia> [[1 2; 3 4] [5 6 7; 8 9 10] [11, 12]]
2×6 Array{Int64,2}:
 1  2  5  6   7  11
 3  4  8  9  10  12
```

This notation can closely match the notation for block matrices used in linear algebra:

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

julia> B = [5 6; 7 8]
2×2 Array{Int64,2}:
 5  6
 7  8

julia> [A B]
2×4 Array{Int64,2}:
 1  2  5  6
 3  4  7  8
```

Note that you cannot horizontally concatenate a single matrix using the `[]` syntax, as that would instead create a one-element vector of matrices:

```
julia> [A]
1-element Array{Array{Int64,2},1}:
 [1 2; 3 4]
```

# Vertical Concatenation

Vertical concatenation is like horizontal concatenation, but in the vertical direction. The function for vertical concatenation is `vcat`.

```
julia> vcat([1 2; 3 4], [5 6; 7 8; 9 10], [11 12])
6×2 Array{Int64,2}:
  1   2
  3   4
  5   6
  7   8
  9  10
 11  12
```

Alternatively, square bracket notation can be used with semicolons `;` as the delimiter:

```
julia> [[1 2; 3 4]; [5 6; 7 8; 9 10]; [11 12]]
6×2 Array{Int64,2}:
```

```
 1   2
 3   4
 5   6
 7   8
 9  10
11  12
```

Vectors can be vertically concatenated too; the result is a vector:

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [4, 5]
2-element Array{Int64,1}:
 4
 5

julia> [A; B]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

Horizontal and vertical concatenation can be combined:

```
julia> A = [1 2
            3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> B = [5 6 7]
1×3 Array{Int64,2}:
 5  6  7

julia> C = [8, 9]
2-element Array{Int64,1}:
 8
 9

julia> [A C; B]
3×3 Array{Int64,2}:
 1  2  8
 3  4  9
 5  6  7
```

Read Arrays online: https://riptutorial.com/julia-lang/topic/5437/arrays

# Chapter 5: Closures

## Syntax

- x -> [body]
- (x, y) -> [body]
- (xs...) -> [body]

## Remarks

0.4.0

In older versions of Julia, closures and anonymous functions had a runtime performance penalty. This penalty has been eliminated in 0.5.

## Examples

### Function Composition

We can define a function to perform function composition using anonymous function syntax:

```
f ∘ g = x -> f(g(x))
```

Note that this definition is equivalent to each of the following definitions:

```
∘(f, g) = x -> f(g(x))
```

or

```
function ∘(f, g)
    x -> f(g(x))
end
```

recalling that in Julia, `f ∘ g` is just syntax sugar for `∘(f, g)`.

We can see that this function composes correctly:

```
julia> double(x) = 2x
double (generic function with 1 method)

julia> triple(x) = 3x
triple (generic function with 1 method)

julia> const sextuple = double ∘ triple
(::#17) (generic function with 1 method)

julia> sextuple(1.5)
```

```
9.0
```

0.5.0

In version v0.5, this definition is very performant. We can look into the LLVM code generated:

```
julia> @code_llvm sextuple(1)

define i64 @"julia_#17_71238"(i64) #0 {
top:
  %1 = mul i64 %0, 6
  ret i64 %1
}
```

It is clear that the two multiplications have been folded into a single multiplication, and that this function is as efficient as is possible.

How does this higher-order function work? It creates a so-called closure, which consists of not just its code, but also keeps track of certain variables from its scope. All functions in Julia that are not created at top-level scope are closures.

0.5.0

One can inspect the variables closed over through the fields of the closure. For instance, we see that:

```
julia> (sin ∘ cos).f
sin (generic function with 10 methods)

julia> (sin ∘ cos).g
cos (generic function with 10 methods)
```

## Implementing Currying

One application of closures is to partially apply a function; that is, provide some arguments now and create a function that takes the remaining arguments. Currying is a specific form of partial application.

Let's start with the simple function `curry(f, x)` that will provide the first argument to a function, and expect additional arguments later. The definition is fairly straightforward:

```
curry(f, x) = (xs...) -> f(x, xs...)
```

Once again, we use anonymous function syntax, this time in combination with variadic argument syntax.

We can implement some basic functions in tacit (or point-free) style using this `curry` function.

```
julia> const double = curry(*, 2)
(::#19) (generic function with 1 method)
```

```
julia> double(10)
20

julia> const simon_says = curry(println, "Simon: ")
(::#19) (generic function with 1 method)

julia> simon_says("How are you?")
Simon: How are you?
```

Functions maintain the generism expected:

```
julia> simon_says("I have ", 3, " arguments.")
Simon: I have 3 arguments.

julia> double([1, 2, 3])
3-element Array{Int64,1}:
 2
 4
 6
```

## Introduction to Closures

Functions are an important part of Julia programming. They can be defined directly within modules, in which case the functions are referred to as *top-level*. But functions can also be defined within other functions. Such functions are called "closures".

Closures capture the variables in their outer function. A top-level function can only use global variables from their module, function parameters, or local variables:

```
x = 0  # global
function toplevel(y)
    println("x = ", x, " is a global variable")
    println("y = ", y, " is a parameter")
    z = 2
    println("z = ", z, " is a local variable")
end
```

A closure, on the other hand, can use all those in addition to variables from outer functions that it captures:

```
x = 0  # global
function toplevel(y)
    println("x = ", x, " is a global variable")
    println("y = ", y, " is a parameter")
    z = 2
    println("z = ", z, " is a local variable")

    function closure(v)
        println("v = ", v, " is a parameter")
        w = 3
        println("w = ", w, " is a local variable")
        println("x = ", x, " is a global variable")
        println("y = ", y, " is a closed variable (a parameter of the outer function)")
        println("z = ", z, " is a closed variable (a local of the outer function)")
    end
```

```
end
```

If we run `c = toplevel(10)`, we see the result is

```
julia> c = toplevel(10)
x = 0 is a global variable
y = 10 is a parameter
z = 2 is a local variable
(::closure) (generic function with 1 method)
```

Note that the tail expression of this function is a function in itself; that is, a closure. We can call the closure `c` like it was any other function:

```
julia> c(11)
v = 11 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 10 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)
```

Note that `c` still has access to the variables `y` and `z` from the `toplevel` call — even though `toplevel` has already returned! Each closure, even those returned by the same function, closes over different variables. We can call `toplevel` again

```
julia> d = toplevel(20)
x = 0 is a global variable
y = 20 is a parameter
z = 2 is a local variable
(::closure) (generic function with 1 method)

julia> d(22)
v = 22 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 20 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)

julia> c(22)
v = 22 is a parameter
w = 3 is a local variable
x = 0 is a global variable
y = 10 is a closed variable (a parameter of the outer function)
z = 2 is a closed variable (a local of the outer function)
```

Note that despite `d` and `c` having the same code, and being passed the same arguments, their output is different. They are distinct closures.

Read Closures online: https://riptutorial.com/julia-lang/topic/5724/closures

# Chapter 6: Combinators

## Remarks

Although combinators have limited practical use, they are a useful tool in education to understand how programming is fundamentally linked to logic, and how very simple building blocks can combine to create very complex behaviour. In the context of Julia, learning how to create and use combinators will strengthen an understanding of how to program in a functional style in Julia.

## Examples

### The Y or Z Combinator

Although Julia is not a purely functional language, it has full support for many of the cornerstones of functional programming: first-class functions, lexical scope, and closures.

The fixed-point combinator is a key combinator in functional programming. Because Julia has eager evaluation semantics (as do many functional languages, including Scheme, which Julia is heavily inspired by), Curry's original Y-combinator will not work out of the box:

```
Y(f) = (x -> f(x(x)))(x -> f(x(x)))
```

However, a close relative of the Y-combinator, the Z-combinator, will indeed work:

```
Z(f) = x -> f(Z(f), x)
```

This combinator takes a function and returns a function that when called with argument `x`, gets passed itself and `x`. Why would it be useful for a function to be passed itself? This allows recursion without actually referencing the name of the function at all!

```
fact(f, x) = x == 0 ? 1 : x * f(x)
```

Hence, `Z(fact)` becomes a recursive implementation of the factorial function, despite no recursion being visible in this function definition. (Recursion is evident in the definition of the `Z` combinator, of course, but that is inevitable in an eager language.) We can verify that our function indeed works:

```
julia> Z(fact)(10)
3628800
```

Not only that, but it is as fast as we can expect from a recursive implementation. The LLVM code demonstrates that the result is compiled into a plain old branch, subtract, call, and multiply:

```
julia> @code_llvm Z(fact)(10)

define i64 @"julia_#1_70252"(i64) #0 {
```

```
top:
  %1 = icmp eq i64 %0, 0
  br i1 %1, label %L11, label %L8

L8:                                               ; preds = %top
  %2 = add i64 %0, -1
  %3 = call i64 @"julia_#1_70060"(i64 %2) #0
  %4 = mul i64 %3, %0
  br label %L11

L11:                                              ; preds = %top, %L8
  %"#temp#.0" = phi i64 [ %4, %L8 ], [ 1, %top ]
  ret i64 %"#temp#.0"
}
```

### The SKI Combinator System

The SKI combinator system is sufficient to represent any lambda calculus terms. (In practice, of course, lambda abstractions blow up to exponential size when they are translated into SKI.) Due to the simplicity of the system, implementing the S, K, and I combinators is extraordinarily simple:

## A Direct Translation from Lambda Calculus

```
const S = f -> g -> z -> f(z)(g(z))
const K = x -> y -> x
const I = x -> x
```

We can confirm, using the unit testing system, that each combinator has the expected behaviour.

The I combinator is easiest to verify; it should return the given value unchanged:

```
using Base.Test
@test I(1) === 1
@test I(I) === I
@test I(S) === S
```

The K combinator is also fairly straightforward: it should discard its second argument.

```
@test K(1)(2) === 1
@test K(S)(I) === S
```

The S combinator is the most complex; its behaviour can be summarized as applying the first two arguments to the third argument, the applying the first result to the second. We can most easily test the S combinator by testing some of its curried forms. `S(K)`, for instance, should simply return its second argument and discard its first, as we see happens:

```
@test S(K)(S)(K) === K
@test S(K)(S)(I) === I
```

`S(I)(I)` should apply its argument to itself:

```
@test S(I)(I)(I) === I
@test S(I)(I)(K) === K(K)
@test S(I)(I)(S(I)) === S(I)(S(I))
```

`S(K(S(I)))(K)` applies its second argument to its first:

```
@test S(K(S(I)))(K)(I)(I) === I
@test S(K(S(I)))(K)(K)(S(K)) === S(K)(K)
```

The I combinator described above has a name in standard `Base` Julia: `identity`. Thus, we could have rewritten the above definitions with the following alternative definition of `I`:

```
const I = identity
```

## Showing SKI Combinators

One weakness with the approach above is that our functions do not show as nicely as we might like. Could we replace

```
julia> S
(::#3) (generic function with 1 method)

julia> K
(::#9) (generic function with 1 method)

julia> I
(::#13) (generic function with 1 method)
```

with some more informative displays? The answer is yes! Let's restart the REPL, and this time define how each function is to be shown:

```
const S = f -> g -> z -> f(z)(g(z));
const K = x -> y -> x;
const I = x -> x;
for f in (:S, :K, :I)
    @eval Base.show(io::IO, ::typeof($f)) = print(io, $(string(f)))
    @eval Base.show(io::IO, ::MIME"text/plain", ::typeof($f)) = show(io, $f)
end
```

It's important to avoid showing anything until we have finished defining functions. Otherwise, we risk invalidating the method cache, and our new methods will not seem to immediately take effect. This is why we have put semicolons in the above definitions. The semicolons suppress the REPL's output.

This makes the functions display nicely:

```
julia> S
S

julia> K
K
```

```
julia> I
I
```

However, we still run into problems when we try to display a closure:

```
julia> S(K)
(::#2) (generic function with 1 method)
```

It would be nicer to display that as `S(K)`. To do that, we must exploit that the closures have their own individual types. We can access these types and add methods to them through reflection, using `typeof` and the `primary` field of the `name` field of the type. Restart the REPL again; we will make further changes:

```
const S = f -> g -> z -> f(z)(g(z));
const K = x -> y -> x;
const I = x -> x;
for f in (:S, :K, :I)
    @eval Base.show(io::IO, ::typeof($f)) = print(io, $(string(f)))
    @eval Base.show(io::IO, ::MIME"text/plain", ::typeof($f)) = show(io, $f)
end
Base.show(io::IO, s::typeof(S(I)).name.primary) = print(io, "S(", s.f, ')')
Base.show(io::IO, s::typeof(S(I)(I)).name.primary) =
    print(io, "S(", s.f, ')', '(', s.g, ')')
Base.show(io::IO, k::typeof(K(I)).name.primary) = print(io, "K(", k.x, ')')
Base.show(io::IO, ::MIME"text/plain", f::Union{
    typeof(S(I)).name.primary,
    typeof(S(I)(I)).name.primary,
    typeof(K(I)).name.primary
}) = show(io, f)
```

And now, at last, things display as we would like them to:

```
julia> S(K)
S(K)

julia> S(K)(I)
S(K)(I)

julia> K
K

julia> K(I)
K(I)

julia> K(I)(K)
I
```

Read Combinators online: https://riptutorial.com/julia-lang/topic/5758/combinators

# Chapter 7: Comparisons

## Syntax

- x < y # if `x` is strictly less than `y`
- x > y # if `x` is strictly greater than `y`
- x == y # if `x` is equal to `y`
- x === y # alternatively `x ≡ y`, if `x` is egal to `y`
- x ≤ y # alternatively `x <= y`, if `x` is less than or equal to `y`
- x ≥ y # alternatively `x >= y`, if `x` is greater than or equal to `y`
- x ≠ y # alternatively `x != y`, if `x` is not equal to `y`
- x ≈ y # if `x` is approximately equal to `y`

## Remarks

Be careful about flipping comparison signs around. Julia defines many comparison functions by default without defining the corresponding flipped version. For instance, one can run

```
julia> Set(1:3) ⊆ Set(0:5)
true
```

but it does not work to do

```
julia> Set(0:5) ⊇ Set(1:3)
ERROR: UndefVarError: ⊇ not defined
```

## Examples

### Chained Comparisons

Multiple comparison operators used together are chained, as if connected via the `&&` operator. This can be useful for readable and mathematically concise comparison chains, such as

```
# same as 0 < i && i <= length(A)
isinbounds(A, i)        = 0 < i ≤ length(A)

# same as Set() != x && issubset(x, y)
isnonemptysubset(x, y) = Set() ≠ x ⊆ y
```

However, there is an important difference between `a > b > c` and `a > b && b > c`; in the latter, the term `b` is evaluated twice. This does not matter much for plain old symbols, but could matter if the terms themselves have side effects. For instance,

```
julia> f(x) = (println(x); 2)
f (generic function with 1 method)
```

```
julia> 3 > f("test") > 1
test
true

julia> 3 > f("test") && f("test") > 1
test
test
true
```

Let's take a deeper look at chained comparisons, and how they work, by seeing how they are parsed and lowered into expressions. First, consider the simple comparison, which we can see is just a plain old function call:

```
julia> dump(:(a > b))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol >
    2: Symbol a
    3: Symbol b
  typ: Any
```

Now if we chain the comparison, we notice that the parsing has changed:

```
julia> dump(:(a > b >= c))
Expr
  head: Symbol comparison
  args: Array{Any}((5,))
    1: Symbol a
    2: Symbol >
    3: Symbol b
    4: Symbol >=
    5: Symbol c
  typ: Any
```

After parsing, the expression is then lowered to its final form:

```
julia> expand(:(a > b >= c))
:(begin
        unless a > b goto 3
        return b >= c
        3:
        return false
    end)
```

and we note indeed that this is the same as for `a > b && b >= c`:

```
julia> expand(:(a > b && b >= c))
:(begin
        unless a > b goto 3
        return b >= c
        3:
        return false
    end)
```

## Ordinal Numbers

We will look at how to implement custom comparisons by implementing a custom type, ordinal numbers. To simplify the implementation, we will focus on a small subset of these numbers: all ordinal numbers up to but not including $\varepsilon_0$. Our implementation is focused on simplicity, not speed; however, the implementation is not slow either.

We store ordinal numbers by their Cantor normal form. Because ordinal arithmetic is not commutative, we will take the common convention of storing most significant terms first.

```
immutable OrdinalNumber <: Number
    βs::Vector{OrdinalNumber}
    cs::Vector{Int}
end
```

Since the Cantor normal form is unique, we may test equality simply through recursive equality:

0.5.0

In version v0.5, there is a very nice syntax for doing this compactly:

```
import Base: ==
α::OrdinalNumber == β::OrdinalNumber = α.βs == β.βs && α.cs == β.cs
```

0.5.0

Otherwise, define the function as is more typical:

```
import Base: ==
==(α::OrdinalNumber, β::OrdinalNumber) = α.βs == β.βs && α.cs == β.cs
```

To finish our order, because this type has a total order, we should overload the `isless` function:

```
import Base: isless
function isless(α::OrdinalNumber, β::OrdinalNumber)
    for i in 1:min(length(α.cs), length(β.cs))
        if α.βs[i] < β.βs[i]
            return true
        elseif α.βs[i] == β.βs[i] && α.cs[i] < β.cs[i]
            return true
        end
    end
    return length(α.cs) < length(β.cs)
end
```

To test our order, we can create some methods to make ordinal numbers. Zero, of course, is obtained by having no terms in the Cantor normal form:

```
const ORDINAL_ZERO = OrdinalNumber([], [])
Base.zero(::Type{OrdinalNumber}) = ORDINAL_ZERO
```

We can defined an `expω` to compute `ω^α`, and use that to compute 1 and ω:

```
expω(α) = OrdinalNumber([α], [1])
const ORDINAL_ONE = expω(ORDINAL_ZERO)
Base.one(::Type{OrdinalNumber}) = ORDINAL_ONE
const ω = expω(ORDINAL_ONE)
```

We now have a fully functional ordering function on ordinal numbers:

```
julia> ORDINAL_ZERO < ORDINAL_ONE < ω < expω(ω)
true

julia> ORDINAL_ONE > ORDINAL_ZERO
true

julia> sort([ORDINAL_ONE, ω, expω(ω), ORDINAL_ZERO])

4-element Array{OrdinalNumber,1}:

OrdinalNumber(OrdinalNumber[],Int64[])

OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[],Int64[])],[1])

OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[],Int64[])],[

OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber(OrdinalNumber[OrdinalNumber
```

In the last example, we see that the printing of ordinal numbers could be better, but the result is as expected.

## Standard Operators

Julia supports a very large set of comparison operators. These include

1. All of the following unicode sequences: `> < >= ≥ <= ≤ == === ≡ != ≠ !== ≢ ∈ ∉ ∋ ∌ ⊆ ⊈ ⊂ ⊄ ⊊ ∝ ∊ ∍ ∥ ∦ ∷ ∺ ∻ ∽ ∾ ≁ ≃ ≄ ≅ ≆ ≇ ≈ ≉ ≊ ≋ ≍ ≎ ≐ ≑ ≒ ≓ ≔ ≕ ≖ ≗ ≘ ≙ ≚ ≛ ≜ ≝ ≞ ≟ ≣ ≦ ≧ ≨ ≩ ≪ ≫ ≬ ≭ ≮ ≯ ≰ ≱ ≲ ≳ ≴ ≵ ≶ ≷ ≸ ≹ ≺ ≻ ≼ ≽ ≾ ≿ ⊀ ⊁ ⊃ ⊅ ⊇ ⊉ ⊋ ⊏ ⊐ ⊑ ⊒ ⊜ ⊩ ⊬ ⊮ ⊰ ⊱ ⊲ ⊳ ⊴ ⊵ ⊶ ⊷ ⋍ ⋐ ⋑ ⋕ ⋖ ⋗ ⋘ ⋙ ⋚ ⋛ ⋜ ⋝ ⋞ ⋟ ⋠ ⋡ ⋢ ⋣ ⋤ ⋥ ⋦ ⋧ ⋨ ⋩ ⋪ ⋫ ⋬ ⋭ ⋲ ⋳ ⋴ ⋵ ⋶ ⋷ ⋸ ⋹ ⋺ ⋻ ⋼ ⋽ ⋾ ⋿ ⟈ ⟉ ⟒ ⦷ ⧀ ⧁ ⧡ ⧣ ⧤ ⧥ ⩦ ⩧ ⩪ ⩫ ⩬ ⩭ ⩮ ⩯ ⩰ ⩱ ⩲ ⩳ ⩴ ⩵ ⩶ ⩷ ⩸ ⩹ ⩺ ⩻ ⩼ ⩽ ⩾ ⩿ ⪀ ⪁ ⪂ ⪃ ⪄ ⪅ ⪆ ⪇ ⪈ ⪉ ⪊ ⪋ ⪌ ⪍ ⪎ ⪏ ⪐ ⪑ ⪒ ⪓ ⪔ ⪕ ⪖ ⪗ ⪘ ⪙ ⪚ ⪛ ⪜ ⪝ ⪞ ⪟ ⪠ ⪡ ⪢ ⪣ ⪤ ⪥ ⪦ ⪧ ⪨ ⪩ ⪪ ⪫ ⪬ ⪭ ⪮ ⪯ ⪰ ⪱ ⪲ ⪳ ⪴ ⪵ ⪶ ⪷ ⪸ ⪹ ⪺ ⪻ ⪼ ⪽ ⪾ ⪿ ⫀ ⫁ ⫂ ⫃ ⫄ ⫅ ⫆ ⫇ ⫈ ⫉ ⫊ ⫋ ⫌ ⫍ ⫎ ⫏ ⫐ ⫑ ⫒ ⫓ ⫔ ⫕ ⫖ ⫗ ⫘ ⫙ ⫷ ⫸ ⫹ ⫺ ⊢ ⊣`;
2. All symbols in point 1, preceded by a dot (`.`) to be made elementwise;
3. The operators `<:`, `>:`, `.!`, and `in`, which cannot be preceded by a dot (`.`).

Not all of these have a definition in the standard `Base` library. However, they are available for other packages to define and use as appropriate.

In everyday use, most of these comparison operators are not relevant. The most common ones used are the standard mathematical functions for ordering; see the Syntax section for a list.

Like most other operators in Julia, comparison operators are functions and can be called as functions. For instance, `(<)(1, 2)` is identical in meaning to `1 < 2`.

### Using ==, ===, and isequal

There are three equality operators: `==`, `===`, and `isequal`. (The last is not really an operator, but it is a function and all operators are functions.)

## When to use `==`

`==` is *value* equality. It returns `true` when two objects represent, in their present state, the same value.

For instance, it is obvious that

```
julia> 1 == 1
true
```

but furthermore

```
julia> 1 == 1.0
true

julia> 1 == 1.0 + 0.0im
true

julia> 1 == 1//1
true
```

The right hand sides of each equality above are of a different type, but they still represent the same value.

For mutable objects, like arrays, `==` compares their present value.

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> C = [1, 3, 2]
3-element Array{Int64,1}:
 1
 3
 2

julia> A == B
true

julia> A == C
false
```

```
julia> A[2], A[3] = A[3], A[2]  # swap 2nd and 3rd elements of A
(3,2)

julia> A
3-element Array{Int64,1}:
 1
 3
 2

julia> A == B
false

julia> A == C
true
```

Most of the time, `==` is the right choice.

## When to use `===`

`===` is a far stricter operation than `==`. Instead of value equality, it measures egality. Two objects are egal if they cannot be distinguished from each other by the program itself. Thus we have

```
julia> 1 === 1
true
```

as there is no way to tell a `1` apart from another `1`. But

```
julia> 1 === 1.0
false
```

because although `1` and `1.0` are the same value, they are of different types, and so the program can tell them apart.

Furthermore,

```
julia> A = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> B = [1, 2, 3]
3-element Array{Int64,1}:
 1
 2
 3

julia> A === B
false

julia> A === A
true
```

which may at first seem surprising! How could the program distinguish between the two vectors `A` and `B`? Because vectors are mutable, it could modify `A`, and then it would behave differently from `B`. But no matter how it modifies `A`, `A` will always behave the same as `A` itself. So `A` is egal to `A`, but not egal to `B`.

Continuing along this vein, observe

```
julia> C = A
3-element Array{Int64,1}:
 1
 2
 3

julia> A === C
true
```

By assigning `A` to `C`, we say that `C` has *aliased* `A`. That is, it has become just another name for `A`. Any modifications done to `A` will be observed by `C` also. Therefore, there is no way to tell the difference between `A` and `C`, so they are egal.

## When to use `isequal`

The difference between `==` and `isequal` is very subtle. The biggest difference is in how floating point numbers are handled:

```
julia> NaN == NaN
false
```

This possibly surprising result is defined by the IEEE standard for floating point types (IEEE-754). But this is not useful in some cases, such as sorting. `isequal` is provided for those cases:

```
julia> isequal(NaN, NaN)
true
```

On the flip side of the spectrum, `==` treats IEEE negative zero and positive zero as the same value (also as specified by IEEE-754). These values have distinct representations in memory, however.

```
julia> 0.0
0.0

julia> -0.0
-0.0

julia> 0.0 == -0.0
true
```

Again for sorting purposes, `isequal` distinguishes between them.

```
julia> isequal(0.0, -0.0)
false
```

# Chapter 8: Comprehensions

## Examples

### Array comprehension

## Basic Syntax

Julia's array comprehensions use the following syntax:

```
[expression for element = iterable]
```

Note that as with `for` loops, all of `=`, `in`, and `∈` are accepted for the comprehension.

This is roughly equivalent to creating an empty array and using a `for` loop to `push!` items to it.

```
result = []
for element in iterable
    push!(result, expression)
end
```

however, the type of an array comprehension is as narrow as possible, which is better for performance.

For example, to get an array of the squares of the integers from `1` to `10`, the following code may be used.

```
squares = [x^2 for x=1:10]
```

This is a clean, concise replacement for the longer `for`-loop version.

```
squares = []
for x in 1:10
    push!(squares, x^2)
end
```

### Conditional Array Comprehension

Before the Julia 0.5, there is no way to use conditions inside the array comprehensions. But, it is no longer true. In Julia 0.5 we can use the conditions inside conditions like the following:

```
julia> [x^2 for x in 0:9 if x > 5]
4-element Array{Int64,1}:
 36
 49
 64
 81
```

Source of the above example can be found here.

If we would like to use nested list comprehension:

```
julia>[(x,y) for x=1:5 , y=3:6 if y>4 && x>3 ]
4-element Array{Tuple{Int64,Int64},1}:
 (4,5)
 (5,5)
 (4,6)
 (5,6)
```

## Multidimensional array comprehensions

Nested `for` loops may be used to iterate over a number of unique iterables.

```
result = []
for a = iterable_a
    for b = iterable_b
        push!(result, expression)
    end
end
```

Similarly, multiple iteration specifications may be supplied to an array comprehension.

```
[expression for a = iterable_a, b = iterable_b]
```

For example, the following may be used to generate the Cartesian product of `1:3` and `1:2`.

```
julia> [(x, y) for x = 1:3, y = 1:2]
3×2 Array{Tuple{Int64,Int64},2}:
 (1,1)  (1,2)
 (2,1)  (2,2)
 (3,1)  (3,2)
```

Flattened multidimensional array comprehensions are similar, except that they lose the shape. For example,

```
julia> [(x, y) for x = 1:3 for y = 1:2]
6-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (1, 2)
 (2, 1)
 (2, 2)
 (3, 1)
 (3, 2)
```

is a flattened variant of the above. The syntactic difference is that an additional `for` is used instead of a comma.

## Generator Comprehensions

Generator comprehensions follow a similar format to array comprehensions, but use parentheses

`()` instead of square brackets `[]`.

```
(expression for element = iterable)
```

Such an expression returns a `Generator` object.

```
julia> (x^2 for x = 1:5)
Base.Generator{UnitRange{Int64},##1#2}(#1,1:5)
```

---

# Function Arguments

Generator comprehensions may be provided as the only argument to a function, without the need for an extra set of parentheses.

```
julia> join(x^2 for x = 1:5)
"1491625"
```

However, if more than one argument is provided, the generator comprehension requries its own set of parentheses.

```
julia> join(x^2 for x = 1:5, ", ")
ERROR: syntax: invalid iteration specification

julia> join((x^2 for x = 1:5), ", ")
"1, 4, 9, 16, 25"
```

Read Comprehensions online: https://riptutorial.com/julia-lang/topic/5477/comprehensions

# Chapter 9: Conditionals

## Syntax

- if cond; body; end
- if cond; body; else; body; end
- if cond; body; elseif cond; body; else; end
- if cond; body; elseif cond; body; end
- cond ? iftrue : iffalse
- cond && iftrue
- cond || iffalse
- ifelse(cond, iftrue, iffalse)

## Remarks

All conditional operators and functions involve using boolean conditions (`true` or `false`). In Julia, the type of booleans is `Bool`. Unlike some other languages, other kinds of numbers (like `1` or `0`), strings, arrays, and so forth *cannot* be used directly in conditionals.

Typically, one uses either predicate functions (functions that return a `Bool`) or comparison operators in the condition of a conditional operator or function.

## Examples

### if...else expression

The most common conditional in Julia is the `if`...`else` expression. For instance, below we implement the Euclidean algorithm for computing the greatest common divisor, using a conditional to handle the base case:

```
mygcd(a, b) = if a == 0
    abs(b)
else
    mygcd(b % a, a)
end
```

The `if`...`else` form in Julia is actually an expression, and has a value; the value is the expression in tail position (that is, the last expression) on the branch that is taken. Consider the following sample input:

```
julia> mygcd(0, -10)
10
```

Here, `a` is `0` and `b` is `-10`. The condition `a == 0` is `true`, so the first branch is taken. The returned value is `abs(b)` which is `10`.

```
julia> mygcd(2, 3)
1
```

Here, `a` is `2` and `b` is `3`. The condition `a == 0` is false, so the second branch is taken, and we compute `mygcd(b % a, a)`, which is `mygcd(3 % 2, 2)`. The `%` operator returns the remainder when `3` is divided by `2`, in this case `1`. Thus we compute `mygcd(1, 2)`, and this time `a` is `1` and `b` is `2`. Once again, `a == 0` is false, so the second branch is taken, and we compute `mygcd(b % a, a)`, which is `mygcd(0, 1)`. This time, `a == 0` at last and so `abs(b)` is returned, which gives the result `1`.

## if...else statement

```
name = readline()
if startswith(name, "A")
    println("Your name begins with A.")
else
    println("Your name does not begin with A.")
end
```

Any expression, such as the `if`...`else` expression, can be put in statement position. This ignores its value but still executes the expression for its side effects.

## if statement

Like any other expression, the return value of an `if`...`else` expression can be ignored (and hence discarded). This is generally only useful when the body of the expression has side effects, such as writing to a file, mutating variables, or printing to the screen.

Furthermore, the `else` branch of an `if`...`else` expression is optional. For instance, we can write the following code to output to screen only if a particular condition is met:

```
second = Dates.second(now())
if iseven(second)
    println("The current second, $second, is even.")
end
```

In the example above, we use time and date functions to get the current second; for instance, if it is currently 10:55:27, the variable `second` will hold `27`. If this number is even, then a line will be printed to screen. Otherwise, nothing will be done.

## Ternary conditional operator

```
pushunique!(A, x) = x in A ? A : push!(A, x)
```

The ternary conditional operator is a less wordy `if`...`else` expression.

The syntax specifically is:

```
[condition] ? [execute if true] : [execute if false]
```

In this example, we add `x` to the collection `A` only if `x` is not already in `A`. Otherwise, we just leave `A` unchanged.

Ternary operator References:

- Julia Documentation
- Wikibooks

### Short-circuit operators: && and ||

## For branching

The short-circuiting conditional operators `&&` and `||` can be used as lightweight replacements for the following constructs:

- `x && y` is equivalent to `x ? y : x`
- `x || y` is equivalent to `x ? x : y`

One use for short-circuit operators is as a more concise way to test a condition and perform a certain action depending on that condition. For instance, the following code uses the `&&` operator to throw an error if the argument `x` is negative:

```
function mysqrt(x)
    x < 0 && throw(DomainError("x is negative"))
    x ^ 0.5
end
```

The `||` operator can also be used for error checking, except that it triggers the error *unless* a condition holds, instead of *if* the condition holds:

```
function halve(x::Integer)
    iseven(x) || throw(DomainError("cannot halve an odd number"))
    x ÷ 2
end
```

Another useful application of this is to supply a default value to an object, only if it is not previously defined:

```
isdefined(:x) || (x = NEW_VALUE)
```

Here, this checks if the symbol x is defined (i.e. if there is an value assigned to the object `x`). If so, then nothing happens. But, if not, then `x` will be assigned `NEW_VALUE`. Note that this example will only work at toplevel scope.

## In conditions

The operators are also useful because they can be used to test two conditions, the second of which is only evaluated depending on the result of the first condition. From the Julia

documentation:

> In the expression `a && b`, the subexpression `b` is only evaluated if `a` evaluates to `true`
>
> In the expression `a || b`, the subexpression `b` is only evaluated if `a` evaluates to `false`

Thus, while both `a & b` and `a && b` will yield `true` if both `a` and `b` are `true`, their behavior if `a` is `false` is different.

For instance, suppose we wish to check if an object is a positive number, where it is possible that it might not even be a number. Consider the differences between these two attempted implementations:

```
CheckPositive1(x) = (typeof(x)<:Number) & (x > 0) ? true : false
CheckPositive2(x) = (typeof(x)<:Number) && (x > 0) ? true : false

CheckPositive1("a")
CheckPositive2("a")
```

`CheckPositive1()` will yield an error if a non-numeric type is supplied to it as an argument. This is because it evaluates *both* expressions, regardless of the result of the first, and the second expression will yield an error when one tries to evaluate it for a non-numeric type.

`CheckPositive2()`, however, will yield `false` (rather than an error) if a non-numeric type is supplied to it, since the second expression is only evaluated if the first is `true`.

More than one short-circuit operator can be strung together. E.g.:

```
1 > 0 && 2 > 0 && 3 > 5
```

## if statement with multiple branches

```
d = Dates.dayofweek(now())
if d == 7
    println("It is Sunday!")
elseif d == 6
    println("It is Saturday!")
elseif d == 5
    println("Almost the weekend!")
else
    println("Not the weekend yet...")
end
```

Any number of `elseif` branches may be used with an `if` statement, possibly with or without a final `else` branch. Subsequent conditions will only be evaluated if all prior conditions have been found to be `false`.

## The ifelse function

```
shift(x) = ifelse(x > 10, x + 1, x - 1)
```

Usage:

```
julia> shift(10)
9

julia> shift(11)
12

julia> shift(-1)
-2
```

The `ifelse` function will evaluate both branches, even the one that is not selected. This can be useful either when the branches have side effects that must be evaluated, or because it can be faster if both branches themselves are cheap.

Read Conditionals online: https://riptutorial.com/julia-lang/topic/4356/conditionals

# Chapter 10: Cross-Version Compatibility

## Syntax

- using Compat
- Compat.String
- Compat.UTF8String
- @compat f.(x, y)

## Remarks

It is sometimes very difficult to get new syntax to play well with multiple versions. As Julia is still undergoing active development, it is often useful simply to drop support for older versions and instead target just the newer ones.

## Examples

### Version numbers

Julia has a built-in implementation of semantic versioning exposed through the `VersionNumber` type.

To construct a `VersionNumber` as a literal, the `@v_str` string macro can be used:

```
julia> vers = v"1.2.0"
v"1.2.0"
```

Alternatively, one can call the `VersionNumber` constructor; note that the constructor accepts up to five arguments, but all except the first are optional.

```
julia> vers2 = VersionNumber(1, 1)
v"1.1.0"
```

Version numbers can be compared using comparison operators, and thus can be sorted:

```
julia> vers2 < vers
true

julia> v"1" < v"0"
false

julia> sort([v"1.0.0", v"1.0.0-dev.100", v"1.0.1"])
3-element Array{VersionNumber,1}:
 v"1.0.0-dev.100"
 v"1.0.0"
 v"1.0.1"
```

Version numbers are used in several places across Julia. For instance, the `VERSION` constant is a

`VersionNumber`:

```
julia> VERSION
v"0.5.0"
```

This is commonly used for conditional code evaluation, depending on the Julia version. For example, to run different code on v0.4 and v0.5, one can do

```
if VERSION < v"0.5"
    println("v0.5 prerelease, v0.4 or older")
else
    println("v0.5 or newer")
end
```

Each installed package is also associated with a current version number:

```
julia> Pkg.installed("StatsBase")
v"0.9.0"
```

### Using Compat.jl

The Compat.jl package enables using some new Julia features and syntax with older versions of Julia. Its features are documented on its README, but a summary of useful applications is given below.

0.5.0

## Unified String type

In Julia v0.4, there were many different types of strings. This system was considered overly complex and confusing, so in Julia v0.5, there remains only the `String` type. `Compat` allows using the `String` type and constructor on version 0.4, under the name `Compat.String`. For example, this v0.5 code

```
buf = IOBuffer()
println(buf, "Hello World!")
String(buf)  # "Hello World!\n"
```

can be directly translated to this code, which works on both v0.5 and v0.4:

```
using Compat
buf = IOBuffer()
println(buf, "Hello World!")
Compat.String(buf)  # "Hello World!\n"
```

Note that there are some caveats.

- On v0.4, `Compat.String` is typealiased to `ByteString`, which is `Union{ASCIIString, UTF8String}`. Thus, types with `String` fields will not be type stable. In these situations, `Compat.UTF8String` is

advised, as it will mean `String` on v0.5, and `UTF8String` on v0.4, both of which are concrete types.
- One has to be careful to use `Compat.String` or `import Compat: String`, because `String` itself has a meaning on v0.4: it is a deprecated alias for `AbstractString`. A sign that `String` was accidentally used instead of `Compat.String` is if at any point, the following warnings appear:

```
WARNING: Base.String is deprecated, use AbstractString instead.
  likely near no file:0
WARNING: Base.String is deprecated, use AbstractString instead.
  likely near no file:0
```

## Compact broadcasting syntax

Julia v0.5 introduces syntactic sugar for `broadcast`. The syntax

```
f.(x, y)
```

is lowered to `broadcast(f, x, y)`. Examples of using this syntax include `sin.([1, 2, 3])` to take the sine of multiple numbers at once.

On v0.5, the syntax can be used directly:

```
julia> sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

However, if we try the same on v0.4, we get an error:

```
julia> sin.([1.0, 2.0, 3.0])
ERROR: TypeError: getfield: expected Symbol, got Array{Float64,1}
```

Luckily, `Compat` makes this new syntax usable from v0.4 also. Once again, we add `using Compat`. This time, we surround the expression with the `@compat` macro:

```
julia> using Compat

julia> @compat sin.([1.0, 2.0, 3.0])
3-element Array{Float64,1}:
 0.841471
 0.909297
 0.14112
```

Read Cross-Version Compatibility online: https://riptutorial.com/julia-lang/topic/5832/cross-version-compatibility

# Chapter 11: Dictionaries

## Examples

### Using Dictionaries

Dictionaries can be constructed by passing it any number of pairs.

```
julia> Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1
```

You can get entries in a dictionary putting the key in square brackets.

```
julia> dict = Dict("A"=>1, "B"=>2)
Dict{String,Int64} with 2 entries:
  "B" => 2
  "A" => 1

julia> dict["A"]
1
```

Read Dictionaries online: https://riptutorial.com/julia-lang/topic/9028/dictionaries

# Chapter 12: Enums

## Syntax

- @enum EnumType val=1 val val
- :symbol

## Remarks

It is sometimes useful to have enumerated types where each instance is of a different type (often a singleton immutable type); this can be important for type stability. Traits are typically implemented with this paradigm. However, this results in additional compile-time overhead.

## Examples

### Defining an enumerated type

An enumerated type is a type that can hold one of a finite list of possible values. In Julia, enumerated types are typically called "enum types". For instance, one could use enum types to describe the seven days of the week, the twelve months of the year, the four suits of a standard 52-card deck, or other similar situations.

We can define enumerated types to model the suits and ranks of a standard 52-card deck. The `@enum` macro is used to define enum types.

```
@enum Suit ♣♦♥♠
@enum Rank ace=1 two three four five six seven eight nine ten jack queen king
```

This defines two types: `Suit` and `Rank`. We can check that the values are indeed of the expected types:

```
julia> ♦
♦::Suit = 1

julia> six
six::Rank = 6
```

Note that each suit and rank has been associated with a number. By default, this number starts at zero. So the second suit, diamonds, was assigned the number 1. In the case of `Rank`, it may make more sense to start the number at one. This was achieved by annotating the definition of `ace` with a `=1` annotation.

Enumerated types come with a lot of functionality, such as equality (and indeed identity) and comparisons built in:

```
julia> seven === seven
true

julia> ten ≠ jack
true

julia> two < three
true
```

Like values of any other immutable type, values of enumerated types can also be hashed and stored in `Dict`s.

We can complete this example by defining a `Card` type that has a `Rank` and a `Suit` field:

```
immutable Card
    rank::Rank
    suit::Suit
end
```

and hence we can create cards with

```
julia> Card(three, ♣)
Card(three::Rank = 3,♣::Suit = 0)
```

But enumerated types also come with their own `convert` methods, so we can indeed simply do

```
julia> Card(7, ♠)
Card(seven::Rank = 7,♠::Suit = 3)
```

and since `7` can be directly converted to `Rank`, this constructor works out of the box.

We might wish to define syntactic sugar for constructing these cards; implicit multiplication provides a convenient way to do it. Define

```
julia> import Base.*

julia> r::Int * s::Suit = Card(r, s)
* (generic function with 156 methods)
```

and then

```
julia> 10♣
Card(ten::Rank = 10,♣::Suit = 0)

julia> 5♠
Card(five::Rank = 5,♠::Suit = 3)
```

once again taking advantage of the in-built `convert` functions.

## Using symbols as lightweight enums

Although the `@enum` macro is quite useful for most use cases, it can be excessive in some use cases. Disadvantages of `@enum` include:

- It creates a new type
- It is a little harder to extend
- It comes with functionality such as conversion, enumeration, and comparison, which may be superfluous in some applications

In cases where a lighter-weight alternative is desired, the `Symbol` type can be used. Symbols are interned strings; they represent sequences of characters, much like strings do, but they are uniquely associated with numbers. This unique association enables fast symbol equality comparison.

We may again implement a `Card` type, this time using `Symbol` fields:

```
const ranks = Set([:ace, :two, :three, :four, :five, :six, :seven, :eight, :nine,
                   :ten, :jack, :queen, :king])
const suits = Set([:♣, :♦, :♥, :♠])
immutable Card
  rank::Symbol
  suit::Symbol
  function Card(r::Symbol, s::Symbol)
    r in ranks || throw(ArgumentError("invalid rank: $r"))
    s in suits || throw(ArgumentError("invalid suit: $s"))
    new(r, s)
  end
end
```

We implement the inner constructor to check for any incorrect values passed to the constructor. Unlike in the example using `@enum` types, `Symbol`s can contain any string, and so we must be careful about what kinds of `Symbol`s we accept. Note here the use of the short-circuit conditional operators.

Now we can construct `Card` objects like we expect:

```
julia> Card(:ace, :♦)
Card(:ace,:♦)

julia> Card(:nine, :♠)
Card(:nine,:♠)

julia> Card(:eleven, :♠)
ERROR: ArgumentError: invalid rank: eleven
 in Card(::Symbol, ::Symbol) at ./REPL[17]:5

julia> Card(:king, :X)
ERROR: ArgumentError: invalid suit: X
 in Card(::Symbol, ::Symbol) at ./REPL[17]:6
```

A major benefit of `Symbol`s is their runtime extensibility. If at runtime, we wish to accept (for example) `:eleven` as a new rank, it suffices to simply run `push!(ranks, :eleven)`. Such runtime extensibility is not possible with `@enum` types.

Read Enums online: https://riptutorial.com/julia-lang/topic/7104/enums

# Chapter 13: Expressions

## Examples

### Intro to Expressions

Expressions are a specific type of object in Julia. You can think of an expression as representing a piece of Julia code that has not yet been evaluated (i.e. executed). There are then specific functions and operations, like `eval()` which will evaluate the expression.

For instance, we could write a script or enter into the interpreter the following: julia> 1+1 2

One way to create an expression is using the `:()` syntax. For example:

```
julia> MyExpression = :(1+1)
:(1 + 1)
julia> typeof(MyExpression)
Expr
```

We now have an `Expr` type object. Having just been formed, it doesn't do anything - it just sits around like any other object until it is acted upon. In this case, we can *evaluate* that expression using the `eval()` function:

```
julia> eval(MyExpression)
2
```

Thus, we see that the following two are equivalent:

```
1+1
eval(:(1+1))
```

Why would we want to go through the much more complicated syntax in `eval(:(1+1))` if we just want to find what 1+1 equals? The basic reason is that we can define an expression at one point in our code, potentially modify it later on, and then evaluate it at a later point still. This can potentially open up powerful new capabilities to the Julia programmer. Expressions are a key component of metaprogramming in Julia.

### Creating Expressions

There are a number of different methods that can be used to create the same type of expression. The expressions intro mentioned the `:()` syntax. Perhaps the best place to start, however is with strings. This helps to reveal some of the fundamental similarities between expressions and strings in Julia.

**Create Expression from String**

From the Julia documentation:

> Every Julia program starts life as a string

In other words, any Julia script is simply written in a text file, which is nothing but a string of characters. Likewise, any Julia command entered into an interpreter is just a string of characters. The role of Julia or any other programming language then is to interpret and evaluate strings of characters in a logical, predictable way so that those strings of characters can be used to describe what the programmer wants the computer to accomplish.

Thus, one way to create an expression is to use the `parse()` function as applied to a string. The following expression, once it is evaluated, will assign the value of 2 to the symbol `x`.

```
MyStr = "x = 2"
MyExpr = parse(MyStr)
julia> x
ERROR: UndefVarError: x not defined
eval(MyExpr)
julia> x
2
```

### Create Expression Using `:()` Syntax

```
MyExpr2 = :(x = 2)
julia> MyExpr == MyExpr2
true
```

Note that with this syntax, Julia will automatically treat the names of objects as referring to symbols. We can see this if we look at the `args` of the expression. (See Fields of Expression Objects for more details on the `args` field in an expression.)

```
julia> MyExpr2.args
2-element Array{Any,1}:
  :x
 2
```

### Create Expression using the `Expr()` Function

```
MyExpr3 = Expr(:(=), :x, 2)
MyExpr3 == MyExpr
```

This syntax is based on prefix notation. In other words, the first argument of the specified to the `Expr()` function is the `head` or prefix. The remaining are the `arguments` of the expression. The `head` determines what operations will be performed on the arguments.

For more details on this, see Fields of Expression Objects

When using this syntax, it is important to distinguish between using objects and symbols for objects. For instance, in the above example, the expression assigns the value of `2` to the symbol `:x`, a perfectly sensible operation. If we used `x` itself in an expression such as that, we would get the nonsensical result:

```
julia> Expr(:(=), x, 5)
:(2 = 5)
```

Similarly, if we examine the `args` we see:

```
julia> Expr(:(=), x, 5).args
2-element Array{Any,1}:
 2
 5
```

Thus, the `Expr()` function does not perform the same automatic transformation into symbols as the `:()` syntax for creating expressions.

**Create multi-line Expressions using `quote...end`**

```
MyQuote =
quote
    x = 2
    y = 3
end
julia> typeof(MyQuote)
Expr
```

Note that with `quote...end` we can create expressions that contain other expressions in their `args` field:

```
julia> typeof(MyQuote.args[2])
Expr
```

See Fields of Expression Objects for more on this `args` field.

**More on Creating Expressions**

This Example just gives the basics for creating expressions. See also, for example, Interpolation and Expressions and Fields of Expression Objects for more information on creating more complex and advanced expressions.

## Fields of Expression Objects

As mentioned in the Intro to Expressions expressions are a specific type of object in Julia. As such, they have fields. The two most used fields of an expression are its `head` and its `args`. For instance, consider the expression

```
MyExpr3 = Expr(:(=), :x, 2)
```

discussed in Creating Expressions. We can see the `head` and `args` as follows:

```
julia> MyExpr3.head
:(=)
```

```
julia> MyExpr3.args
2-element Array{Any,1}:
  :x
 2
```

Expressions are based on prefix notation. As such, the `head` generally specifies the operation that is to be performed on the `args`. The head must be of Julia type `Symbol`.

When an expression is to assign a value (when it gets evaluated), it will generally use a head of `:(=)`. There are of course obvious variations to this that can be employed, e.g.:

```
ex1 = Expr(:(+=), :x, 2)
```

### :call for expression heads

Another common `head` for expressions is `:call`. E.g.

```
ex2 = Expr(:call, :(*), 2, 3)
eval(ex2) ## 6
```

Following the conventions of prefix notation, operators are evaluated from left to right. Thus, this expression here means that we will call the function that is specified on the first element of `args` on the subsequent elements. We similarly could have:

```
julia> ex2a = Expr(:call, :(-), 1, 2, 3)
:(1 - 2 - 3)
```

Or other, potentially more interesting functions, e.g.

```
julia> ex2b = Expr(:call, :rand, 2,2)
:(rand(2,2))

julia> eval(ex2b)
2x2 Array{Float64,2}:
 0.429397  0.164478
 0.104994  0.675745
```

### Automatic determination of `head` when using `:()` expression creation notation

Note that `:call` is implicitly used as the head in certain constructions of expressions, e.g.

```
julia> :(x + 2).head
:call
```

Thus, with the `:()` syntax for creating expressions, Julia will seek to automatically determine the correct head to use. Similarly:

```
julia> :(x = 2).head
:(=)
```

In fact, if you aren't certain what the right head to use for an expression that you are forming using, for instance, `Expr()` this can be a helpful tool to get tips and ideas for what to use.

## Interpolation and Expressions

Creating Expressions mentions that expressions are closely related to strings. As such, the principles of interpolation within strings are also relevant for Expressions. For instance, in basic string interpolation, we can have something like:

```
n = 2
julia> MyString = "there are $n ducks"
"there are 2 ducks"
```

We use the `$` sign to insert the value of `n` into the string. We can use the same technique with expressions. E.g.

```
a = 2
ex1 = :(x = 2*$a)   ##      :(x = 2 * 2)
a = 3
eval(ex1)
x # 4
```

Contrast this this:

```
a = 2
ex2 = :(x = 2*a) # :(x = 2a)
a = 3
eval(ex2)
x # 6
```

Thus, with the first example, we set in advance the value of `a` that will be used at the time that the expression is evaluated. With the second example, however, the Julia compiler will only look to `a` to find its value *at the time of evaluation* for our expression.

## External References on Expressions

There are a number of useful web resources that can help further your knowledge of expressions in Julia. These include:

- Julia Docs - Metaprogramming
- Wikibooks - Julia Metaprogramming
- Julia's macros, expressions, etc. for and by the confused, by Gray Calhoun
- Month of Julia - Metaprogramming, by Andrew Collier
- Symbolic Differentiation in Julia, by John Myles White

SO Posts:

- What is a "symbol" in Julia? Answer by Stefan Karpinski
- Why does julia express this expression in this complex way?
- Explanation of Julia expression interpolation example

# Chapter 14: for Loops

## Syntax

- for i in iter; ...; end
- while cond; ...; end
- break
- continue
- @parallel (op) for i in iter; ...; end
- @parallel for i in iter; ...; end
- @goto label
- @label label

## Remarks

Whenever it makes code shorter and easier to read, consider using higher-order functions, such as `map` or `filter`, instead of loops.

## Examples

### Fizz Buzz

A common use case for a `for` loop is to iterate over a predefined range or collection, and do the same task for all its elements. For instance, here we combine a `for` loop with a conditional `if-elseif-else` statement:

```
for i in 1:100
    if i % 15 == 0
        println("FizzBuzz")
    elseif i % 3 == 0
        println("Fizz")
    elseif i % 5 == 0
        println("Buzz")
    else
        println(i)
    end
end
```

This is the classic Fizz Buzz interview question. The (truncated) output is:

```
1
2
Fizz
4
Buzz
Fizz
7
8
```

## Find smallest prime factor

In some situations, one might want to return from a function before finishing an entire loop. The `return` statement can be used for this.

```
function primefactor(n)
    for i in 2:n
        if n % i == 0
            return i
        end
    end
    @assert false  # unreachable
end
```

Usage:

```
julia> primefactor(100)
2

julia> primefactor(97)
97
```

Loops can also be terminated early with the `break` statement, which terminates just the enclosing loop instead of the entire function.

## Multidimensional iteration

In Julia, a for loop can contain a comma (`,`) to specify iterating over multiple dimensions. This acts similarly to nesting a loop within another, but can be more compact. For instance, the below function generates elements of the Cartesian product of two iterables:

```
function cartesian(xs, ys)
    for x in xs, y in ys
        produce(x, y)
    end
end
```

Usage:

```
julia> collect(@task cartesian(1:2, 1:4))
8-element Array{Tuple{Int64,Int64},1}:
 (1,1)
 (1,2)
 (1,3)
 (1,4)
 (2,1)
 (2,2)
 (2,3)
 (2,4)
```

However, indexing over arrays of any dimension should be done with `eachindex`, not with a multidimensional loop (if possible):

```
s = zero(eltype(A))
for ind in eachindex(A)
    s += A[ind]
end
```

## Reduction and parallel loops

Julia provides macros to simplify distributing computation across multiple machines or workers. For instance, the following computes the sum of some number of squares, possibly in parallel.

```
function sumofsquares(A)
    @parallel (+) for i in A
        i ^ 2
    end
end
```

Usage:

```
julia> sumofsquares(1:10)
385
```

For more on this topic, see the example on `@parallel` within the Parallel Processesing topic.

Read for Loops online: https://riptutorial.com/julia-lang/topic/4355/for-loops

# Chapter 15: Functions

## Syntax

- f(n) = ...
- function f(n) ... end
- n::Type
- x -> ...
- f(n) do ... end

## Remarks

Aside from generic functions (which are most common), there are also built-in functions. Such functions include `is`, `isa`, `typeof`, `throw`, and similar functions. Built-in functions are typically implemented in C instead of Julia, so they cannot be specialized on argument types for dispatch.

## Examples

### Square a number

This is the easiest syntax to define a function:

```
square(n) = n * n
```

To call a function, use round brackets (without spaces in between):

```
julia> square(10)
100
```

Functions are objects in Julia, and we can show them in the REPL as with any other objects:

```
julia> square
square (generic function with 1 method)
```

All Julia functions are generic (otherwise known as polymorphic) by default. Our `square` function works just as well with floating point values:

```
julia> square(2.5)
6.25
```

...or even matrices:

```
julia> square([2 4
               2 1])
2×2 Array{Int64,2}:
```

```
 12  12
  6   9
```

### Recursive functions

## Simple recursion

Using recursion and the ternary conditional operator, we can create an alternative implementation of the built-in `factorial` function:

```
myfactorial(n) = n == 0 ? 1 : n * myfactorial(n - 1)
```

Usage:

```
julia> myfactorial(10)
3628800
```

## Working with trees

Recursive functions are often most useful on data structures, especially tree data structures. Since expressions in Julia are tree structures, recursion can be quite useful for metaprogramming. For instance, the below function gathers a set of all heads used in an expression.

```
heads(ex::Expr) = reduce(∪, Set((ex.head,)), (heads(a) for a in ex.args))
heads(::Any) = Set{Symbol}()
```

We can check that our function is working as intended:

```
julia> heads(:(7 + 4x > 1 > A[0]))
Set(Symbol[:comparison,:ref,:call])
```

This function is compact and uses a variety of more advanced techniques, such as the `reduce` higher order function, the `Set` data type, and generator expressions.

### Introduction to Dispatch

We can use the `::` syntax to dispatch on the type of an argument.

```
describe(n::Integer) = "integer $n"
describe(n::AbstractFloat) = "floating point $n"
```

Usage:

```
julia> describe(10)
"integer 10"

julia> describe(1.0)
```

```
"floating point 1.0"
```

Unlike many languages, which typically provide either static multiple dispatch or dynamic single dispatch, Julia has full dynamic multiple dispatch. That is, functions can be specialized for more than one argument. This comes in handy when defining specialized methods for operations on certain types, and fallback methods for other types.

```
describe(n::Integer, m::Integer) = "integers n=$n and m=$m"
describe(n, m::Integer) = "only m=$m is an integer"
describe(n::Integer, m) = "only n=$n is an integer"
```

Usage:

```
julia> describe(10, 'x')
"only n=10 is an integer"

julia> describe('x', 10)
"only m=10 is an integer"

julia> describe(10, 10)
"integers n=10 and m=10"
```

## Optional Arguments

Julia allows functions to take optional arguments. Behind the scenes, this is implemented as another special case of multiple dispatch. For instance, let's solve the popular Fizz Buzz problem. By default, we will do it for numbers in the range `1:10`, but we will allow a different value if necessary. We will also allow different phrases to be used for `Fizz` or `Buzz`.

```
function fizzbuzz(xs=1:10, fizz="Fizz", buzz="Buzz")
    for i in xs
        if i % 15 == 0
            println(fizz, buzz)
        elseif i % 3 == 0
            println(fizz)
        elseif i % 5 == 0
            println(buzz)
        else
            println(i)
        end
    end
end
```

If we inspect `fizzbuzz` in the REPL, it says that there are four methods. One method was created for each combination of arguments allowed.

```
julia> fizzbuzz
fizzbuzz (generic function with 4 methods)

julia> methods(fizzbuzz)
# 4 methods for generic function "fizzbuzz":
fizzbuzz() at REPL[96]:2
```

```
fizzbuzz(xs) at REPL[96]:2
fizzbuzz(xs, fizz) at REPL[96]:2
fizzbuzz(xs, fizz, buzz) at REPL[96]:2
```

We can verify that our default values are used when no parameters are provided:

```
julia> fizzbuzz()
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
```

but that the optional parameters are accepted and respected if we provide them:

```
julia> fizzbuzz(5:8, "fuzz", "bizz")
bizz
fuzz
7
8
```

# Parametric Dispatch

It is frequently the case that a function should dispatch on parametric types, such as `Vector{T}` or `Dict{K,V}`, but the type parameters are not fixed. This case can be dealt with by using parametric dispatch:

```
julia> foo{T<:Number}(xs::Vector{T}) = @show xs .+ 1
foo (generic function with 1 method)

julia> foo(xs::Vector) = @show xs
foo (generic function with 2 methods)

julia> foo([1, 2, 3])
xs .+ 1 = [2,3,4]
3-element Array{Int64,1}:
 2
 3
 4

julia> foo([1.0, 2.0, 3.0])
xs .+ 1 = [2.0,3.0,4.0]
3-element Array{Float64,1}:
 2.0
 3.0
 4.0

julia> foo(["x", "y", "z"])
xs = String["x","y","z"]
3-element Array{String,1}:
```

```
 "x"
 "y"
 "z"
```

One may be tempted to simply write `xs::Vector{Number}`. But this only works for objects whose type is explicitly `Vector{Number}`:

```
julia> isa(Number[1, 2], Vector{Number})
true

julia> isa(Int[1, 2], Vector{Number})
false
```

This is due to parametric invariance: the object `Int[1, 2]` is *not* a `Vector{Number}`, because it can only contain `Int`s, whereas a `Vector{Number}` would be expected to be able to contain any kinds of numbers.

## Writing Generic Code

Dispatch is an incredibly powerful feature, but frequently it is better to write generic code that works for all types, instead of specializing code for each type. Writing generic code avoids code duplication.

For example, here is code to compute the sum of squares of a vector of integers:

```
function sumsq(v::Vector{Int})
    s = 0
    for x in v
        s += x ^ 2
    end
    s
end
```

But this code *only* works for a vector of `Int`s. It will not work on a `UnitRange`:

```
julia> sumsq(1:10)
ERROR: MethodError: no method matching sumsq(::UnitRange{Int64})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
```

It will not work on a `Vector{Float64}`:

```
julia> sumsq([1.0, 2.0])
ERROR: MethodError: no method matching sumsq(::Array{Float64,1})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
```

A better way to write this `sumsq` function should be

```
function sumsq(v::AbstractVector)
    s = zero(eltype(v))
```

```
    for x in v
        s += x ^ 2
    end
    s
end
```

This will work on the two cases listed above. But there are some collections that we might want to sum the squares of that aren't vectors at all, in any sense. For instance,

```
julia> sumsq(take(countfrom(1), 100))
ERROR: MethodError: no method matching sumsq(::Base.Take{Base.Count{Int64}})
Closest candidates are:
  sumsq(::Array{Int64,1}) at REPL[8]:2
  sumsq(::AbstractArray{T,1}) at REPL[11]:2
```

shows that we cannot sum the squares of a lazy iterable.

An even more generic implementation is simply

```
function sumsq(v)
    s = zero(eltype(v))
    for x in v
        s += x ^ 2
    end
    s
end
```

Which works in all cases:

```
julia> sumsq(take(countfrom(1), 100))
338350
```

This is the most idiomatic Julia code, and can handle all sorts of situations. In some other languages, removing type annotations may affect performance, but that is not the case in Julia; only type stability is important for performance.

## Imperative factorial

A long-form syntax is available for defining multi-line functions. This can be useful when we use imperative structures such as loops. The expression in tail position is returned. For instance, the below function uses a `for` loop to compute the factorial of some integer `n`:

```
function myfactorial(n)
    fact = one(n)
    for m in 1:n
        fact *= m
    end
    fact
end
```

Usage:

```
julia> myfactorial(10)
3628800
```

In longer functions, it is common to see the `return` statement used. The `return` statement is not necessary in tail position, but it is still sometimes used for clarity. For instance, another way of writing the above function would be

```
function myfactorial(n)
    fact = one(n)
    for m in 1:n
        fact *= m
    end
    return fact
end
```

which is identical in behaviour to the function above.

### Anonymous functions

## Arrow syntax

Anonymous functions can be created using the `->` syntax. This is useful for passing functions to higher-order functions, such as the `map` function. The below function computes the square of each number in an array `A`.

```
squareall(A) = map(x -> x ^ 2, A)
```

An example of using this function:

```
julia> squareall(1:10)
10-element Array{Int64,1}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

## Multiline syntax

Multiline anonymous functions can be created using `function` syntax. For instance, the following example computes the factorials of the first `n` numbers, but using an anonymous function instead of the built in `factorial`.

```
julia> map(function (n)
               product = one(n)
```

```
            for i in 1:n
                product *= i
            end
            product
        end, 1:10)
10-element Array{Int64,1}:
       1
       2
       6
      24
     120
     720
    5040
   40320
  362880
 3628800
```

## Do block syntax

Because it is so common to pass an anonymous function as the first argument to a function, there is a `do` block syntax. The syntax

```
map(A) do x
    x ^ 2
end
```

is equivalent to

```
map(x -> x ^ 2, A)
```

but the former can be more clear in many situations, especially if a lot of computation is being done in the anonymous function. `do` block syntax is especially useful for file input and output for resource management reasons.

Read Functions online: https://riptutorial.com/julia-lang/topic/3079/functions

# Chapter 16: Higher-Order Functions

## Syntax

- foreach(f, xs)
- map(f, xs)
- filter(f, xs)
- reduce(f, v0, xs)
- foldl(f, v0, xs)
- foldr(f, v0, xs)

## Remarks

Functions can be accepted as parameters and can also be produced as return types. Indeed, functions can be created inside the body of other functions. These inner functions are known as closures.

## Examples

### Functions as arguments

Functions are objects in Julia. Like any other objects, they can be passed as arguments to other functions. Functions that accept functions are known as higher-order functions.

For instance, we can implement an equivalent of the standard library's `foreach` function by taking a function `f` as the first parameter.

```
function myforeach(f, xs)
    for x in xs
        f(x)
    end
end
```

We can test that this function indeed works as we expect:

```
julia> myforeach(println, ["a", "b", "c"])
a
b
c
```

By taking a function as the *first* parameter, instead of a later parameter, we can use Julia's do block syntax. The do block syntax is just a convenient way to pass an anonymous function as the first argument to a function.

```
julia> myforeach([1, 2, 3]) do x
           println(x^x)
```

```
    end
1
4
27
```

Our implementation of `myforeach` above is roughly equivalent to the built-in `foreach` function. Many other built-in higher order functions also exist.

Higher-order functions are quite powerful. Sometimes, when working with higher-order functions, the exact operations being performed become unimportant and programs can become quite abstract. Combinators are examples of systems of highly abstract higher-order functions.

## Map, filter, and reduce

Two of the most fundamental higher-order functions included in the standard library are `map` and `filter`. These functions are generic and can operate on any iterable. In particular, they are well-suited for computations on arrays.

Suppose we have a dataset of schools. Each school teaches a particular subject, has a number of classes, and an average number of students per class. We can model a school with the following immutable type:

```
immutable School
    subject::Symbol
    nclasses::Int
    nstudents::Int  # average no. of students per class
end
```

Our dataset of schools will be a `Vector{School}`:

```
dataset = [School(:math, 3, 30), School(:math, 5, 20), School(:science, 10, 5)]
```

Suppose we wish to find the number of students in total enrolled in a math program. To do this, we require several steps:

- we must narrow the dataset down to only schools that teach math (`filter`)
- we must compute the number of students at each school (`map`)
- and we must reduce that list of numbers of students to a single value, the sum (`reduce`)

A naïve (not most performant) solution would simply be to use those three higher-order functions directly.

```
function nmath(data)
    maths = filter(x -> x.subject === :math, data)
    students = map(x -> x.nclasses * x.nstudents, maths)
    reduce(+, 0, students)
end
```

and we verify there are 190 math students in our dataset:

```
julia> nmath(dataset)
190
```

Functions exist to combine these functions and thus improve performance. For instance, we could have used the `mapreduce` function to perform the mapping and reduction in one step, which would save time and memory.

The `reduce` is only meaningful for associative operations like `+`, but occasionally it is useful to perform a reduction with a non-associative operation. The higher-order functions `foldl` and `foldr` are provided to force a particular reduction order.

Read Higher-Order Functions online: https://riptutorial.com/julia-lang/topic/6955/higher-order-functions

# Chapter 17: Input

## Syntax

- readline()
- readlines()
- readstring(STDIN)
- chomp(str)
- open(f, file)
- eachline(io)
- readstring(file)
- read(file)
- readcsv(file)
- readdlm(file)

## Parameters

| Parameter | Details |
|---|---|
| **`chomp(str)`** | **Remove up to one trailing newline from a string.** |
| `str` | The string to strip a trailing newline from. Note that strings are immutable by convention. This function returns a new string. |
| **`open(f, file)`** | **Open a file, call the function, and close the file afterward.** |
| `f` | The function to call on the IO stream opening the file generates. |
| `file` | The path of the file to open. |

## Examples

### Reading a String from Standard Input

The `STDIN` stream in Julia refers to standard input. This can represent either user input, for interactive command-line programs, or input from a file or pipeline that has been redirected into the program.

The `readline` function, when not provided any arguments, will read data from `STDIN` until a newline is encountered, or the `STDIN` stream enters the end-of-file state. These two cases can be distinguished by whether the `\n` character has been read as the final character:

```
julia> readline()
some stuff
```

```
"some stuff\n"

julia> readline()  # Ctrl-D pressed to send EOF signal here
""
```

Often, for interactive programs, we do not care about the EOF state, and just want a string. For instance, we may prompt the user for input:

```
function askname()
    print("Enter your name: ")
    readline()
end
```

This is not quite satisfactory, however, because of the additional newline:

```
julia> askname()
Enter your name: Julia
"Julia\n"
```

The `chomp` function is available to remove up to one trailing newline off a string. For example:

```
julia> chomp("Hello, World!")
"Hello, World!"

julia> chomp("Hello, World!\n")
"Hello, World!"
```

We may therefore augment our function with `chomp` so that the result is as expected:

```
function askname()
    print("Enter your name: ")
    chomp(readline())
end
```

which has a more desirable result:

```
julia> askname()
Enter your name: Julia
"Julia"
```

Sometimes, we may wish to read as many lines as is possible (until the input stream enters the end-of-file state). The `readlines` function provides that capability.

```
julia> readlines()  # note Ctrl-D is pressed after the last line
A, B, C, D, E, F, G
H, I, J, K, LMNO, P
Q, R, S
T, U, V
W, X
Y, Z
6-element Array{String,1}:
 "A, B, C, D, E, F, G\n"
 "H, I, J, K, LMNO, P\n"
```

```
 "Q, R, S\n"
 "T, U, V\n"
 "W, X\n"
 "Y, Z\n"
```

0.5.0

Once again, if we dislike the newlines at the end of lines read by `readlines`, we can use the `chomp` function to remove them. This time, we broadcast the `chomp` function across the entire array:

```
julia> chomp.(readlines())
A, B, C, D, E, F, G
H, I, J, K, LMNO, P
Q, R, S
T, U, V
W, X
Y, Z
6-element Array{String,1}:
 "A, B, C, D, E, F, G"
 "H, I, J, K, LMNO, P"
 "Q, R, S"
 "T, U, V"
 "W, X  "
 "Y, Z"
```

Other times, we may not care about lines at all, and simply want to read as much as possible as a single string. The `readstring` function accomplishes this:

```
julia> readstring(STDIN)
If music be the food of love, play on,
Give me excess of it; that surfeiting,
The appetite may sicken, and so die.  # [END OF INPUT]
"If music be the food of love, play on,\nGive me excess of it; that surfeiting,\nThe appetite
may sicken, and so die.\n"
```

(the `# [END OF INPUT]` is not part of the original input; it has been added for clarity.)

Note that `readstring` must be passed the `STDIN` argument.

## Reading Numbers from Standard Input

Reading numbers from standard input is a combination of reading strings and parsing such strings as numbers.

The `parse` function is used to parse a string into the desired number type:

```
julia> parse(Int, "17")
17

julia> parse(Float32, "-3e6")
-3.0f6
```

The format expected by `parse(T, x)` is similar to, but not exactly the same, as the format Julia

expects from number literals:

```
julia> -00000023
-23

julia> parse(Int, "-00000023")
-23

julia> 0x23 |> Int
35

julia> parse(Int, "0x23")
35

julia> 1_000_000
1000000

julia> parse(Int, "1_000_000")
ERROR: ArgumentError: invalid base 10 digit '_' in "1_000_000"
 in tryparse_internal(::Type{Int64}, ::String, ::Int64, ::Int64, ::Int64, ::Bool) at
./parse.jl:88
 in parse(::Type{Int64}, ::String) at ./parse.jl:152
```

Combining the `parse` and `readline` functions allows us to read a single number from a line:

```
function asknumber()
    print("Enter a number: ")
    parse(Float64, readline())
end
```

which works as expected:

```
julia> asknumber()
Enter a number: 78.3
78.3
```

The usual caveats about floating-point precision apply. Note that `parse` can be used with `BigInt` and `BigFloat` to remove or minimize loss of precision.

Sometimes, it is useful to read more than one number from the same line. Typically, the line can be split with whitespace:

```
function askints()
    print("Enter some integers, separated by spaces: ")
    [parse(Int, x) for x in split(readline())]
end
```

which can be used as follows:

```
julia> askints()
Enter some integers, separated by spaces: 1 2 3 4
4-element Array{Int64,1}:
 1
 2
```

```
3
4
```

### Reading Data from a File

## Reading strings or bytes

Files can be opened for reading using the `open` function, which is often used together with do block syntax:

```
open("myfile") do f
    for (i, line) in enumerate(eachline(f))
        print("Line $i: $line")
    end
end
```

Suppose `myfile` exists and its contents are

```
What's in a name? That which we call a rose
By any other name would smell as sweet.
```

Then, this code would produce the following result:

```
Line 1: What's in a name? That which we call a rose
Line 2: By any other name would smell as sweet.
```

Note that `eachline` is a lazy iterable over the lines of the file. It is preferred to `readlines` for performance reasons.

Because `do` block syntax is just syntactic sugar for anonymous functions, we can pass named functions to `open` too:

```
julia> open(readstring, "myfile")
"What's in a name? That which we call a rose\nBy any other name would smell as sweet.\n"

julia> open(read, "myfile")
84-element Array{UInt8,1}:
 0x57
 0x68
 0x61
 0x74
 0x27
 0x73
 0x20
 0x69
 0x6e
 0x20
    ⋮
 0x73
 0x20
 0x73
 0x77
```

```
0x65
0x65
0x74
0x2e
0x0a
```

The functions `read` and `readstring` provide convenience methods that will open a file automatically:

```
julia> readstring("myfile")
"What's in a name? That which we call a rose\nBy any other name would smell as sweet.\n"
```

## Reading structured data

Suppose we had a CSV file with the following contents, in a file named `file.csv`:

```
Make,Model,Price
Foo,2015A,8000
Foo,2015B,14000
Foo,2016A,10000
Foo,2016B,16000
Bar,2016Q,20000
```

Then we may use the `readcsv` function to read this data into a `Matrix`:

```
julia> readcsv("file.csv")
6×3 Array{Any,2}:
 "Make"  "Model"       "Price"
 "Foo"   "2015A"   8000
 "Foo"   "2015B"  14000
 "Foo"   "2016A"  10000
 "Foo"   "2016B"  16000
 "Bar"   "2016Q"  20000
```

If the file were instead delimited with tabs, in a file named `file.tsv`, then the `readdlm` function can be used instead, with the `delim` argument set to `'\t'`. More advanced workloads should use the CSV.jl package.

Read Input online: https://riptutorial.com/julia-lang/topic/7201/input

# Chapter 18: Iterables

## Syntax

- start(itr)
- next(itr, s)
- done(itr, s)
- take(itr, n)
- drop(itr, n)
- cycle(itr)
- Base.product(xs, ys)

## Parameters

| Parameter | Details |
|---|---|
| **For** | **All Functions** |
| `itr` | The iterable to operate on. |
| **For** | **`next` and `done`** |
| `s` | An iterator state describing the current position of the iteration. |
| **For** | **`take` and `drop`** |
| `n` | The number of elements to take or drop. |
| **For** | **`Base.product`** |
| `xs` | The iterable to take first elements of pairs from. |
| `ys` | The iterable to take second elements of pairs from. |
| ... | (Note that `product` accepts any number of arguments; if more than two are provided, it will construct tuples of length greater than two.) |

## Examples

### New iterable type

In Julia, when looping through an iterable object `I` is done with the `for` syntax:

```
for i = I   # or  "for i in I"
    # body
```

```
end
```

Behind the scenes, this is translated to:

```
state = start(I)
while !done(I, state)
    (i, state) = next(I, state)
    # body
end
```

Therefore, if you want `I` to be an iterable, you need to define `start`, `next` and `done` methods for its type. Suppose you define a type `Foo` containing an array as one of the fields:

```
type Foo
    bar::Array{Int,1}
end
```

We instantiate a `Foo` object by doing:

```
julia> I = Foo([1,2,3])
Foo([1,2,3])

julia> I.bar
3-element Array{Int64,1}:
 1
 2
 3
```

If we want to iterate through `Foo`, with each element `bar` being returned by each iteration, we define the methods:

```
import Base: start, next, done

start(I::Foo) = 1

next(I::Foo, state) = (I.bar[state], state+1)

function done(I::Foo, state)
    if state == length(I.bar)
        return true
    end
    return false
end
```

Note that since these functions belong to the `Base` module, we must first `import` their names before adding new methods to them.

After the methods are defined, `Foo` is compatible with the iterator interface:

```
julia> for i in I
           println(i)
       end
```

```
1
2
3
```

### Combining Lazy Iterables

The standard library comes with a rich collection of lazy iterables (and libraries such as Iterators.jl provide even more). Lazy iterables can be composed to create more powerful iterables in constant time. The most important lazy iterables are take and drop, from which many other functions can be created.

## Lazily slice an iterable

Arrays can be sliced with slice notation. For instance, the following returns the 10th to 15th elements of an array, inclusive:

```
A[10:15]
```

However, slice notation does not work with all iterables. For instance, we cannot slice a generator expression:

```
julia> (i^2 for i in 1:10)[3:5]
ERROR: MethodError: no method matching getindex(::Base.Generator{UnitRange{Int64},##1#2},
::UnitRange{Int64})
```

Slicing strings may not have the expected Unicode behaviour:

```
julia> "aaaa"[2:3]
ERROR: UnicodeError: invalid character index
 in getindex(::String, ::UnitRange{Int64}) at ./strings/string.jl:130

julia> "aaaa"[3:4]
"a"
```

We can define a function `lazysub(itr, range::UnitRange)` to do this kind of slicing on arbitrary iterables. This is defined in terms of `take` and `drop`:

```
lazysub(itr, r::UnitRange) = take(drop(itr, first(r) - 1), last(r) - first(r) + 1)
```

The implementation here works because for `UnitRange` value `a:b`, the following steps are performed:

- drops the first `a-1` elements
- takes the `a`th element, `a+1`th element, and so forth, until the `a+(b-a)=b`th element

In total, `b-a` elements are taken. We can confirm our implementation is correct in each case above:

```
julia> collect(lazysub("aaaa", 2:3))
2-element Array{Char,1}:
```

```
 'a'
 'a'

julia> collect(lazysub((i^2 for i in 1:10), 3:5))
3-element Array{Int64,1}:
  9
 16
 25
```

## Lazily shift an iterable circularly

The `circshift` operation on arrays will shift the array as if it were a circle, then relinearize it. For example,

```
julia> circshift(1:10, 3)
10-element Array{Int64,1}:
  8
  9
 10
  1
  2
  3
  4
  5
  6
  7
```

Can we do this lazily for all iterables? We can use the `cycle`, `drop`, and `take` iterables to implement this functionality.

```
lazycircshift(itr, n) = take(drop(cycle(itr), length(itr) - n), length(itr))
```

Along with lazy types being more performant in many situations, this lets us do `circshift`-like functionality on types that would otherwise not support it:

```
julia> circshift("Hello, World!", 3)
ERROR: MethodError: no method matching circshift(::String, ::Int64)
Closest candidates are:
  circshift(::AbstractArray{T,N}, ::Real) at abstractarraymath.jl:162
  circshift(::AbstractArray{T,N}, ::Any) at abstractarraymath.jl:195

julia> String(collect(lazycircshift("Hello, World!", 3)))
"ld!Hello, Wor"
```

0.5.0

## Making a multiplication table

Let's make a multiplication table using lazy iterable functions to create a matrix.

The key functions to use here are:

- `Base.product`, which computes a Cartesian product.
- `prod`, which computes a regular product (as in multiplication)
- `:`, which creates a range
- `map`, which is a higher order function applying a function to each element of a collection

The solution is:

```
julia> map(prod, Base.product(1:10, 1:10))
10×10 Array{Int64,2}:
  1   2   3   4   5   6   7   8   9   10
  2   4   6   8  10  12  14  16  18   20
  3   6   9  12  15  18  21  24  27   30
  4   8  12  16  20  24  28  32  36   40
  5  10  15  20  25  30  35  40  45   50
  6  12  18  24  30  36  42  48  54   60
  7  14  21  28  35  42  49  56  63   70
  8  16  24  32  40  48  56  64  72   80
  9  18  27  36  45  54  63  72  81   90
 10  20  30  40  50  60  70  80  90  100
```

## Lazily-Evaluated Lists

It's possible to make a simple lazily-evaluated list using mutable types and closures. A lazily-evaluated list is a list whose elements are not evaluated when it's constructed, but rather when it is accessed. Benefits of lazily evaluated lists include the possibility of being infinite.

```
import Base: getindex
type Lazy
    thunk
    value
    Lazy(thunk) = new(thunk)
end

evaluate!(lazy::Lazy) = (lazy.value = lazy.thunk(); lazy.value)
getindex(lazy::Lazy) = isdefined(lazy, :value) ? lazy.value : evaluate!(lazy)

import Base: first, tail, start, next, done, iteratorsize, HasLength, SizeUnknown
abstract List
immutable Cons <: List
    head
    tail::Lazy
end
immutable Nil <: List end

macro cons(x, y)
    quote
        Cons($(esc(x)), Lazy(() -> $(esc(y))))
    end
end

first(xs::Cons) = xs.head
tail(xs::Cons) = xs.tail[]
start(xs::Cons) = xs
next(::Cons, xs) = first(xs), tail(xs)
done(::List, ::Cons) = false
done(::List, ::Nil) = true
```

```
iteratorsize(::Nil) = HasLength()
iteratorsize(::Cons) = SizeUnknown()
```

Which indeed works as it would in a language like Haskell, where all lists are lazily-evaluated:

```
julia> xs = @cons(1, ys)
Cons(1,Lazy(false,#3,#undef))

julia> ys = @cons(2, xs)
Cons(2,Lazy(false,#5,#undef))

julia> [take(xs, 5)...]
5-element Array{Int64,1}:
 1
 2
 1
 2
 1
```

In practice, it is better to use the Lazy.jl package. However, the implementation of the lazy list above sheds lights into important details about how to construct one's own iterable type.

Read Iterables online: https://riptutorial.com/julia-lang/topic/5466/iterables

# Chapter 19: JSON

## Syntax

- using JSON
- JSON.parse(str)
- JSON.json(obj)
- JSON.print(io, obj, indent)

## Remarks

Since neither Julia `Dict` nor JSON objects are inherently ordered, it's best not to rely on the order of key-value pairs in a JSON object.

## Examples

### Installing JSON.jl

JSON is a popular data interchange format. The most popular JSON library for Julia is JSON.jl. To install this package, use the package manager:

```
julia> Pkg.add("JSON")
```

The next step is to test whether the package is working on your machine:

```
julia> Pkg.test("JSON")
```

If all tests passed, then the library is ready for use.

### Parsing JSON

JSON that has been encoded as a string can easily be parsed into a standard Julia type:

```
julia> using JSON

julia> JSON.parse("""{
           "this": ["is", "json"],
           "numbers": [85, 16, 12.0],
           "and": [true, false, null]
       }""")
Dict{String,Any} with 3 entries:
  "this"    => Any["is","json"]
  "numbers" => Any[85,16,12.0]
  "and"     => Any[true,false,nothing]
```

There are a few immediate properties of JSON.jl of note:

- JSON types map to sensible types in Julia: Object becomes `Dict`, array becomes `Vector`, number becomes `Int64` or `Float64`, boolean becomes `Bool`, and null becomes `nothing::Void`.
- JSON is an untyped container format: Thus returned Julia vectors are of type `Vector{Any}`, and returned dictionaries are of type `Dict{String, Any}`.
- JSON standard does not distinguish between integers and decimal numbers, but JSON.jl does. A number without a decimal point or scientific notation is parsed into `Int64`, whereas a number with a decimal point is parsed into `Float64`. This matches closely with the behavior of JSON parsers in many other languages.

## Serializing JSON

The `JSON.json` function serializes a Julia object into a Julia `String` containing JSON:

```
julia> using JSON

julia> JSON.json(Dict(:a => :b, :c => [1, 2, 3.0], :d => nothing))
"{\"c\":[1.0,2.0,3.0],\"a\":\"b\",\"d\":null}"

julia> println(ans)
{"c":[1.0,2.0,3.0],"a":"b","d":null}
```

If a string is not desired, JSON can be printed directly to an IO stream:

```
julia> JSON.print(STDOUT, [1, 2, true, false, "x"])
[1,2,true,false,"x"]
```

Note that `STDOUT` is the default, and can be omitted in the above call.

Prettier printing can be achieved by passing the optional `indent` parameter:

```
julia> JSON.print(STDOUT, Dict(:a => :b, :c => :d), 4)
{
    "c": "d",
    "a": "b"
}
```

There is a sane default serialization for complex Julia types:

```
julia> immutable Point3D
           x::Float64
           y::Float64
           z::Float64
       end

julia> JSON.print(Point3D(1.0, 2.0, 3.0), 4)
{
    "y": 2.0,
    "z": 3.0,
    "x": 1.0
}
```

Read JSON online: https://riptutorial.com/julia-lang/topic/5468/json

# Chapter 20: Metaprogramming

## Syntax

- macro name(ex) ... end
- quote ... end
- :(...)
- $x
- Meta.quot(x)
- QuoteNode(x)
- esc(x)

## Remarks

Julia's metaprogramming features are heavily inspired by those of Lisp-like languages, and will seem familiar to those with some Lisp background. Metaprogramming is very powerful. When used correctly, it can lead to more concise and readable code.

The `quote ... end` is quasiquote syntax. Instead of the expressions within being evaluated, they are simply parsed. The value of the `quote ... end` expression is the resulting Abstract Syntax Tree (AST).

The `:(...)` syntax is similar to the `quote ... end` syntax, but it is more lightweight. This syntax is more concise than `quote ... end`.

Inside a quasiquote, the `$` operator is special and *interpolates* its argument into the AST. The argument is expected to be an expression which is spliced directly into the AST.

The `Meta.quot(x)` function quotes its argument. This is often useful in combination with using `$` for interpolation, as it allows expressions and symbols to be spliced literally into the AST.

## Examples

### Reimplementing the @show macro

In Julia, the `@show` macro is often useful for debugging purposes. It displays both the expression to be evaluated and its result, finally returning the value of the result:

```
julia> @show 1 + 1
1 + 1 = 2
2
```

It is straightforward to create our own version of `@show`:

```
julia> macro myshow(expression)
```

```
            quote
                value = $expression
                println($(Meta.quot(expression)), " = ", value)
                value
            end
        end
```

To use the new version, simply use the `@myshow` macro:

```
julia> x = @myshow 1 + 1
1 + 1 = 2
2

julia> x
2
```

## Until loop

We're all used to the `while` syntax, that executes its body while the condition is evaluated to `true`. What if we want to implement an `until` loop, that executes a loop until the condition is evaluated to `true`?

In Julia, we can do this by creating a `@until` macro, that stops to execute its body when the condition is met:

```
macro until(condition, expression)
    quote
        while !($condition)
            $expression
        end
    end |> esc
end
```

Here we have used the function chaining syntax `|>`, which is equivalent to calling the `esc` function on the entire `quote` block. The `esc` function prevents macro hygiene from applying to the contents of the macro; without it, variables scoped in the macro will be renamed to prevent collisions with outside variables. See the Julia documentation on macro hygiene for more details.

You can use more than one expression in this loop, by simply putting everything inside a `begin ... end` block:

```
julia> i = 0;

julia> @until i == 10 begin
            println(i)
            i += 1
        end
0
1
2
3
4
5
```

```
6
7
8
9

julia> i
10
```

## QuoteNode, Meta.quot, and Expr(:quote)

There are three ways to quote something using a Julia function:

```
julia> QuoteNode(:x)
:(:x)

julia> Meta.quot(:x)
:(:x)

julia> Expr(:quote, :x)
:(:x)
```

What does "quoting" mean, and what is it good for? Quoting allows us to protect expressions from being interpreted as special forms by Julia. A common use case is when we generate expressions that should contain things that evaluate to symbols. (For example, this macro needs to return a expression that evaluates to a symbol.) It doesn't work simply to return the symbol:

```
julia> macro mysym(); :x; end
@mysym (macro with 1 method)

julia> @mysym
ERROR: UndefVarError: x not defined

julia> macroexpand(:(@mysym))
:x
```

What's going on here? `@mysym` expands to `:x`, which as an expression becomes interpreted as the variable `x`. But nothing has been assigned to `x` yet, so we get an `x not defined` error.

To get around this, we must quote the result of our macro:

```
julia> macro mysym2(); Meta.quot(:x); end
@mysym2 (macro with 1 method)

julia> @mysym2
:x

julia> macroexpand(:(@mysym2))
:(:x)
```

Here, we have used the `Meta.quot` function to turn our symbol into a quoted symbol, which is the result we want.

What is the difference between `Meta.quot` and `QuoteNode`, and which should I use? In almost all

cases, the difference does not really matter. It is perhaps a little safer sometimes to use `QuoteNode` instead of `Meta.quot`. Exploring the difference is informative into how Julia expressions and macros work, however.

## The difference between `Meta.quot` and `QuoteNode`, explained

Here's a rule of thumb:

- If you need or want to support interpolation, use `Meta.quot`;
- If you can't or don't want to allow interpolation, use `QuoteNode`.

In short, the difference is that `Meta.quot` allows interpolation within the quoted thing, while `QuoteNode` protects its argument from any interpolation. To understand interpolation, it is important to mention the `$` expression. There is a kind of expression in Julia called a `$` expression. These expressions allow for escaping. For instance, consider the following expression:

```
julia> ex = :( x = 1; :($x + $x) )
quote
    x = 1
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

When evaluated, this expression will evaluate `1` and assign it to `x`, then construct an expression of the form `_ + _` where the _ will be replaced by the value of `x`. Thus, the result of this should be the *expression* `1 + 1` (which is not yet evaluated, and so distinct from the *value* `2`). Indeed, this is the case:

```
julia> eval(ex)
:(1 + 1)
```

Let's say now that we're writing a macro to build these kinds of expressions. Our macro will take an argument, which will replace the `1` in the `ex` above. This argument can be any expression, of course. Here is something that is not quite what we want:

```
julia> macro makeex(arg)
           quote
               :( x = $(esc($arg)); :($x + $x) )
           end
       end
@makeex (macro with 1 method)

julia> @makeex 1
quote
    x = $(Expr(:escape, 1))
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> @makeex 1 + 1
quote
    x = $(Expr(:escape, 2))
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

The second case is incorrect, because we ought to keep `1 + 1` unevaluated. We fix that by quoting the argument with `Meta.quot`:

```
julia> macro makeex2(arg)
           quote
               :( x = $$(Meta.quot(arg)); :($x + $x) )
           end
       end
@makeex2 (macro with 1 method)

julia> @makeex2 1 + 1
quote
    x = 1 + 1
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

Macro hygiene does not apply to the contents of a quote, so escaping is not necessary in this case (and in fact not legal) in this case.

As mentioned earlier, `Meta.quot` allows interpolation. So let's try that out:

```
julia> @makeex2 1 + $(sin(1))
quote
    x = 1 + 0.8414709848078965
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> let q = 0.5
           @makeex2 1 + $q
       end
quote
    x = 1 + 0.5
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

From the first example, we see that interpolation allows us to inline the `sin(1)`, instead of having the expression be a literal `sin(1)`. The second example shows that this interpolation is done in the macro invocation scope, not the macro's own scope. That's because our macro hasn't actually evaluated any code; all it's doing is generating code. The evaluation of the code (which makes its way into the expression) is done when the expression the macro generates is actually run.

What if we had used `QuoteNode` instead? As you may guess, since `QuoteNode` prevents interpolation from happening at all, this means it won't work.

```
julia> macro makeex3(arg)
           quote
               :( x = $$(QuoteNode(arg)); :($x + $x) )
           end
       end
@makeex3 (macro with 1 method)

julia> @makeex3 1 + $(sin(1))
quote
    x = 1 + $(Expr(:$, :(sin(1))))
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end
```

```
end

julia> let q = 0.5
           @makeex3 1 + $q
       end
quote
    x = 1 + $(Expr(:$, :q))
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> eval(@makeex3 $(sin(1)))
ERROR: unsupported or misplaced expression $
 in eval(::Module, ::Any) at ./boot.jl:234
 in eval(::Any) at ./boot.jl:233
```

In this example, we might agree that `Meta.quot` gives greater flexibility, as it allows interpolation. So why might we ever consider using `QuoteNode`? In some cases, we may not actually desire interpolation, and actually want the literal `$` expression. When would that be desirable? Let's consider a generalization of `@makeex` where we can pass additional arguments determining what comes to the left and right of the `+` sign:

```
julia> macro makeex4(expr, left, right)
           quote
               quote
                   $$(Meta.quot(expr))
                   :($$$(Meta.quot(left)) + $$$(Meta.quot(right)))
               end
           end
       end
@makeex4 (macro with 1 method)

julia> @makeex4 x=1 x x
quote  # REPL[110], line 4:
    x = 1 # REPL[110], line 5:
    $(Expr(:quote, :($(Expr(:$, :x)) + $(Expr(:$, :x)))))
end

julia> eval(ans)
:(1 + 1)
```

A limitation of our implementation of `@makeex4` is that we can't use expressions as either the left and right sides of the expression directly, because they get interpolated. In other words, the expressions may get evaluated for interpolation, but we might want them preserved. (Since there are many levels of quoting and evaluation here, let us clarify: our macro generates *code* that constructs an *expression* that when evaluated produces another *expression*. Phew!)

```
julia> @makeex4 x=1 x/2 x
quote  # REPL[110], line 4:
    x = 1 # REPL[110], line 5:
    $(Expr(:quote, :($(Expr(:$, :(x / 2))) + $(Expr(:$, :x)))))
end

julia> eval(ans)
:(0.5 + 1)
```

We ought to allow the user to specify when interpolation is to happen, and when it shouldn't. Theoretically, that's an easy fix: we can just remove one of the `$` signs in our application, and let the user contribute their own. What this means is that we interpolate a quoted version of the expression entered by the user (which we've already quoted and interpolated once). This leads to the following code, which can be a little confusing at first, due to the multiple nested levels of quoting and unquoting. Try to read and understand what each escape is for.

```
julia> macro makeex5(expr, left, right)
           quote
               quote
                   $$(Meta.quot(expr))
                   :($$(Meta.quot($(Meta.quot(left)))) + $$(Meta.quot($(Meta.quot(right)))))
               end
           end
       end
@makeex5 (macro with 1 method)

julia> @makeex5 x=1 1/2 1/4
quote  # REPL[121], line 4:
    x = 1 # REPL[121], line 5:
    $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :(1 / 2)))))) + $(Expr(:$, :($(Expr(:quote,
:(1 / 4)))))))))
end

julia> eval(ans)
:(1 / 2 + 1 / 4)

julia> @makeex5 y=1 $y $y
ERROR: UndefVarError: y not defined
```

Things started well, but something has gone wrong. The macro's generated code is trying to interpolate the copy of `y` in the macro invocation scope; but there is *no* copy of `y` in the macro invocation scope. Our error is allowing interpolation with the second and third arguments in the macro. To fix this error, we must use `QuoteNode`.

```
julia> macro makeex6(expr, left, right)
           quote
               quote
                   $$(Meta.quot(expr))
                   :($$(Meta.quot($(QuoteNode(left)))) + $$(Meta.quot($(QuoteNode(right)))))
               end
           end
       end
@makeex6 (macro with 1 method)

julia> @makeex6 y=1 1/2 1/4
quote  # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
    $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :(1 / 2)))))) + $(Expr(:$, :($(Expr(:quote,
:(1 / 4)))))))))
end

julia> eval(ans)
:(1 / 2 + 1 / 4)

julia> @makeex6 y=1 $y $y
quote  # REPL[129], line 4:
```

```
    y = 1 # REPL[129], line 5:
    $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :($(Expr(:$, :y)))))))) + $(Expr(:$,
:($(Expr(:quote, :($(Expr(:$, :y))))))))))
end

julia> eval(ans)
:(1 + 1)

julia> @makeex6 y=1 1+$y $y
quote  # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
    $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :(1 + $(Expr(:$, :y)))))))) + $(Expr(:$,
:($(Expr(:quote, :($(Expr(:$, :y))))))))))
end

julia> @makeex6 y=1 $y/2 $y
quote  # REPL[129], line 4:
    y = 1 # REPL[129], line 5:
    $(Expr(:quote, :($(Expr(:$, :($(Expr(:quote, :($(Expr(:$, :y)) / 2))))))) + $(Expr(:$,
:($(Expr(:quote, :($(Expr(:$, :y))))))))))
end

julia> eval(ans)
:(1 / 2 + 1)
```

By using `QuoteNode`, we have protected our arguments from interpolation. Since `QuoteNode` only has the effect of additional protections, it is never harmful to use `QuoteNode`, unless you desire interpolation. However, understanding the difference makes it possible to understand where and why `Meta.quot` could be a better choice.

This long exercise is with an example that is plainly too complex to show up in any reasonable application. Therefore, we have justified the following rule of thumb, mentioned earlier:

- If you need or want to support interpolation, use `Meta.quot`;
- If you can't or don't want to allow interpolation, use `QuoteNode`.

## What about Expr(:quote)?

`Expr(:quote, x)` is equivalent to `Meta.quot(x)`. However, the latter is more idiomatic and is preferred. For code that heavily uses metaprogramming, a `using Base.Meta` line is often used, which allows `Meta.quot` to be referred to as simply `quot`.

### Guide

# π's Metaprogramming bits & bobs

**Goals:**

- Teach through minimal targeted functional/useful/non-abstract examples (e.g. `@swap` or `@assert`) that introduce concepts in suitable contexts

- Prefer to let the code illustrate/demonstrate the concepts rather than paragraphs of explanation
- Avoid linking 'required reading' to other pages -- it interrupts the narrative
- Present things in a sensible order that will making learning easiest

**Resources:**

julialang.org
wikibook (@Cormullion)
5 layers (Leah Hanson)
SO-Doc Quoting (@TotalVerb)
SO-Doc -- Symbols that are not legal identifiers (@TotalVerb)
SO: What is a Symbol in Julia (@StefanKarpinski)
Discourse thread (@p-i-) Metaprogramming

Most of the material has come from the discourse channel, most of that has come from fcard... please prod me if I had forgotten attributions.

# Symbol

```
julia> mySymbol = Symbol("myName")  # or 'identifier'
:myName

julia> myName = 42
42

julia> mySymbol |> eval  # 'foo |> bar' puts output of 'foo' into 'bar', so 'bar(foo)'
42

julia> :( $mySymbol = 1 ) |> eval
1

julia> myName
1
```

**Passing flags into functions:**

```
function dothing(flag)
  if flag == :thing_one
    println("did thing one")
  elseif flag == :thing_two
    println("did thing two")
  end
end
julia> dothing(:thing_one)
did thing one

julia> dothing(:thing_two)
did thing two
```

**A hashkey example:**

```
number_names = Dict{Symbol, Int}()
number_names[:one] = 1
number_names[:two] = 2
number_names[:six] = 6
```

**(Advanced)** (@fcard) `:foo` a.k.a. `:(foo)` yields a symbol if `foo` is a valid identifier, otherwise an expression.

```
# NOTE: Different use of ':' is:
julia> :mySymbol = Symbol('hello world')

#(You can create a symbol with any name with Symbol("<name>"),
# which lets us create such gems as:
julia> one_plus_one = Symbol("1 + 1")
Symbol("1 + 1")

julia> eval(one_plus_one)
ERROR: UndefVarError: 1 + 1 not defined
...

julia> valid_math = :($one_plus_one = 3)
:(1 + 1 = 3)

julia> one_plus_one_plus_two = :($one_plus_one + 2)
:(1 + 1 + 2)

julia> eval(quote
           $valid_math
           @show($one_plus_one_plus_two)
       end)
1 + 1 + 2 = 5
...
```

Basically you can treat Symbols as lightweight strings. That's not what they're for, but you can do it, so why not. Julia's Base itself does it, `print_with_color(:red, "abc")` prints a red-colored abc .

# Expr (AST)

(Almost) everything in Julia is an expression, i.e. an instance of `Expr`, which will hold an AST.

```
# when you type ...
julia> 1+1
2

# Julia is doing: eval(parse("1+1"))
# i.e. First it parses the string "1+1" into an `Expr` object ...
julia> ast = parse("1+1")
:(1 + 1)

# ... which it then evaluates:
julia> eval(ast)
2

# An Expr instance holds an AST (Abstract Syntax Tree).  Let's look at it:
julia> dump(ast)
Expr
```

```
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Int64 1
  typ: Any

# TRY: fieldnames(typeof(ast))

julia>      :(a + b*c + 1)  ==
       parse("a + b*c + 1") ==
       Expr(:call, :+, :a, Expr(:call, :*, :b, :c), 1)
true
```

### Nesting `Expr`s:

```
julia> dump( :(1+2/3) )
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Int64 1
    3: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol /
        2: Int64 2
        3: Int64 3
      typ: Any
  typ: Any

# Tidier rep'n using s-expr
julia> Meta.show_sexpr( :(1+2/3) )
(:call, :+, 1, (:call, :/, 2, 3))
```

## multiline `Expr`s using `quote`

```
julia> blk = quote
           x=10
           x+1
       end
quote  # REPL[121], line 2:
    x = 10 # REPL[121], line 3:
    x + 1
end

julia> blk == :( begin  x=10; x+1  end )
true

# Note: contains debug info:
julia> Meta.show_sexpr(blk)
(:block,
  (:line, 2, Symbol("REPL[121]")),
  (:(=), :x, 10),
  (:line, 3, Symbol("REPL[121]")),
  (:call, :+, :x, 1)
)
```

```
# ... unlike:
julia> noDbg = :( x=10; x+1 )
quote
    x = 10
    x + 1
end
```

... so `quote` is functionally the same but provides extra debug info.

(*) **TIP**: Use `let` to keep `x` within the block

### `quote` -ing a `quote`

`Expr(:quote, x)` is used to represent quotes within quotes.

```
Expr(:quote, :(x + y)) == :(:(x + y))

Expr(:quote, Expr(:$, :x)) == :(:($x))
```

`QuoteNode(x)` is similar to `Expr(:quote, x)` but it prevents interpolation.

```
eval(Expr(:quote, Expr(:$, 1))) == 1

eval(QuoteNode(Expr(:$, 1))) == Expr(:$, 1)
```

(Disambiguate the various quoting mechanisms in Julia metaprogramming

## Are $ and :(...) somehow inverses of one another?

`:(foo)` means "don't look at the value, look at the expression" `$foo` means "change the expression to its value"

`:($(foo)) == foo`. `$(:(foo))` is an error. `$(...)` isn't an operation and doesn't do anything by itself, it's an "interpolate this!" sign that the quoting syntax uses. i.e. It only exists within a quote.

## Is `$foo` the same as `eval(foo)` ?

**No!** `$foo` is exchanged for the compile-time value `eval(foo)` means to do that at runtime

`eval` will occur in the global scope interpolation is local

`eval(:<expr>)` should return the same as just `<expr>` (assuming `<expr>` is a valid expression in the current global space)

```
eval(:(1 + 2)) == 1 + 2

eval(:(let x=1; x + 1 end)) == let x=1; x + 1 end
```

## `macro` S

Ready? :)

```
# let's try to make this!
julia> x = 5; @show x;
x = 5
```

## Let's make our own `@show` macro:

```
macro log(x)
  :(
    println( "Expression: ", $(string(x)), " has value: ", $x )
  )
end

u = 42
f = x -> x^2
@log(u)       # Expression: u has value: 42
@log(42)      # Expression: 42 has value: 42
@log(f(42))   # Expression: f(42) has value: 1764
@log(:u)      # Expression: :u has value: u
```

### `expand` to lower an `Expr`

5 layers (Leah Hanson) <-- explains how Julia takes source code as a string, tokenizes it into an `Expr`-tree (AST), expands out all the macros (still AST), ***lowers*** (lowered AST), then converts into LLVM (and beyond -- at the moment we don't need to worry what lies beyond!)

Q: `code_lowered` acts on functions. Is it possible to lower an `Expr`? A: yup!

```
# function -> lowered-AST
julia> code_lowered(*,(String,String))
1-element Array{LambdaInfo,1}:
 LambdaInfo template for *(s1::AbstractString, ss::AbstractString...) at strings/basic.jl:84

# Expr(i.e. AST) -> lowered-AST
julia> expand(:(x ? y : z))
:(begin
        unless x goto 3
        return y
        3:
        return z
    end)

julia> expand(:(y .= x.(i)))
:((Base.broadcast!)(x,y,i))

# 'Execute' AST or lowered-AST
julia> eval(ast)
```

If you want to only expand macros you can use `macroexpand`:

```
# AST -> (still nonlowered-)AST but with macros expanded:
julia> macroexpand(:(@show x))
quote
    (Base.println)("x = ",(Base.repr)(begin  # show.jl, line 229:
                #28#value = x
            end))
    #28#value
end
```

...which returns a non-lowered AST but with all macros expanded.

#### `esc()`

`esc(x)` returns an Expr that says "don't apply hygiene to this", it's the same as `Expr(:escape, x)`. Hygiene is what keeps a macro self-contained, and you `esc` things if you want them to "leak". e.g.

### Example: `swap` macro to illustrate `esc()`

```
macro swap(p, q)
  quote
    tmp = $(esc(p))
    $(esc(p)) = $(esc(q))
    $(esc(q)) = tmp
  end
end

x,y = 1,2
@swap(x,y)
println(x,y)  # 2 1
```

`$` allows us to 'escape out of' the `quote`. So why not simply `$p` and `$q`? i.e.

```
    # FAIL!
    tmp = $p
    $p = $q
    $q = tmp
```

Because that would look first to the `macro` scope for `p`, and it would find a local `p` i.e. the parameter `p` (yes, if you subsequently access `p` without `esc`-ing, the macro considers the `p` parameter as a local variable).

So `$p = ...` is just a assigning to the local `p`. it's not affecting whatever variable was passed-in in the calling context.

Ok so how about:

```
    # Almost!
    tmp = $p          # <-- you might think we don't
    $(esc(p)) = $q    #       need to esc() the RHS
    $(esc(q)) = tmp
```

So `esc(p)` is 'leaking' `p` into the calling context. *"The thing that was passed into the macro that **we receive as `p`**"*

```
julia> macro swap(p, q)
         quote
           tmp = $p
           $(esc(p)) = $q
           $(esc(q)) = tmp
         end
       end
@swap (macro with 1 method)

julia> x, y = 1, 2
(1,2)

julia> @swap(x, y);

julia> @show(x, y);
x = 2
y = 1

julia> macroexpand(:(@swap(x, y)))
quote  # REPL[34], line 3:
    #10#tmp = x # REPL[34], line 4:
    x = y # REPL[34], line 5:
    y = #10#tmp
end
```

As you can see `tmp` gets the hygiene treatment `#10#tmp`, whereas `x` and `y` don't. Julia is making a unique identifier for `tmp`, something you can manually do with `gensym`, ie:

```
julia> gensym(:tmp)
Symbol("##tmp#270")
```

But: **There is a gotcha:**

```
julia> module Swap
       export @swap

       macro swap(p, q)
         quote
           tmp = $p
           $(esc(p)) = $q
           $(esc(q)) = tmp
         end
       end
       end
Swap

julia> using Swap

julia> x,y = 1,2
(1,2)

julia> @swap(x,y)
ERROR: UndefVarError: x not defined
```

Another thing julia's macro hygiene does is, if the macro is from another module, it makes any variables (that were not assigned inside the macro's returning expression, like `tmp` in this case) globals of the current module, so `$p` becomes `Swap.$p`, likewise `$q` -> `Swap.$q`.

In general, if you need a variable that is outside the macro's scope you should esc it, so you should `esc(p)` and `esc(q)` regardless if they are on the LHS or RHS of a expression, or even by themselves.

people have already mentioned `gensym`s a few times and soon you will be seduced by the dark side of defaulting to escaping the whole expression with a few `gensym`s peppered here and there, but... Make sure to understand how hygiene works before trying to be smarter than it! It's not a particularly complex algorithm so it shouldn't take too long, but don't rush it! Don't use that power until you understand all the ramifications of it... (@fcard)

## Example: `until` macro

(@Ismael-VC)

```
"until loop"
macro until(condition, block)
    quote
        while ! $condition
            $block
        end
    end |> esc
end

julia> i=1;  @until(  i==5,  begin; print(i); i+=1; end  )
1234
```

(@fcard) `|>` is controversial, however. I am surprised a mob hasn't come to argue yet. (maybe everyone is just tired of it). There is a recommendation of having most if not all of the macro just be a call to a function, so:

```
macro until(condition, block)
    esc(until(condition, block))
end

function until(condition, block)
    quote
        while !$condition
            $block
        end
    end
end
```

...is a safer alternative.

##@fcard's simple macro challenge

Task: Swap the operands, so `swaps(1/2)` gives `2.00` i.e. `2/1`

```
macro swaps(e)
    e.args[2:3] = e.args[3:-1:2]
    e
end
```

```
@swaps(1/2)
2.00
```

More macro challenges from @fcard here

---

# Interpolation and `assert` macro

http://docs.julialang.org/en/release-0.5/manual/metaprogramming/#building-an-advanced-macro

```
macro assert(ex)
    return :( $ex ? nothing : throw(AssertionError($(string(ex)))) )
end
```

Q: Why the last `$`? A: It interpolates, i.e. forces Julia to `eval` that `string(ex)` as execution passes through the invocation of this macro. i.e. If you just run that code it won't force any evaluation. But the moment you do `assert(foo)` Julia will ***invoke*** this macro replacing its 'AST token/Expr' with whatever it returns, and the `$` *will* kick into action.

## A fun hack for using { } for blocks

(@fcard) I don't think there is anything technical keeping `{}` from being used as blocks, in fact one can even pun on the residual `{}` syntax to make it work:

```
julia> macro c(block)
         @assert block.head == :cell1d
         esc(quote
           $(block.args...)
         end)
       end
@c (macro with 1 method)

julia> @c {
         print(1)
         print(2)
         1+2
       }
123
```

*(unlikely to still work if/when the {} syntax is repurposed)

---

***So first Julia sees the macro token, so it will read/parse tokens until the matching `end`, and create what? An `Expr` with `.head=:macro` or something? Does it store `"a+1"` as a string or does it break it apart into `:+(:a, 1)`? How to view?***

?

(@fcard) In this case because of lexical scope, a is undefined in `@M`s scope so it uses the global variable... I actually forgot to escape the flipplin' expression in my dumb example, but the *"only works within the same module"* part of it still applies.

```
julia> module M
       macro m()
         :(a+1)
       end
       end
M

julia> a = 1
1

julia> M.@m
ERROR: UndefVarError: a not defined
```

The reason being that, if the macro is used in any module other than the one it was defined in, any variables not defined within the code-to-be-expanded are treated as globals of the macro's module.

```
julia> macroexpand(:(M.@m))
:(M.a + 1)
```

---

# ADVANCED

###@Ismael-VC

```
@eval begin
    "do-until loop"
    macro $(:do)(block, until::Symbol, condition)
        until ≠ :until &&
            error("@do expected `until` got `$until`")
        quote
            let
                $block
                @until $condition begin
                    $block
                end
            end
        end |> esc
    end
end
julia> i = 0
0

julia> @do begin
           @show i
           i += 1
       end until i == 5
i = 0
i = 1
i = 2
i = 3
i = 4
```

**Scott's macro:**

```
"""
Internal function to return captured line number information from AST

##Parameters
- a:     Expression in the julia type Expr

##Return

- Line number in the file where the calling macro was invoked
"""
_lin(a::Expr) = a.args[2].args[1].args[1]

"""
Internal function to return captured file name information from AST

##Parameters
- a:     Expression in the julia type Expr

##Return
- The name of the file where the macro was invoked
"""
_fil(a::Expr) = string(a.args[2].args[1].args[2])

"""
Internal function to determine if a symbol is a status code or variable
"""
function _is_status(sym::Symbol)
    sym in (:OK, :WARNING, :ERROR) && return true
    str = string(sym)
    length(str) > 4 && (str[1:4] == "ERR_" || str[1:5] == "WARN_" || str[1:5] == "INFO_")
end

"""
Internal function to return captured error code from AST

##Parameters
- a:     Expression in the julia type Expr

##Return
- Error code from the captured info in the AST from the calling macro
"""
_err(a::Expr) =
    (sym = a.args[2].args[2] ; _is_status(sym) ? Expr(:., :Status, QuoteNode(sym)) : sym)

"""
Internal function to produce a call to the log function based on the macro arguments and the
AST from the ()->ERRCODE anonymous function definition used to capture error code, file name
and line number where the macro is used

##Parameters
- level:     Loglevel which has to be logged with macro
- a:         Expression in the julia type Expr
- msgs:      Optional message

##Return
- Statuscode
"""
function _log(level, a, msgs)
    if isempty(msgs)
        :( log($level, $(esc(:Symbol))($(_fil(a))), $(_lin(a)), $(_err(a)) )
    else
```

```
        :( log($level, $(esc(:Symbol))($(_fil(a))), $(_lin(a)), $(_err(a)),
message=$(esc(msgs[1]))) )
    end
end

macro warn(a, msgs...)  ; _log(Warning, a, msgs) ; end
```

# junk / unprocessed ...

## view/dump a macro

(@p-i-) Suppose I just do `macro m(); a+1; end` in a fresh REPL. With no `a` defined. How can I 'view' it? like, is there some way to 'dump' a macro? Without actually executing it

(@fcard) All the code in macros are actually put into functions, so you can only view their lowered or type-inferred code.

```
julia> macro m()  a+1  end
@m (macro with 1 method)

julia> @code_typed @m
LambdaInfo for @m()
:(begin
        return Main.a + 1
    end)

julia> @code_lowered @m
CodeInfo(:(begin
        nothing
        return Main.a + 1
    end))
# ^ or: code_lowered(eval(Symbol("@m")))[1] # ouf!
```

Other ways to get a macro's function:

```
julia> macro getmacro(call) call.args[1] end
@getmacro (macro with 1 method)

julia> getmacro(name) = getfield(current_module(), name.args[1])
getmacro (generic function with 1 method)

julia> @getmacro @m
@m (macro with 1 method)

julia> getmacro(:@m)
@m (macro with 1 method)
```

```
julia> eval(Symbol("@M"))
@M (macro with 1 method)

julia> dump( eval(Symbol("@M")) )
@M (function of type #@M)

julia> code_typed( eval(Symbol("@M")) )
```

```
1-element Array{Any,1}:
 LambdaInfo for @M()

julia> code_typed( eval(Symbol("@M")) )[1]
LambdaInfo for @M()
:(begin
        return $(Expr(:copyast, :($(QuoteNode(:(a + 1))))))
    end::Expr)

julia> @code_typed @M
LambdaInfo for @M()
:(begin
        return $(Expr(:copyast, :($(QuoteNode(:(a + 1))))))
    end::Expr)
```

^ looks like I can use `code_typed` instead

### How to understand `eval(Symbol("@M"))`?

(@fcard) Currently, every macro has a function associated with it. If you have a macro called `M`, then the macro's function is called `@M`. Generally you can get a function's value with e.g. `eval(:print)` but with a macro's function you need to do `Symbol("@M")`, since just `:@M` becomes an `Expr(:macrocall, Symbol("@M"))` and evaluating that causes a macro-expansion.

## Why doesn't `code_typed` display params?

(@p-i-)

```
julia> code_typed( x -> x^2 )[1]
LambdaInfo for (::##5#6)(::Any)
:(begin
        return x ^ 2
    end)
```

^ here I see one `::Any` param, but it doesn't seem to be connected with the token `x`.

```
 julia> code_typed( print )[1]
LambdaInfo for print(::IO, ::Char)
:(begin
        (Base.write)(io,c)
        return Base.nothing
    end::Void)
```

^ similarly here; there is nothing to connect `io` with the `::IO` So surely this can't be a complete dump of the AST representation of that particular `print` method…?

(@fcard) `print(::IO, ::Char)` only tells you what method it is, it's not part of the AST. It isn't even present in master anymore:

```
julia> code_typed(print)[1]
CodeInfo(:(begin
        (Base.write)(io,c)
```

```
        return Base.nothing
    end))=>Void
```

(@p-i-) I don't understand what you mean by that. It seems to be dumping the AST for the body of that method, no? I thought `code_typed` gives the AST for a function. But it seems to be missing the first step, i.e. setting up tokens for params.

(@fcard) `code_typed` is meant to only show the body's AST, but for now it does give the complete AST of the method, in the form of a `LambdaInfo` (0.5) or `CodeInfo` (0.6), but a lot of the information is omitted when printed to the repl. You will need to inspect the `LambdaInfo` field by field in order to get all the details. `dump` is going to flood your repl, so you could try:

```
macro method_info(call)
  quote
    method = @code_typed $(esc(call))
    print_info_fields(method)
  end
end

function print_info_fields(method)
  for field in fieldnames(typeof(method))
    if isdefined(method, field) && !(field in [Symbol(""), :code])
      println("  $field = ", getfield(method, field))
    end
  end
  display(method)
end

print_info_fields(x::Pair) = print_info_fields(x[1])
```

Which gives all the values of the named fields of a method's AST:

```
julia> @method_info print(STDOUT, 'a')
  rettype = Void
  sparam_syms = svec()
  sparam_vals = svec()
  specTypes = Tuple{Base.#print,Base.TTY,Char}
  slottypes = Any[Base.#print,Base.TTY,Char]
  ssavaluetypes = Any[]
  slotnames = Any[Symbol("#self#"),:io,:c]
  slotflags = UInt8[0x00,0x00,0x00]
  def = print(io::IO, c::Char) at char.jl:45
  nargs = 3
  isva = false
  inferred = true
  pure = false
  inlineable = true
  inInference = false
  inCompile = false
  jlcall_api = 0
  fptr = Ptr{Void} @0x00007f7a7e96ce10
LambdaInfo for print(::Base.TTY, ::Char)
:(begin
        $(Expr(:invoke, LambdaInfo for write(::Base.TTY, ::Char), :(Base.write), :(io), :(c)))
        return Base.nothing
    end::Void)
```

See the lil' `def = print(io::IO, c::Char)`? There you go! (also the `slotnames = [..., :io, :c]` part)
Also yes, the difference in output is because I was showing the results on master.

## ???

(@Ismael-VC) you mean like this? Generic dispatch with Symbols

You can do it this way:

```
julia> function dispatchtest{alg}(::Type{Val{alg}})
           println("This is the generic dispatch. The algorithm is $alg")
       end
dispatchtest (generic function with 1 method)

julia> dispatchtest(alg::Symbol) = dispatchtest(Val{alg})
dispatchtest (generic function with 2 methods)

julia> function dispatchtest(::Type{Val{:Euler}})
           println("This is for the Euler algorithm!")
       end
dispatchtest (generic function with 3 methods)

julia> dispatchtest(:Foo)
This is the generic dispatch. The algorithm is Foo

julia> dispatchtest(:Euler)
```

This is for the Euler algorithm! I wonder what does @fcard thinks about generic symbol dispatch! ---^ :angel:

### Module Gotcha

```
@def m begin
  a+2
end

@m # replaces the macro at compile-time with the expression a+2
```

More accurately, only works within the toplevel of the module the macro was defined in.

```
julia> module M
       macro m1()
         a+1
       end
       end
M

julia> macro m2()
         a+1
       end
@m2 (macro with 1 method)

julia> a = 1
1
```

```
julia> M.@m1
ERROR: UndefVarError: a not defined

julia> @m2
2

julia> let a = 20
         @m2
       end
2
```

`esc` keeps this from happening, but defaulting to always using it goes against the language design. A good defense for this is to keep one from using and introducing names within macros, which makes them hard to track to a human reader.

### Python `dict`/JSON like syntax for `Dict` literals.

# Introduction

Julia uses the following syntax for dictionaries:

```
Dict({k₁ => v₁, k₂ => v₂, …, k□₋₁ => v□₋₁, k□ => v□)
```

While Python and JSON looks like this:

```
{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}
```

For **illustrative purposes** we could also use this syntax in Julia and add new semantics to it (`Dict` syntax is the idiomatic way in Julia, which is recommended).

First let's see what *kind* of expression it is:

```
julia> parse("{1:2 , 3: 4}") |> Meta.show_sexpr
(:cell1d, (:(:), 1, 2), (:(:), 3, 4))
```

This means we need to take this `:cell1d` expression and either transform it or return a new expression that should look like this:

```
julia> parse("Dict(1 => 2 , 3 => 4)") |> Meta.show_sexpr
(:call, :Dict, (:(=>), 1, 2), (:(=>), 3, 4))
```

# Macro definition

The following macro, while simple, allows to demonstrate such code generation and transformation:

```
macro dict(expr)
    # Check the expression has the correct form:
```

```
    if expr.head ≠ :cell1d || any(sub_expr.head ≠ :(:) for sub_expr ∈ expr.args)
        error("syntax: expected `{k₁: v₁, k₂: v₂, …, kₙ₋₁: vₙ₋₁, kₙ: vₙ}`")
    end

    # Create empty `:Dict` expression which will be returned:
    block = Expr(:call, :Dict)    # :(Dict())

    # Append `(key => value)` pairs to the block:
    for pair in expr.args
        k, v = pair.args
        push!(block.args, :($k => $v))
    end    # :(Dict(k₁ => v₁, k₂ => v₂, …, kₙ₋₁ => vₙ₋₁, kₙ => vₙ))

    # Block is escaped so it can reach variables from it's calling scope:
    return esc(block)
end
```

Let's check out the resulting macro expansion:

```
julia> :(@dict {"a": :b, 'c': 1, :d: 2.0}) |> macroexpand
:(Dict("a" => :b,'c' => 1,:d => 2.0))
```

# Usage

```
julia> @dict {"a": :b, 'c': 1, :d: 2.0}
Dict{Any,Any} with 3 entries:
  "a" => :b
  :d  => 2.0
  'c' => 1

julia> @dict {
           "string": :b,
           'c'     : 1,
           :symbol : π,
           Function: print,
           (1:10)  : range(1, 10)
       }
Dict{Any,Any} with 5 entries:
  1:10     => 1:10
  Function => print
  "string" => :b
  :symbol  => π = 3.1415926535897...
  'c'      => 1
```

The last example is exactly equivalent to:

```
Dict(
    "string" => :b,
    'c'      => 1,
    :symbol  => π,
    Function => print,
    (1:10)   => range(1, 10)
)
```

# Misusage

```
julia> @dict {"one": 1, "two": 2, "three": 3, "four": 4, "five" => 5}
syntax: expected `{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}`

julia> @dict ["one": 1, "two": 2, "three": 3, "four": 4, "five" => 5]
syntax: expected `{k₁: v₁, k₂: v₂, …, k□₋₁: v□₋₁, k□: v□}`
```

Notice that Julia has other uses for colon `:` as such you will need to wrap range literal expressions with parenthesis or use the `range` function, for example.

Read Metaprogramming online: https://riptutorial.com/julia-lang/topic/1945/metaprogramming

# Chapter 21: Modules

## Syntax

- module Module; ...; end
- using Module
- import Module

## Examples

### Wrap Code in a Module

The `module` keyword can be used to begin a module, which allows code to be organized and namespaced. Modules can define an external interface, typically consisting of `export`ed symbols. To support this external interface, modules can have unexported internal functions and types not intended for public use.

Some modules primarily exist to wrap a type and associated functions. Such modules, by convention, are usually named with the plural form of the type's name. For instance, if we have a module that provides a `Building` type, we can call such a module `Buildings`.

```
module Buildings

immutable Building
    name::String
    stories::Int
    height::Int  # in metres
end

name(b::Building) = b.name
stories(b::Building) = b.stories
height(b::Building) = b.height

function Base.show(io::IO, b::Building)
    Base.print(stories(b), "-story ", name(b), " with height ", height(b), "m")
end

export Building, name, stories, height

end
```

The module can then be used with the `using` statement:

```
julia> using Buildings

julia> Building("Burj Khalifa", 163, 830)
163-story Burj Khalifa with height 830m

julia> height(ans)
830
```

## Using Modules to Organize Packages

Typically, packages consist of one or more modules. As packages grow, it may be useful to organize the main module of the package into smaller modules. A common idiom is to define those modules as submodules of the main module:

```
module RootModule

module SubModule1

...

end

module SubModule2

...

end

end
```

Initially, neither root module nor submodules have access to each others' exported symbols. However, relative imports are supported to address this issue:

```
module RootModule

module SubModule1

const x = 10
export x

end

module SubModule2

# import submodule of parent module
using ..SubModule1
const y = 2x
export y

end

# import submodule of current module
using .SubModule1
using .SubModule2
const z = x + y

end
```

In this example, the value of `RootModule.z` is `30`.

Read Modules online: https://riptutorial.com/julia-lang/topic/7368/modules

# Chapter 22: Packages

## Syntax

- Pkg.add(package)
- Pkg.checkout(package, branch="master")
- Pkg.clone(url)
- Pkg.dir(package)
- Pkg.pin(package, version)
- Pkg.rm(package)

## Parameters

| Parameter | Details |
|---|---|
| `Pkg.add(`*`package`*`)` | Download and install the given registered package. |
| `Pkg.checkout(`*`package`*`, `*`branch`*`)` | Check out the given branch for the given registered package. *`branch`* is optional and defaults to `"master"`. |
| `Pkg.clone(`*`url`*`)` | Clone the Git repository at the given URL as a package. |
| `Pkg.dir(`*`package`*`)` | Get the location on disk for the given package. |
| `Pkg.pin(`*`package`*`, `*`version`*`)` | Force the package to remain at the given version. *`version`* is optional and defaults to the current version of the package. |
| `Pkg.rm(`*`package`*`)` | Remove the given package from the list of required packages. |

## Examples

### Install, use, and remove a registered package

After finding an official Julia package, it is straightforward to download and install the package. Firstly, it's recommended to refresh the local copy of METADATA:

```
julia> Pkg.update()
```

This will ensure that you get the latest versions of all packages.

Suppose that the package we want to install is named `Currencies.jl`. The command to run to install this package would be:

```
julia> Pkg.add("Currencies")
```

This command will install not only the package itself, but also all of its dependencies.

If the installation is successful, you can test that the package works properly:

```
julia> Pkg.test("Currencies")
```

Then, to use the package, use

```
julia> using Currencies
```

and proceed as described by the package's documentation, usually linked to or included from its README.md file.

To uninstall a package that is no longer needed, use the `Pkg.rm` function:

```
julia> Pkg.rm("Currencies")
```

Note that this may not actually remove the package directory; instead it will merely mark the package as no longer required. Often, this is perfectly fine — it will save time in case you need the package again in the future. But if necessary, to remove the package physically, call the `rm` function, then call `Pkg.resolve`:

```
julia> rm(Pkg.dir("Currencies"); recursive=true)

julia> Pkg.resolve()
```

## Check out a different branch or version

Sometimes, the latest tagged version of a package is buggy or is missing some required features. Advanced users may wish to update to the latest development version of a package (sometimes referred to as the "master", named after the usual name for a development branch in Git). The benefits of this include:

- Developers contributing to a package should contribute to the latest development version.
- The latest development version may have useful features, bugfixes, or performance enhancements.
- Users reporting a bug may wish to check if a bug occurs on the latest development version.

However, there are many drawbacks to running the latest development version:

- The latest development version may be poorly-tested and have serious bugs.
- The latest development version can change frequently, breaking your code.

To check out the latest development branch of a package named `JSON.jl`, for example, use

```
Pkg.checkout("JSON")
```

To check out a different branch or tag (not named "master"), use

```
Pkg.checkout("JSON", "v0.6.0")
```

However, if the tag represents a version, it's usually better to use

```
Pkg.pin("JSON", v"0.6.0")
```

Note that a version literal is used here, not a plain string. The `Pkg.pin` version informs the package manager of the version constraint, allowing the package manager to offer feedback on what problems it might cause.

To return to the latest tagged version,

```
Pkg.free("JSON")
```

## Install an unregistered package

Some experimental packages are not included in the METADATA package repository. These packages can be installed by directly cloning their Git repositories. Note that there may be dependencies of unregistered packages that are themselves unregistered; those dependencies cannot be resolved by the package manager and must be resolved manually. For example, to install the unregistered package `OhMyREPL.jl`:

```
Pkg.clone("https://github.com/KristofferC/Tokenize.jl")
Pkg.clone("https://github.com/KristofferC/OhMyREPL.jl")
```

Then, as is usual, use `using` to use the package:

```
using OhMyREPL
```

Read Packages online: https://riptutorial.com/julia-lang/topic/5815/packages

# Chapter 23: Parallel Processing

## Examples

### pmap

`pmap` takes a function (that you specify) and applies it to all of the elements in an array. This work is divided up amongst the available workers. `pmap` then returns places the results from that function into another array.

```
addprocs(3)
sqrts = pmap(sqrt, 1:10)
```

if you function takes multiple arguments, you can supply multiple vectors to `pmap`

```
dots = pmap(dot, 1:10, 11:20)
```

As with `@parallel`, however, if the function given to `pmap` is not in base Julia (i.e. it is user-defined or defined in a package) then you must make sure that function is available to all workers first:

```
@everywhere begin
    function rand_det(n)
        det(rand(n,n))
    end
end

determinants = pmap(rand_det, 1:10)
```

See also this SO Q&A.

### @parallel

@parallel can be used to parallellize a loop, dividing steps of the loop up over different workers. As a very simple example:

```
addprocs(3)

a = collect(1:10)

for idx = 1:10
    println(a[idx])
end
```

For a slightly more complex example, consider:

```
@time begin
    @sync begin
        @parallel for idx in 1:length(a)
```

```
            sleep(a[idx])
        end
    end
end
27.023411 seconds (13.48 k allocations: 762.532 KB)
julia> sum(a)
55
```

Thus, we see that if we had executed this loop without `@parallel` it would have taken 55 seconds, rather than 27, to execute.

We can also supply a reduction operator for the `@parallel` macro. Suppose we have an array, we want to sum each column of the array and then multiply these sums by each other:

```
A = rand(100,100);

@parallel (*) for idx = 1:size(A,1)
    sum(A[:,idx])
end
```

There are several important things to keep in mind when using `@parallel` to avoid unexpected behavior.

**First:** if you want to use any functions in your loops that are not in base Julia (e.g. either functions you define in your script or that you import from packages), then you must make those functions accessible to the workers. Thus, for example, the following would *not* work:

```
myprint(x) = println(x)
for idx = 1:10
    myprint(a[idx])
end
```

Instead, we would need to use:

```
@everywhere begin
    function myprint(x)
        println(x)
    end
end

@parallel for idx in 1:length(a)
    myprint(a[idx])
end
```

**Second** Although each worker will be able to access the objects in the scope of the controller, they will *not* be able to modify them. Thus

```
a = collect(1:10)
@parallel for idx = 1:length(a)
   a[idx] += 1
end

julia> a'
```

```
1x10 Array{Int64,2}:
 1  2  3  4  5  6  7  8  9  10
```

Whereas, if we had executed the loop wihtout the @parallel it would have successfully modified the array `a`.

TO ADDRESS THIS, we can instead make `a` a `SharedArray` type object so that each worker can access and modify it:

```
a = convert(SharedArray{Float64,1}, collect(1:10))
@parallel for idx = 1:length(a)
    a[idx] += 1
end

julia> a'
1x10 Array{Float64,2}:
 2.0  3.0  4.0  5.0  6.0  7.0  8.0  9.0  10.0  11.0
```

## @spawn and @spawnat

The macros `@spawn` and `@spawnat` are two of the tools that Julia makes available to assign tasks to workers. Here is an example:

```
julia> @spawnat 2 println("hello world")
RemoteRef{Channel{Any}}(2,1,3)

julia>  From worker 2:  hello world
```

Both of these macros will evaluate an expression on a worker process. The only difference between the two is that `@spawnat` allows you to choose which worker will evaluate the expression (in the example above worker 2 is specified) whereas with `@spawn` a worker will be automatically chosen, based on availability.

In the above example, we simply had worker 2 execute the println function. There was nothing of interest to return or retrieve from this. Often, however, the expression we sent to the worker will yield something we wish to retrieve. Notice in the example above, when we called `@spawnat`, before we got the printout from worker 2, we saw the following:

```
RemoteRef{Channel{Any}}(2,1,3)
```

This indicates that the `@spawnat` macro will return a `RemoteRef` type object. This object in turn will contain the return values from our expression that is sent to the worker. If we want to retrieve those values, we can first assign the `RemoteRef` that `@spawnat` returns to an object and then, and then use the `fetch()` function which operates on a `RemoteRef` type object, to retrieve the results stored from an evaluation performed on a worker.

```
julia> result = @spawnat 2 2 + 5
RemoteRef{Channel{Any}}(2,1,26)

julia> fetch(result)
```

```
7
```

The key to being able to use `@spawn` effectively is understanding the nature behind the expressions that it operates on. Using `@spawn` to send commands to workers is slightly more complicated than just typing directly what you would type if you were running an "interpreter" on one of the workers or executing code natively on them. For instance, suppose we wished to use `@spawnat` to assign a value to a variable on a worker. We might try:

```
@spawnat 2 a = 5
RemoteRef{Channel{Any}}(2,1,2)
```

Did it work? Well, let's see by having worker 2 try to print `a`.

```
julia> @spawnat 2 println(a)
RemoteRef{Channel{Any}}(2,1,4)

julia>
```

Nothing happened. Why? We can investigate this more by using `fetch()` as above. `fetch()` can be very handy because it will retrieve not just successful results but also error messages as well. Without it, we might not even know that something has gone wrong.

```
julia> result = @spawnat 2 println(a)
RemoteRef{Channel{Any}}(2,1,5)

julia> fetch(result)
ERROR: On worker 2:
UndefVarError: a not defined
```

The error message says that `a` is not defined on worker 2. But why is this? The reason is that we need to wrap our assignment operation into an expression that we then use `@spawn` to tell the worker to evaluate. Below is an example, with explanation following:

```
julia> @spawnat 2 eval(:(a = 2))
RemoteRef{Channel{Any}}(2,1,7)

julia> @spawnat 2 println(a)
RemoteRef{Channel{Any}}(2,1,8)

julia>  From worker 2:  2
```

The `:()` syntax is what Julia uses to designate expressions. We then use the `eval()` function in Julia, which evaluates an expression, and we use the `@spawnat` macro to instruct that the expression be evaluated on worker 2.

We could also achieve the same result as:

```
julia> @spawnat(2, eval(parse("c = 5")))
RemoteRef{Channel{Any}}(2,1,9)

julia> @spawnat 2 println(c)
```

```
RemoteRef{Channel{Any}}(2,1,10)

julia>  From worker 2:  5
```

This example demonstrates two additional notions. First, we see that we can also create an expression using the `parse()` function called on a string. Secondly, we see that we can use parentheses when calling `@spawnat`, in situations where this might make our syntax more clear and manageable.

## When to use @parallel vs. pmap

The Julia documentation advises that

> pmap() is designed for the case where each function call does a large amount of work. In contrast, @parallel for can handle situations where each iteration is tiny, perhaps merely summing two numbers.

There are several reasons for this. First, `pmap` incurs greater start up costs initiating jobs on workers. Thus, if the jobs are very small, these startup costs may become inefficient. Conversely, however, `pmap` does a "smarter" job of allocating jobs amongst workers. In particular, it builds a queue of jobs and sends a new job to each worker whenever that worker becomes available. `@parallel` by contrast, divvies up all work to be done amongst the workers when it is called. As such, if some workers take longer on their jobs than others, you can end up with a situation where most of your workers have finished and are idle while a few remain active for an inordinate amount of time, finishing their jobs. Such a situation, however, is less likely to occur with very small and simple jobs.

The following illustrates this: suppose we have two workers, one of which is slow and the other of which is twice as fast. Ideally, we would want to give the fast worker twice as much work as the slow worker. (or, we could have fast and slow jobs, but the principal is the exact same). `pmap` will accomplish this, but `@parallel` won't.

For each test, we initialize the following:

```
addprocs(2)

@everywhere begin
    function parallel_func(idx)
        workernum = myid() - 1
        sleep(workernum)
        println("job $idx")
    end
end
```

Now, for the `@parallel` test, we run the following:

```
@parallel for idx = 1:12
    parallel_func(idx)
end
```

And get back print output:

```
julia>     From worker 2:    job 1
    From worker 3:    job 7
    From worker 2:    job 2
    From worker 2:    job 3
    From worker 3:    job 8
    From worker 2:    job 4
    From worker 2:    job 5
    From worker 3:    job 9
    From worker 2:    job 6
    From worker 3:    job 10
    From worker 3:    job 11
    From worker 3:    job 12
```

It's almost sweet. The workers have "shared" the work evenly. Note that each worker has completed 6 jobs, even though worker 2 is twice as fast as worker 3. It may be touching, but it is inefficient.

For for the `pmap` test, I run the following:

```
pmap(parallel_func, 1:12)
```

and get the output:

```
From worker 2:    job 1
From worker 3:    job 2
From worker 2:    job 3
From worker 2:    job 5
From worker 3:    job 4
From worker 2:    job 6
From worker 2:    job 8
From worker 3:    job 7
From worker 2:    job 9
From worker 2:    job 11
From worker 3:    job 10
From worker 2:    job 12
```

Now, note that worker 2 has performed 8 jobs and worker 3 has performed 4. This is exactly in proportion to their speed, and what we want for optimal efficiency. `pmap` is a hard task master - from each according to their ability.

## @async and @sync

According to the documentation under `?@async`, "`@async` wraps an expression in a Task." What this means is that for whatever falls within its scope, Julia will start this task running but then proceed to whatever comes next in the script without waiting for the task to complete. Thus, for instance, without the macro you will get:

```
julia> @time sleep(2)
  2.005766 seconds (13 allocations: 624 bytes)
```

But with the macro, you get:

```
julia> @time @async sleep(2)
  0.000021 seconds (7 allocations: 657 bytes)
Task (waiting) @0x0000000112a65ba0

julia>
```

Julia thus allows the script to proceed (and the `@time` macro to fully execute) without waiting for the task (in this case, sleeping for two seconds) to complete.

The `@sync` macro, by contrast, will "Wait until all dynamically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@parallel` are complete." (according to the documentation under `?@sync`). Thus, we see:

```
julia> @time @sync @async sleep(2)
  2.002899 seconds (47 allocations: 2.986 KB)
Task (done) @0x0000000112bd2e00
```

In this simple example then, there is no point to including a single instance of `@async` and `@sync` together. But, where `@sync` can be useful is where you have `@async` applied to multiple operations that you wish to allow to all start at once without waiting for each to complete.

For example, suppose we have multiple workers and we'd like to start each of them working on a task simultaneously and then fetch the results from those tasks. An initial (but incorrect) attempt might be:

```
addprocs(2)
@time begin
    a = cell(nworkers())
    for (idx, pid) in enumerate(workers())
        a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 4.011576 seconds (177 allocations: 9.734 KB)
```

The problem here is that the loop waits for each `remotecall_fetch()` operation to finish, i.e. for each process to complete its work (in this case sleeping for 2 seconds) before continuing to start the next `remotecall_fetch()` operation. In terms of a practical situation, we're not getting the benefits of parallelism here, since our processes aren't doing their work (i.e. sleeping) simultaneously.

We can correct this, however, by using a combination of the `@async` and `@sync` macros:

```
@time begin
    a = cell(nworkers())
    @sync for (idx, pid) in enumerate(workers())
        @async a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 2.009416 seconds (274 allocations: 25.592 KB)
```

Now, if we count each step of the loop as a separate operation, we see that there are two

separate operations preceded by the `@async` macro. The macro allows each of these to start up, and the code to continue (in this case to the next step of the loop) before each finishes. But, the use of the `@sync` macro, whose scope encompasses the whole loop, means that we won't allow the script to proceed past that loop until all of the operations preceded by `@async` have completed.

It is possible to get an even more clear understanding of the operation of these macros by further tweaking the above example to see how it changes under certain modifications. For instance, suppose we just have the `@async` without the `@sync`:

```
@time begin
    a = cell(nworkers())
    for (idx, pid) in enumerate(workers())
        println("sending work to $pid")
        @async a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
## 0.001429 seconds (27 allocations: 2.234 KB)
```

Here, the `@async` macro allows us to continue in our loop even before each `remotecall_fetch()` operation finishes executing. But, for better or worse, we have no `@sync` macro to prevent the code from continuing past this loop until all of the `remotecall_fetch()` operations finish.

Nevertheless, each `remotecall_fetch()` operation is still running in parallel, even once we go on. We can see that because if we wait for two seconds, then the array a, containing the results, will contain:

```
sleep(2)
julia> a
2-element Array{Any,1}:
 nothing
 nothing
```

(The "nothing" element is the result of a successful fetch of the results of the sleep function, which does not return any values)

We can also see that the two `remotecall_fetch()` operations start at essentially the same time because the `print` commands that precede them also execute in rapid succession (output from these commands not shown here). Contrast this with the next example where the `print` commands execute at a 2 second lag from each other:

If we put the `@async` macro on the whole loop (instead of just the inner step of it), then again our script will continue immediately without waiting for the `remotecall_fetch()` operations to finish. Now, however, we only allow for the script to continue past the loop as a whole. We don't allow each individual step of the loop to start before the previous one finished. As such, unlike in the example above, two seconds after the script proceeds after the loop, the `results` array still has one element as `#undef` indicating that the second `remotecall_fetch()` operation still has not completed.

```
@time begin
    a = cell(nworkers())
    @async for (idx, pid) in enumerate(workers())
        println("sending work to $pid")
```

```
        a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
# 0.001279 seconds (328 allocations: 21.354 KB)
# Task (waiting) @0x0000000115ec9120
## This also allows us to continue to

sleep(2)

a
2-element Array{Any,1}:
    nothing
 #undef
```

And, not surprisingly, if we put the `@sync` and `@async` right next to each other, we get that each `remotecall_fetch()` runs sequentially (rather than simultaneously) but we don't continue in the code until each has finished. In other words, this would be essentially the equivalent of if we had neither macro in place, just like `sleep(2)` behaves essentially identically to `@sync @async sleep(2)`

```
@time begin
    a = cell(nworkers())
    @sync @async for (idx, pid) in enumerate(workers())
        a[idx] = remotecall_fetch(pid, sleep, 2)
    end
end
# 4.019500 seconds (4.20 k allocations: 216.964 KB)
# Task (done) @0x0000000115e52a10
```

Note also that it is possible to have more complicated operations inside the scope of the `@async` macro. The documentation gives an example containing an entire loop within the scope of `@async`.

Recall that the help for the sync macros states that it will "Wait until all dynamically-enclosed uses of `@async`, `@spawn`, `@spawnat` and `@parallel` are complete." For the purposes of what counts as "complete" it matters how you define the tasks within the scope of the `@sync` and `@async` macros. Consider the below example, which is a slight variation on one of the examples given above:

```
@time begin
    a = cell(nworkers())
    @sync for (idx, pid) in enumerate(workers())
        @async a[idx] = remotecall(pid, sleep, 2)
    end
end
## 0.172479 seconds (93.42 k allocations: 3.900 MB)

julia> a
2-element Array{Any,1}:
 RemoteRef{Channel{Any}}(2,1,3)
 RemoteRef{Channel{Any}}(3,1,4)
```

The earlier example took roughly 2 seconds to execute, indicating that the two tasks were run in parallel and that the script waiting for each to complete execution of their functions before proceeding. This example, however, has a much lower time evaluation. The reason is that for the purposes of `@sync` the `remotecall()` operation has "finished" once it has sent the worker the job to do. (Note that the resulting array, a, here, just contains `RemoteRef` object types, which just indicate

that there is something going on with a particular process which could in theory be fetched at some point in the future). By contrast, the `remotecall_fetch()` operation has only "finished" when it gets the message from the worker that its task is complete.

Thus, if you are looking for ways to ensure that certain operations with workers have completed before moving on in your script (as for instance is discussed in this post) it is necessary to think carefully about what counts as "complete" and how you will measure and then operationalize that in your script.

## Adding Workers

When you first start Julia, by default, there will only be a single process running and available to give work to. You can verify this using:

```
julia> nprocs()
1
```

In order to take advantage of parallel processing, you must first add additional workers who will then be available to do work that you assign to them. You can do this within your script (or from the interpreter) using: `addprocs(n)` where `n` is the number of processes you want to use.

Alternatively, you can add processes when you start Julia from the command line using:

```
$ julia -p n
```

where `n` is how many *additional* processes you want to add. Thus, if we start Julia with

```
$ julia -p 2
```

When Julia starts we will get:

```
julia> nprocs()
3
```

Read Parallel Processing online: https://riptutorial.com/julia-lang/topic/4542/parallel-processing

# Chapter 24: Reading a DataFrame from a file

## Examples

### Reading a dataframe from delimiter separated data

You may want to read a `DataFrame` from a CSV (Comma separated values) file or maybe even from a TSV or WSV (tabs and whitespace separated files). If your file has the right extension, you can use the `readtable` function to read in the dataframe:

```
readtable("dataset.CSV")
```

But what if your file doesn't have the right extension? You can specify the delimiter that your file uses (comma, tab, whitespace etc) as a keyword argument to the `readtable` function:

```
readtable("dataset.txt", separator=',')
```

### Handling different comment comment marks

Data sets often contain comments that explain the data format or contain the license and usage terms. You usually want to ignore these lines when you read in the `DataFrame`.

The `readtable` function assumes that comment lines begin with the '#' character. However, your file may use comment marks like `%` or `//`. To make sure that `readtable` handles these correctly, you can specify the comment mark as a keyword argument:

```
readtable("dataset.csv", allowcomments=true, commentmark='%')
```

Read Reading a DataFrame from a file online: https://riptutorial.com/julia-lang/topic/7340/reading-a-dataframe-from-a-file

# Chapter 25: Regexes

## Syntax

- Regex("[regex]")
- r"[regex]"
- match(needle, haystack)
- matchall(needle, haystack)
- eachmatch(needle, haystack)
- ismatch(needle, haystack)

## Parameters

| Parameter | Details |
|---|---|
| `needle` | the `Regex` to look for in the `haystack` |
| `haystack` | the text in which to look for the `needle` |

## Examples

### Regex literals

Julia supports regular expressions[1]. The PCRE library is used as the regex implementation. Regexes are like a mini-language within a language. Since most languages and many text editors provide some support for regex, documentation and examples of how to use regex in general are outside the scope of this example.

It is possible to construct a `Regex` from a string using the constructor:

```
julia> Regex("(cat|dog)s?")
```

But for convenience and easier escaping, the `@r_str` string macro can be used instead:

```
julia> r"(cat|dog)s?"
```

[1]: Technically, Julia supports regexes, which are distinct from and more powerful than what are called regular expressions in language theory. Frequently, the term "regular expression" will be used to refer to regexes also.

### Finding matches

There are four primary useful functions for regular expressions, all of which take arguments in `needle, haystack` order. The terminology "needle" and "haystack" come from the English idiom

"finding a needle in a haystack". In the context of regexes, the regex is the needle, and the text is the haystack.

The `match` function can be used to find the first match in a string:

```
julia> match(r"(cat|dog)s?", "my cats are dogs")
RegexMatch("cats", 1="cat")
```

The `matchall` function can be used to find all matches of a regular expression in a string:

```
julia> matchall(r"(cat|dog)s?", "The cat jumped over the dogs.")
2-element Array{SubString{String},1}:
 "cat"
 "dogs"
```

The `ismatch` function returns a boolean indicating whether a match was found inside the string:

```
julia> ismatch(r"(cat|dog)s?", "My pigs")
false

julia> ismatch(r"(cat|dog)s?", "My cats")
true
```

The `eachmatch` function returns an iterator over `RegexMatch` objects, suitable for use with `for` loops:

```
julia> for m in eachmatch(r"(cat|dog)s?", "My cats and my dog")
           println("Matched $(m.match) at index $(m.offset)")
       end
Matched cats at index 4
Matched dog at index 16
```

## Capture groups

The substrings captured by capture groups are accessible from `RegexMatch` objects using indexing notation.

For instance, the following regex parses North American phone numbers written in `(555)-555-5555` format:

```
julia> phone = r"\((\d{3})\)-(\d{3})-(\d{4})"
```

and suppose we wish to extract the phone numbers from a text:

```
julia> text = """
       My phone number is (555)-505-1000.
       Her phone number is (555)-999-9999.
       """
"My phone number is (555)-505-1000.\nHer phone number is (555)-999-9999.\n"
```

Using the `matchall` function, we can get an array of the substrings matched themselves:

```
julia> matchall(phone, text)
2-element Array{SubString{String},1}:
 "(555)-505-1000"
 "(555)-999-9999"
```

But suppose we want to access the area codes (the first three digits, enclosed in brackets). Then we can use the `eachmatch` iterator:

```
julia> for m in eachmatch(phone, text)
           println("Matched $(m.match) with area code $(m[1])")
       end
Matched (555)-505-1000 with area code 555
Matched (555)-999-9999 with area code 555
```

Note here that we use `m[1]` because the area code is the first capture group in our regular expression. We can get all three components of the phone number as a tuple using a function:

```
julia> splitmatch(m) = m[1], m[2], m[3]
splitmatch (generic function with 1 method)
```

Then we can apply such a function to a particular `RegexMatch`:

```
julia> splitmatch(match(phone, text))
("555","505","1000")
```

Or we could `map` it across each match:

```
julia> map(splitmatch, eachmatch(phone, text))
2-element Array{Tuple{SubString{String},SubString{String},SubString{String}},1}:
 ("555","505","1000")
 ("555","999","9999")
```

Read Regexes online: https://riptutorial.com/julia-lang/topic/5890/regexes

# Chapter 26: REPL

## Syntax

- julia>
- help?>
- shell>
- \[latex]

## Remarks

Other packages may define their own REPL modes in addition to the default modes. For instance, the `Cxx` package defines the `cxx>` shell mode for a C++ REPL. These modes are usually accessible with their own special keys; see package documentation for more details.

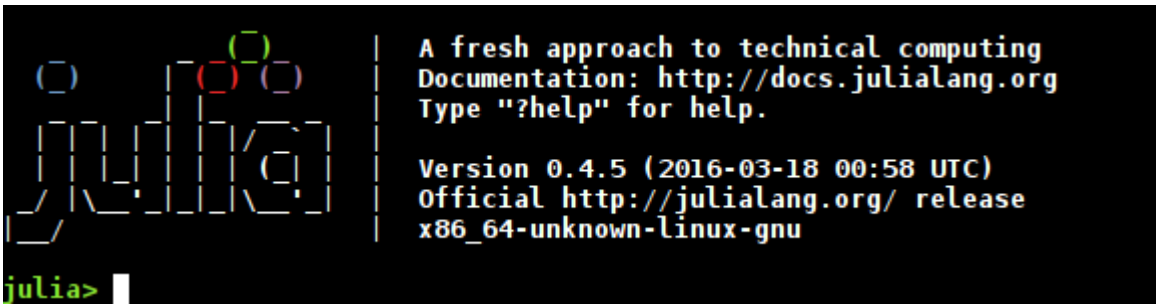
## Examples

### Launch the REPL

After installing Julia, to launch the read-eval-print-loop (REPL):

## On Unix Systems

Open a terminal window, then type `julia` at the prompt, then hit `Return`. You should see something like this come up:




## On Windows

Find the Julia program in your start menu, and click it. The REPL should be launched.

### Using the REPL as a Calculator

The Julia REPL is an excellent calculator. We can start with some simple operations:

```
julia> 1 + 1
```

```
2

julia> 8 * 8
64

julia> 9 ^ 2
81
```

The `ans` variable contains the result of the last calculation:

```
julia> 4 + 9
13

julia> ans + 9
22
```

We can define our own variables using the assignment `=` operator:

```
julia> x = 10
10

julia> y = 20
20

julia> x + y
30
```

Julia has implicit multiplication for numeric literals, which makes some calculations quicker to write:

```
julia> 10x
100

julia> 2(x + y)
60
```

If we make a mistake and do something that is not allowed, the Julia REPL will throw an error, often with a helpful tip on how to fix the problem:

```
julia> 1 ^ -1
ERROR: DomainError:
Cannot raise an integer x to a negative power -n.
Make x a float by adding a zero decimal (e.g. 2.0^-n instead of 2^-n), or write
1/x^n, float(x)^-n, or (x//1)^-n.
 in power_by_squaring at ./intfuncs.jl:82
 in ^ at ./intfuncs.jl:106

julia> 1.0 ^ -1
1.0
```

To access or edit previous commands, use the ↑ (Up) key, which moves to the last item in history. The ↓ moves to the next item in history. The ← and → keys can be used to move and make edits to a line.

Julia has some built-in mathematical constants, including `e` and `pi` (or `π`).

```
julia> e
e = 2.7182818284590...

julia> pi
π = 3.1415926535897...

julia> 3π
9.42477796076938
```

We can type characters like `π` quickly by using their LaTeX codes: press `\`, then `p` and `i`, then hit the `Tab` key to substitute the `\pi` just typed with `π`. This works for other Greek letters and additional unicode symbols.

We can use any of Julia's built-in math functions, which range from simple to fairly powerful:

```
julia> cos(π)
-1.0

julia> besselh(1, 1, 1)
0.44005058574493355 - 0.7812128213002889im
```

Complex numbers are supported using `im` as an imaginary unit:

```
julia> abs(3 + 4im)
5.0
```

Some functions will not return a complex result unless you give it a complex input, even if the input is real:

```
julia> sqrt(-1)
ERROR: DomainError:
sqrt will only return a complex result if called with a complex argument. Try
sqrt(complex(x)).
 in sqrt at math.jl:146

julia> sqrt(-1+0im)
0.0 + 1.0im

julia> sqrt(complex(-1))
0.0 + 1.0im
```

Exact operations on rational numbers are possible using the `//` rational division operator:

```
julia> 1//3 + 1//3
2//3
```

See the Arithmetic topic for more about what sorts of arithmetic operators are supported by Julia.

# Dealing with Machine Precision

Note that machine integers are constrained in size, and will overflow if the result is too big to be stored:

```
julia> 2^62
4611686018427387904

julia> 2^63
-9223372036854775808
```

This can be prevented by using arbitrary-precision integers in the computation:

```
julia> big"2"^62
4611686018427387904

julia> big"2"^63
9223372036854775808
```

Machine floating points are also limited in precision:

```
julia> 0.1 + 0.2
0.30000000000000004
```

More (but still limited) precision is possible by again using `big`:

```
julia> big"0.1" + big"0.2"
3.000000000000000000000000000000000000000000000000000000000000000000000000000000000000017e-01
```

Exact arithmetic can be done in some cases using `Rational`s:

```
julia> 1//10 + 2//10
3//10
```

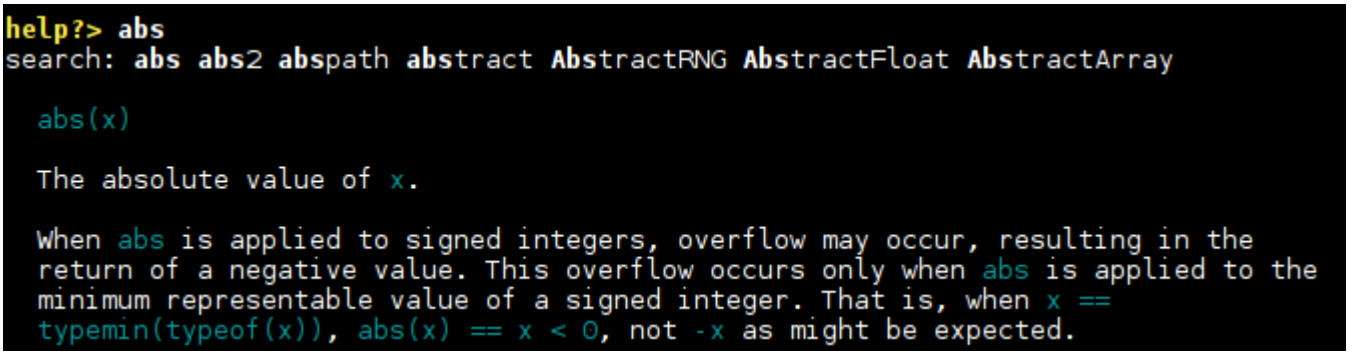
### Using REPL Modes

There are three built-in REPL modes in Julia: the Julia mode, the help mode, and the shell mode.

## The Help Mode

The Julia REPL comes with a built-in help system. Press `?` at the `julia>` prompt to access the `help?>` prompt.

At the help prompt, type the name of some function or type to get help for:




Even if you do not spell the function correctly, Julia can suggest some functions that are possibly what you meant:

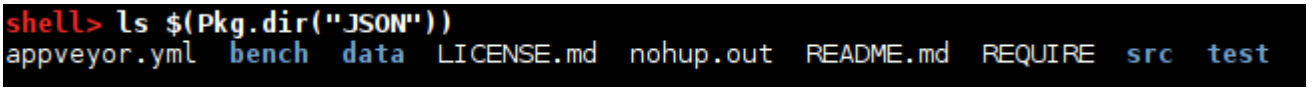
```
help?> printline
search:

Couldn't find printline
Perhaps you meant println, pipeline, @inline or print
  No documentation found.

  Binding printline does not exist.
```

This documentation works for other modules too, as long as they use the Julia documentation system.

```
julia> using Currencies

help?> @usingcurrencies
  Export each given currency symbol into the current namespace. The individual unit
  exported will be a full unit of the currency specified, not the smallest possible
  unit. For instance, @usingcurrencies EUR will export EUR, a currency unit worth
  1€, not a currency unit worth 0.01€.

  @usingcurrencies EUR, GBP, AUD
  7AUD  # 7.00 AUD

  There is no sane unit for certain currencies like XAU or XAG, so this macro does
  not work for those. Instead, define them manually:

  const XAU = Monetary(:XAU; precision=4)
```

## The Shell Mode

See Using Shell from inside the REPL for more details about how to use Julia's shell mode, which is accessible by hitting ; at the prompt. This shell mode supports interpolating data from the Julia REPL session, which makes it easy to call Julia functions and make their results into shell commands:




Read REPL online: https://riptutorial.com/julia-lang/topic/5739/repl

# Chapter 27: Shell Scripting and Piping

## Syntax

- ;shell command

## Examples

### Using Shell from inside the REPL

From inside the interative Julia shell (also known as REPL), you can access the system's shell by typing ; right after the prompt:

```
shell>
```

From here on, you can type any shell comand and they will be run from inside the REPL:

```
shell> ls
Desktop     Documents   Pictures   Templates
Downloads   Music       Public     Videos
```

To exit this mode, type `backspace` when the prompt is empty.

### Shelling out from Julia code

Julia code can create, manipulate, and execute command literals, which execute in the OS's system environment. This is powerful but often makes programs less portable.

A command literal can be created using the `` literal. Information can be interpolated using the `$` interpolation syntax, as with string literals. Julia variables passed through command literals need not be escaped first; they are not actually passed to the shell, but rather directly to the kernel. However, Julia displays these objects so that they appear properly escaped.

```
julia> msg = "a commit message"
"a commit message"

julia> command = `git commit -am $msg`
`git commit -am 'a commit message'`

julia> cd("/directory/where/there/are/unstaged/changes")

julia> run(command)
[master (root-commit) 0945387] add a
 4 files changed, 1 insertion(+)
```

Read Shell Scripting and Piping online: https://riptutorial.com/julia-lang/topic/5420/shell-scripting-and-piping

# Chapter 28: String Macros

## Syntax

- macro"string" # short, string macro form
- @macro_str "string" # long, regular macro form
- macro`command`

## Remarks

String macros are not quite as powerful as plain old strings — because interpolation must be implemented in the macro's logic, string macros are unable to contain string literals of the same delimiter for interpolation.

For instance, although

```
julia> "$("x")"
"x"
```

works, the string macro text form

```
julia> doc"$("x")"
ERROR: KeyError: key :x not found
```

gets parsed incorrectly. This can be somewhat mitigated by using triple-quotes as the outer string delimiter;

```
julia> doc"""$("x")"""
"x"
```

does indeed work properly.

## Examples

### Using string macros

String macros are syntactic sugar for certain macro invocations. The parser expands syntax like

```
mymacro"my string"
```

into

```
@mymacro_str "my string"
```

which then, like any other macro call, gets substituted with whatever expression the `@mymacro_str`

macro returns. Base Julia comes with several string macros, such as:

**@b_str**

This string macro constructs byte arrays instead of strings. The contents of the string, encoded as UTF-8, will be used as the array of bytes. This can be useful for interfacing with low-level APIs, many of which work with byte arrays instead of strings.

```
julia> b"Hello World!"
12-element Array{UInt8,1}:
 0x48
 0x65
 0x6c
 0x6c
 0x6f
 0x20
 0x57
 0x6f
 0x72
 0x6c
 0x64
 0x21
```

**@big_str**

This macro will return a `BigInt` or a `BigFloat` parsed from the string it's given.

```
julia> big"1"
1

julia> big"1.0"
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

This macro exists because `big(0.1)` does not behave as one might initially expect: the `0.1` is a `Float64` approximation of true `0.1` (`1//10`), and promoting that to `BigFloat` will keep the approximation error of `Float64`. Using the macro will parse `0.1` directly to a `BigFloat`, reducing the approximation error.

```
julia> big(0.1)
1.000000000000000055511151231257827021181583404541015625000000000000000000000000e-01

julia> big"0.1"
1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000002e-01
```

**@doc_str**

This string macro constructs `Base.Markdown.MD` objects, which are used in the internal documentation system to provide rich-text documentation for any environment. These MD objects render well in a terminal:

```
julia> doc"""
       This is a markdown documentation string.

       ## Heading

       Math ``1 + 2`` and `code` are supported.
       """
 This is a markdown documentation string.

    Heading
   =========

 Math 1 + 2 and code are supported.
```

and also in a browser:

```
In [2]: doc"""
        This is a markdown documentation string.

        ## Heading

        Math ``1 + 2`` and `code` are supported.
        """
```

Out[2]: This is a markdown documentation string.

## Heading

Math $1 + 2$ and code are supported.

**@html_str**

This string macro constructs HTML string literals, which render nicely in a browser:

```
In [1]: html"""
        <p><abbr title="Hypertext Markup Language">HTML</abbr> text.</p>
        """
```

Out[1]: HTML text.

**@ip_str**

This string macro constructs IP address literals. It works with both IPv4 and IPv6:

```
julia> ip"127.0.0.1"
ip"127.0.0.1"

julia> ip"::"
ip"::"
```

**@r_str**

This string macro constructs `Regex` literals.

**`@s_str`**

This string macro constructs `SubstitutionString` literals, which work together with `Regex` literals to allow more advanced textual substitution.

**`@text_str`**

This string macro is similar in spirit to `@doc_str` and `@html_str`, but does not have any fancy formatting features:

```
In [3]: text"""
        This is some plain text.
        """
Out[3]: This is some plain text.
```

**`@v_str`**

This string macro constructs `VersionNumber` literals. See Version Numbers for a description of what they are and how to use them.

**`@MIME_str`**

This string macro constructs the singleton types of MIME types. For instance, `MIME"text/plain"` is the type of `MIME("text/plain")`.

## Symbols that are not legal identifiers

Julia Symbol literals must be legal identifiers. This works:

```
julia> :cat
:cat
```

But this does not:

```
julia> :2cat
ERROR: MethodError: no method matching *(::Int64, ::Base.#cat)
Closest candidates are:
  *(::Any, ::Any, ::Any, ::Any...) at operators.jl:288

*{T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}}(::T<:Union{Int128,I
::T<:Union{Int128,Int16,Int32,Int64,Int8,UInt128,UInt16,UInt32,UInt64,UInt8}) at int.jl:33
  *(::Real, ::Complex{Bool}) at complex.jl:180
  ...
```

What looks like a symbol literal here is actually being parsed as an implicit multiplication of `:2` (which is just `2`) and the function `cat`, which obviously does not work.

We can use

```
julia> Symbol("2cat")
Symbol("2cat")
```

to work around the issue.

A string macro could help to make this more terse. If we define the `@sym_str` macro:

```
macro sym_str(str)
    Meta.quot(Symbol(str))
end
```

then we can simply do

```
julia> sym"2cat"
Symbol("2cat")
```

to create symbols which are not valid Julia identifiers.

Of course, these techniques can also create symbols that *are* valid Julia identifiers. For example,

```
julia> sym"test"
:test
```

### Implementing interpolation in a string macro

String macros do not come with built-in interpolation facilities. However, it is possible to manually implement this functionality. Note that it is not possible to embed without escaping string literals that have the same delimiter as the surrounding string macro; that is, although `""" $("x") """` is possible, `" $("x") "` is not. Instead, this must be escaped as `" $(\"x\") "`. See the remarks section for more details about this limitation.

There are two approaches to implementing interpolation manually: implement parsing manually, or get Julia to do the parsing. The first approach is more flexible, but the second approach is easier.

## Manual parsing

```
macro interp_str(s)
    components = []
    buf = IOBuffer(s)
    while !eof(buf)
        push!(components, rstrip(readuntil(buf, '$'), '$'))
        if !eof(buf)
            push!(components, parse(buf; greedy=false))
        end
    end
    quote
        string($(map(esc, components)...))
    end
end
```

```
macro e_str(s)
    esc(parse("\"$(escape_string(s))\""))
end
```

This method escapes the string (but note that `escape_string` does *not* escape the `$` signs) and passes it back to Julia's parser to parse. Escaping the string is necessary to ensure that `"` and `\` do not affect the string's parsing. The resulting expression is a `:string` expression, which can be examined and decomposed for macro purposes.

### Command macros

0.6.0-dev

In Julia v0.6 and later, command macros are supported in addition to regular string macros. A command macro invocation like

```
mymacro`xyz`
```

gets parsed as the macro call

```
@mymacro_cmd "xyz"
```

Note that this is similar to string macros, except with `_cmd` instead of `_str`.

We typically use command macros for code, which in many languages frequently contains `"` but rarely contains `` ` ``. For instance, it is fairly straightforward to reimplement a simple version of quasiquoting using command macros:

```
macro julia_cmd(s)
    esc(Meta.quot(parse(s)))
end
```

We can use this macro either inline:

```
julia> julia`1+1`
:(1 + 1)

julia> julia`hypot2(x,y)=x^2+y^2`
:(hypot2(x,y) = begin  # none, line 1:
            x ^ 2 + y ^ 2
        end)
```

or multiline:

```
julia> julia```
       function hello()
           println("Hello, World!")
```

```
       end
       ```
:(function hello() # none, line 2:
        println("Hello, World!")
    end)
```

Interpolation using `$` is supported:

```
julia> x = 2
2

julia> julia`1 + $x`
:(1 + 2)
```

but the version given here only allows one expression:

```
julia> julia```
       x = 2
       y = 3
       ```
ERROR: ParseError("extra token after end of expression")
```

However, extending it to handle multiple expressions is not difficult.

Read String Macros online: https://riptutorial.com/julia-lang/topic/5817/string-macros

# Chapter 29: String Normalization

## Syntax

- normalize_string(s::String, ...)

## Parameters

| Parameter | Details |
|---|---|
| `casefold=true` | Fold the string to a canonical case based off the Unicode standard. |
| `stripmark=true` | Strip diacritical marks (i.e. accents) from characters in the input string. |

## Examples

### Case-Insensitive String Comparison

Strings can be compared with the `==` operator in Julia, but this is sensitive to differences in case. For instance, `"Hello"` and `"hello"` are considered different strings.

```
julia> "Hello" == "Hello"
true

julia> "Hello" == "hello"
false
```

To compare strings in a case-insensitive manner, normalize the strings by case-folding them first. For example,

```
equals_ignore_case(s, t) =
    normalize_string(s, casefold=true) == normalize_string(t, casefold=true)
```

This approach also handles non-ASCII Unicode correctly:

```
julia> equals_ignore_case("Hello", "hello")
true

julia> equals_ignore_case("Weierstraß", "WEIERSTRASS")
true
```

Note that in German, the uppercase form of the ß character is SS.

### Diacritic-Insensitive String Comparison

Sometimes, one wants strings like `"resume"` and `"ré sumé "` to compare equal. That is, graphemes

that share a basic glyph, but possibly differ because of additions to those basic glyphs. Such comparison can be accomplished by stripping diacritical marks.

```
equals_ignore_mark(s, t) =
    normalize_string(s, stripmark=true) == normalize_string(t, stripmark=true)
```

This allows the above example to work correctly. Additionally, it works well even with non-ASCII Unicode characters.

```
julia> equals_ignore_mark("resume", "ré sumé ")
true

julia> equals_ignore_mark("αβγ", "à β ỹ ")
true
```

Read String Normalization online: https://riptutorial.com/julia-lang/topic/7612/string-normalization

# Chapter 30: Strings

## Syntax

- "[string]"
- '[Unicode scalar value]'
- graphemes([string])

## Parameters

| Parameter | Details |
|---|---|
| **For** | `sprint(f, xs...)` |
| `f` | A function that takes an `IO` object as its first argument. |
| `xs` | Zero or more remaining arguments to pass to `f`. |

## Examples

### Hello, World!

Strings in Julia are delimited using the `"` symbol:

```
julia> mystring = "Hello, World!"
"Hello, World!"
```

Note that unlike some other languages, the `'` symbol *cannot* be used instead. `'` defines a *character literal*; this is a `Char` data type and will only store a single Unicode scalar value:

```
julia> 'c'
'c'

julia> 'character'
ERROR: syntax: invalid character literal
```

One can extract the unicode scalar values from a string by iterating over it with a `for` loop:

```
julia> for c in "Hello, World!"
           println(c)
       end
H
e
l
l
o
,
```

```
W
o
r
l
d
!
```

## Graphemes

Julia's `Char` type represents a Unicode scalar value, which only in some cases corresponds to what humans perceive as a "character". For instance, one representation of the character é, as in résumé, is actually a combination of two Unicode scalar values:

```
julia> collect("é ")
2-element Array{Char,1}:
 'e'
 ' '
```

The Unicode descriptions for these codepoints are "LATIN SMALL LETTER E" and "COMBINING ACUTE ACCENT". Together, they define a single "human" character, which is Unicode terms is called a grapheme. More specifically, Unicode Annex #29 motivates the definition of a grapheme cluster because:

> It is important to recognize that what the user thinks of as a “character”—a basic unit of a writing system for a language—may not be just a single Unicode code point. Instead, that basic unit may be made up of multiple Unicode code points. To avoid ambiguity with the computer use of the term character, this is called a user-perceived character. For example, “G” + acute-accent is a user-perceived character: users think of it as a single character, yet is actually represented by two Unicode code points. These user-perceived characters are approximated by what is called a grapheme cluster, which can be determined programmatically.

Julia provides the `graphemes` function to iterate over the grapheme clusters in a string:

```
julia> for c in graphemes("ré sumé ")
           println(c)
       end
r
é
s
u
m
é
```

Note how the result, printing each character on its own line, is better than if we had iterated over the Unicode scalar values:

```
julia> for c in "ré sumé "
           println(c)
       end
```

```
r
e

s
u
m
e
```

Typically, when working with characters in a user-perceived sense, it is more useful to deal with grapheme clusters than with Unicode scalar values. For instance, suppose we want to write a function to compute the length of a single word. A naïve solution would be to use

```
julia> wordlength(word) = length(word)
wordlength (generic function with 1 method)
```

We note that the result is counter-intuitive when the word includes grapheme clusters that consist of more than one codepoint:

```
julia> wordlength("ré sumé ")
8
```

When we use the more correct definition, using the `graphemes` function, we get the expected result:

```
julia> wordlength(word) = length(graphemes(word))
wordlength (generic function with 1 method)

julia> wordlength("ré sumé ")
6
```

## Convert numeric types to strings

There are numerous ways to convert numeric types to strings in Julia:

```
julia> a = 123
123

julia> string(a)
"123"

julia> println(a)
123
```

The `string()` function can also take more arguments:

```
julia> string(a, "b")
"123b"
```

You can also insert (aka interpolate) integers (and certain other types) into strings using `$`:

```
julia> MyString = "my integer is $a"
"my integer is 123"
```

**Performance Tip:** The above methods can be quite convenient at times. But, if you will be performing many, many such operations and you are concerned about execution speed of your code, the Julia performance guide recommends against this, and instead in favor of the below methods:

You can supply multiple arguments to `print()` and `println()` which will operate on them exactly as `string()` operates on multiple arguments:

```
julia> println(a, "b")
123b
```

Or, when writing to file, you can similarly use, e.g.

```
open("/path/to/MyFile.txt", "w") do file
    println(file, a, "b", 13)
end
```

or

```
file = open("/path/to/MyFile.txt", "a")
println(file, a, "b", 13)
close(file)
```

These are faster because they avoid needing to first form a string from given pieces and then output it (either to the console display or a file) and instead just sequentially output the various pieces.

Credits: Answer based on SO Question What's the best way to convert an Int to a String in Julia? with Answer by Michael Ohlrogge and Input from Fengyang Wang

## String interpolation (insert value defined by variable into string)

In Julia, as in many other languages, it is possible to interpolate by inserting values defined by variables into strings. For a simple example:

```
n = 2
julia> MyString = "there are $n ducks"
"there are 2 ducks"
```

We can use other types than numeric, e.g.

```
Result = false
julia> println("test results is $Result")
test results is false
```

You can have multiple interpolations within a given string:

```
MySubStr = "a32"
MyNum = 123.31
```

```
println("$MySubStr  ,   $MyNum")
```

**Performance Tip** Interpolation is quite convenient. But, if you are going to be doing it many times very rapidly, it is not the most efficient. Instead, see Convert numeric types to strings for suggestions when performance is an issue.

## Using sprint to Create Strings with IO Functions

Strings can be made from functions that work with `IO` objects by using the `sprint` function. For instance, the `code_llvm` function accepts an `IO` object as the first argument. Typically, it is used like

```
julia> code_llvm(STDOUT, *, (Int, Int))

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
  %2 = mul i64 %1, %0
  ret i64 %2
}
```

Suppose we want that output as a string instead. Then we can simply do

```
julia> sprint(code_llvm, *, (Int, Int))
"\ndefine i64 @\"jlsys_*_46115\"(i64, i64) #0 {\ntop:\n  %2 = mul i64 %1, %0\n  ret i64
%2\n}\n"

julia> println(ans)

define i64 @"jlsys_*_46115"(i64, i64) #0 {
top:
  %2 = mul i64 %1, %0
  ret i64 %2
}
```

Converting the results of "interactive" functions like `code_llvm` into strings can be useful for automated analysis, such as testing whether generated code may have regressed.

The `sprint` function is a higher-order function which takes the function operating on `IO` objects as its first argument. Behind the scenes, it creates an `IOBuffer` in RAM, calls the given function, and takes the data from the buffer into a `String` object.

Read Strings online: https://riptutorial.com/julia-lang/topic/5562/strings

# Chapter 31: sub2ind

## Syntax

- sub2ind(dims::Tuple{Vararg{Integer}}, I::Integer...)
- sub2ind{T<:Integer}(dims::Tuple{Vararg{Integer}}, I::AbstractArray{T<:Integer,1}...)

## Parameters

| parameter | details |
|---|---|
| dims::Tuple{Vararg{Integer}} | size of the array |
| I::Integer... | subscripts(scalar) of the array |
| I::AbstractArray{T<:Integer,1}... | subscripts(vector) of the array |

## Remarks

The second example shows that the result of `sub2ind` might be very buggy in some specific cases.

## Examples

### Convert subscripts to linear indices

```
julia> sub2ind((3,3), 1, 1)
1

julia> sub2ind((3,3), 1, 2)
4

julia> sub2ind((3,3), 2, 1)
2

julia> sub2ind((3,3), [1,1,2], [1,2,1])
3-element Array{Int64,1}:
 1
 4
 2
```

### Pits & Falls

```
# no error, even the subscript is out of range.
julia> sub2ind((3,3), 3, 4)
12
```

One cannot determine whether a subscript is in the range of an array by comparing its index:

```
julia> sub2ind((3,3), -1, 2)
2

julia> 0 < sub2ind((3,3), -1, 2) <= 9
true
```

Read sub2ind online: https://riptutorial.com/julia-lang/topic/1914/sub2ind

# Chapter 32: Time

## Syntax

- now()
- Dates.today()
- Dates.year(t)
- Dates.month(t)
- Dates.day(t)
- Dates.hour(t)
- Dates.minute(t)
- Dates.second(t)
- Dates.millisecond(t)
- Dates.format(t, s)

## Examples

### Current Time

To get the current date and time, use the `now` function:

```
julia> now()
2016-09-04T00:16:58.122
```

This is the local time, which includes the machine's configured time zone. To get the time in the Coordinated Universal Time (UTC) time zone, use `now(Dates.UTC)`:

```
julia> now(Dates.UTC)
2016-09-04T04:16:58.122
```

To get the current date, without the time, use `today()`:

```
julia> Dates.today()
2016-10-30
```

The return value of `now` is a `DateTime` object. There are functions to get the individual components of a `DateTime`:

```
julia> t = now()
2016-09-04T00:16:58.122

julia> Dates.year(t)
2016

julia> Dates.month(t)
9
```

```
julia> Dates.day(t)
4

julia> Dates.hour(t)
0

julia> Dates.minute(t)
16

julia> Dates.second(t)
58

julia> Dates.millisecond(t)
122
```

It is possible to format a `DateTime` using a specially-formatted format string:

```
julia> Dates.format(t, "yyyy-mm-dd at HH:MM:SS")
"2016-09-04 at 00:16:58"
```

Since many of the `Dates` functions are exported from the `Base.Dates` module, it can save some typing to write

```
using Base.Dates
```

which then enables accessing the qualified functions above without the `Dates.` qualification.

Read Time online: https://riptutorial.com/julia-lang/topic/5812/time

# Chapter 33: Tuples

## Syntax

- a,
- a, b
- a, b = xs
- ()
- (a,)
- (a, b)
- (a, b...)
- Tuple{T, U, V}
- NTuple{N, T}
- Tuple{T, U, Vararg{V}}

## Remarks

Tuples have much better runtime performance than arrays for two reasons: their types are more precise, and their immutability allows them to be allocated on the stack instead of the heap. However, this more precise typing comes with both more compile-time overhead and more difficulty achieving type stability.

## Examples

### Introduction to Tuples

`Tuple`s are immutable ordered collections of arbitrary distinct objects, either of the same type or of different types. Typically, tuples are constructed using the `(x, y)` syntax.

```
julia> tup = (1, 1.0, "Hello, World!")
(1,1.0,"Hello, World!")
```

The individual objects of a tuple can be retrieved using indexing syntax:

```
julia> tup[1]
1

julia> tup[2]
1.0

julia> tup[3]
"Hello, World!"
```

They implement the iterable interface, and can therefore be iterated over using `for` loops:

```
julia> for item in tup
```

```
            println(item)
        end
1
1.0
Hello, World!
```

Tuples also support a variety of generic collections functions, such as `reverse` or `length`:

```
julia> reverse(tup)
("Hello, World!",1.0,1)

julia> length(tup)
3
```

Furthermore, tuples support a variety of higher-order collections operations, including `any`, `all`, `map`, or `broadcast`:

```
julia> map(typeof, tup)
(Int64,Float64,String)

julia> all(x -> x < 2, (1, 2, 3))
false

julia> all(x -> x < 4, (1, 2, 3))
true

julia> any(x -> x < 2, (1, 2, 3))
true
```

The empty tuple can be constructed using `()`:

```
julia> ()
()

julia> isempty(ans)
true
```

However, to construct a tuple of one element, a trailing comma is required. This is because the parentheses (`(` and `)`) would otherwise be treated as grouping operations together instead of constructing a tuple.

```
julia> (1)
1

julia> (1,)
(1,)
```

For consistency, a trailing comma is also allowed for tuples with more than one element.

```
julia> (1, 2, 3,)
(1,2,3)
```

## Tuple types

The `typeof` a tuple is a subtype of `Tuple`:

```
julia> typeof((1, 2, 3))
Tuple{Int64,Int64,Int64}

julia> typeof((1.0, :x, (1, 2)))
Tuple{Float64,Symbol,Tuple{Int64,Int64}}
```

Unlike other data types, `Tuple` types are covariant. Other data types in Julia are generally invariant. Thus,

```
julia> Tuple{Int, Int} <: Tuple{Number, Number}
true

julia> Vector{Int} <: Vector{Number}
false
```

This is the case because everywhere a `Tuple{Number, Number}` is accepted, so too would a `Tuple{Int, Int}`, since it also has two elements, both of which are numbers. That is not the case for a `Vector{Int}` versus a `Vector{Number}`, as a function accepting a `Vector{Number}` may wish to store a floating point (e.g. `1.0`) or a complex number (e.g. `1+3im`) in such a vector.

The covariance of tuple types means that `Tuple{Number}` (again unlike `Vector{Number}`) is actually an abstract type:

```
julia> isleaftype(Tuple{Number})
false

julia> isleaftype(Vector{Number})
true
```

Concrete subtypes of `Tuple{Number}` include `Tuple{Int}`, `Tuple{Float64}`, `Tuple{Rational{BigInt}}`, and so forth.

`Tuple` types may contain a terminating `Vararg` as their last parameter to indicate an indefinite number of objects. For instance, `Tuple{Vararg{Int}}` is the type of all tuples containing any number of `Int`s, possibly zero:

```
julia> isa((), Tuple{Vararg{Int}})
true

julia> isa((1,), Tuple{Vararg{Int}})
true

julia> isa((1,2,3,4,5), Tuple{Vararg{Int}})
true

julia> isa((1.0,), Tuple{Vararg{Int}})
false
```

whereas `Tuple{String, Vararg{Int}}` accepts tuples consisting of a string, followed by any number (possibly zero) of `Int`s.

```
julia> isa(("x", 1, 2), Tuple{String, Vararg{Int}})
true

julia> isa((1, 2), Tuple{String, Vararg{Int}})
false
```

Combined with co-variance, this means that `Tuple{Vararg{Any}}` describes any tuple. Indeed, `Tuple{Vararg{Any}}` is just another way of saying `Tuple`:

```
julia> Tuple{Vararg{Any}} == Tuple
true
```

`Vararg` accepts a second numeric type parameter indicating how many times exactly its first type parameter should occur. (By default, if unspecified, this second type parameter is a typevar that can take any value, which is why any number of `Int`s are accepted in the `Vararg`s above.) `Tuple` types ending in a specified `Vararg` will automatically be expanded to the requested number of elements:

```
julia> Tuple{String,Vararg{Int, 3}}
Tuple{String,Int64,Int64,Int64}
```

Notation exists for homogenous tuples with a specified `Vararg`: `NTuple{N, T}`. In this notation, `N` denotes the number of elements in the tuple, and `T` denotes the type accepted. For instance,

```
julia> NTuple{3, Int}
Tuple{Int64,Int64,Int64}

julia> NTuple{10, Int}
NTuple{10,Int64}

julia> ans.types
svec(Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64)
```

Note that `NTuple`s beyond a certain size are shown simply as `NTuple{N, T}`, instead of the expanded `Tuple` form, but they are still the same type:

```
julia> Tuple{Int,Int,Int,Int,Int,Int,Int,Int,Int,Int}
NTuple{10,Int64}
```

## Dispatching on tuple types

Because Julia function parameter lists are themselves tuples, dispatching on various kinds of tuples is often easier done through the method parameters themselves, often with liberal usage for the "splatting" `...` operator. For instance, consider the implementation of `reverse` for tuples, from `Base`:

```
revargs() = ()
revargs(x, r...) = (revargs(r...)..., x)

reverse(t::Tuple) = revargs(t...)
```

Implementing methods on tuples this way preserves type stability, which is crucial for performance. We can see that there is no overhead to this approach using the `@code_warntype` macro:

```
julia> @code_warntype reverse((1, 2, 3))
Variables:
  #self#::Base.#reverse
  t::Tuple{Int64,Int64,Int64}

Body:
  begin
      SSAValue(1) = (Core.getfield)(t::Tuple{Int64,Int64,Int64},2)::Int64
      SSAValue(2) = (Core.getfield)(t::Tuple{Int64,Int64,Int64},3)::Int64
      return
(Core.tuple)(SSAValue(2),SSAValue(1),(Core.getfield)(t::Tuple{Int64,Int64,Int64},1)::Int64)::Tupl

  end::Tuple{Int64,Int64,Int64}
```

Although somewhat hard to read, the code here is simply getting creating a new tuple with values 3rd, 2nd, and 1st elements of the original tuple, respectively. On many machines, this compiles down to extremely efficient LLVM code, which consists of loads and stores.

```
julia> @code_llvm reverse((1, 2, 3))

define void @julia_reverse_71456([3 x i64]* noalias sret, [3 x i64]*) #0 {
top:
  %2 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 1
  %3 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 2
  %4 = load i64, i64* %3, align 1
  %5 = load i64, i64* %2, align 1
  %6 = getelementptr inbounds [3 x i64], [3 x i64]* %1, i64 0, i64 0
  %7 = load i64, i64* %6, align 1
  %.sroa.0.0..sroa_idx = getelementptr inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 0
  store i64 %4, i64* %.sroa.0.0..sroa_idx, align 8
  %.sroa.2.0..sroa_idx1 = getelementptr inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 1
  store i64 %5, i64* %.sroa.2.0..sroa_idx1, align 8
  %.sroa.3.0..sroa_idx2 = getelementptr inbounds [3 x i64], [3 x i64]* %0, i64 0, i64 2
  store i64 %7, i64* %.sroa.3.0..sroa_idx2, align 8
  ret void
}
```

## Multiple return values

Tuples are frequently used for multiple return values. Much of the standard library, including two of the functions of the iterable interface (`next` and `done`), returns tuples containing two related but distinct values.

The parentheses around tuples can be omitted in certain situations, making multiple return values easier to implement. For instance, we can create a function to return both positive and negative

square roots of a real number:

```
julia> pmsqrt(x::Real) = sqrt(x), -sqrt(x)
pmsqrt (generic function with 1 method)

julia> pmsqrt(4)
(2.0,-2.0)
```

Destructuring assignment can be used to unpack the multiple return values. To store the square roots in variables `a` and `b`, it suffices to write:

```
julia> a, b = pmsqrt(9.0)
(3.0,-3.0)

julia> a
3.0

julia> b
-3.0
```

Another example of this is the `divrem` and `fldmod` functions, which do an integer (truncating or floored, respectively) division and remainder operation at the same time:

```
julia> q, r = divrem(10, 3)
(3,1)

julia> q
3

julia> r
1
```

Read Tuples online: https://riptutorial.com/julia-lang/topic/6675/tuples

# Chapter 34: Type Stability

## Introduction

**Type instability** occurs when a variable's type can change at runtime, and hence cannot be inferred at compile-time. Type instability often causes performance problems, so being able to write and identify type-stable code is important.

## Examples

### Write type-stable code

```
function sumofsins1(n::Integer)
    r = 0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end

function sumofsins2(n::Integer)
    r = 0.0
    for i in 1:n
        r += sin(3.4)
    end
    return r
end
```

Timing the above two functions shows major differences in terms of time and memory allocations.

```
julia> @time [sumofsins1(100_000) for i in 1:100];
0.638923 seconds (30.12 M allocations: 463.094 MB, 10.22% gc time)

julia> @time [sumofsins2(100_000) for i in 1:100];
0.163931 seconds (13.60 k allocations: 611.350 KB)
```

This is because of type-unstable code in `sumofsins1` where the type of `r` needs to be checked for every iteration.

Read Type Stability online: https://riptutorial.com/julia-lang/topic/6084/type-stability

# Chapter 35: Types

## Syntax

- immutable MyType; field; field; end
- type MyType; field; field; end

## Remarks

Types are key to Julia's performance. An important idea for performance is type stability, which occurs when the type a function returns only depends on the types, not the values, of its arguments.

## Examples

### Dispatching on Types

On Julia, you can define more than one method for each function. Suppose we define three methods of the same function:

```
foo(x) = 1
foo(x::Number) = 2
foo(x::Int) = 3
```

When deciding what method to use (called dispatch), Julia chooses the more specific method that matches the types of the arguments:

```
julia> foo('one')
1

julia> foo(1.0)
2

julia> foo(1)
3
```

This facilitates polymorphism. For instance, we can easily create a linked list by defining two immutable types, named `Nil` and `Cons`. These names are traditionally used to describe an empty list and a non-empty list, respectively.

```
abstract LinkedList
immutable Nil <: LinkedList end
immutable Cons <: LinkedList
    first
    rest::LinkedList
end
```

We will represent the empty list by `Nil()` and any other lists by `Cons(first, rest)`, where `first` is the first element of the linked list and `rest` is the linked list consisting of all remaining elements. For example, the list `[1, 2, 3]` will be represented as

```
julia> Cons(1, Cons(2, Cons(3, Nil())))
Cons(1,Cons(2,Cons(3,Nil())))
```

## Is the list empty?

Suppose we want to extend the standard library's `isempty` function, which works on a variety of different collections:

```
julia> methods(isempty)
# 29 methods for generic function "isempty":
isempty(v::SimpleVector) at essentials.jl:180
isempty(m::Base.MethodList) at reflection.jl:394
...
```

We can simply use the function dispatch syntax, and define two additional methods of `isempty`. Since this function is from the `Base` module, we have to qualify it as `Base.isempty` in order to extend it.

```
Base.isempty(::Nil) = true
Base.isempty(::Cons) = false
```

Here, we did not need the argument values at all to determine whether the list is empty. Merely the type alone suffices to compute that information. Julia allows us to omit the names of arguments, keeping only their type annotation, if we need not use their values.

We can test that our `isempty` methods work:

```
julia> using Base.Test

julia> @test isempty(Nil())
Test Passed
  Expression: isempty(Nil())

julia> @test !isempty(Cons(1, Cons(2, Cons(3, Nil()))))
Test Passed
  Expression: !(isempty(Cons(1,Cons(2,Cons(3,Nil())))))
```

and indeed the number of methods for `isempty` have increased by `2`:

```
julia> methods(isempty)
# 31 methods for generic function "isempty":
isempty(v::SimpleVector) at essentials.jl:180
isempty(m::Base.MethodList) at reflection.jl:394
```

Clearly, determining whether a linked list is empty or not is a trivial example. But it leads up to something more interesting:

# How long is the list?

The `length` function from the standard library gives us the length of a collection or certain iterables. There are many ways to implement `length` for a linked list. In particular, using a `while` loop is likely fastest and most memory-efficient in Julia. But premature optimization is to be avoided, so let's suppose for a second that our linked list need not be efficient. What's the simplest way to write a `length` function?

```
Base.length(::Nil) = 0
Base.length(xs::Cons) = 1 + length(xs.rest)
```

The first definition is straightforward: an empty list has length `0`. The second definition is also easy to read: to count the length of a list, we count the first element, then count the length of the rest of the list. We can test this method similarly to how we tested `isempty`:

```
julia> @test length(Nil()) == 0
Test Passed
  Expression: length(Nil()) == 0
   Evaluated: 0 == 0

julia> @test length(Cons(1, Cons(2, Cons(3, Nil())))) == 3
Test Passed
  Expression: length(Cons(1,Cons(2,Cons(3,Nil())))) == 3
   Evaluated: 3 == 3
```

# Next steps

This toy example is pretty far from implementing all of the functionality that would be desired in a linked list. It is missing, for instance, the iteration interface. However, it illustrates how dispatch can be used to write short and clear code.

## Immutable Types

The simplest composite type is an immutable type. Instances of immutable types, like tuples, are values. Their fields cannot be changed after they are created. In many ways, an immutable type is like a `Tuple` with names for the type itself and for each field.

# Singleton types

Composite types, by definition, contain a number of simpler types. In Julia, this number can be zero; that is, an immutable type is allowed to contain *no* fields. This is comparable to the empty tuple `()`.

Why might this be useful? Such immutable types are known as "singleton types", as only one instance of them could ever exist. The values of such types are known as "singleton values". The standard library `Base` contains many such singleton types. Here is a brief list:

- `Void`, the type of `nothing`. We can verify that `Void.instance` (which is special syntax for retrieving the singleton value of a singleton type) is indeed `nothing`.
- Any media type, such as `MIME"text/plain"`, is a singleton type with a single instance, `MIME("text/plain")`.
- The `Irrational{:π}`, `Irrational{:e}`, `Irrational{:ψ}`, and similar types are singleton types, and their singleton instances are the irrational values `π = 3.1415926535897...`, etc.
- The iterator size traits `Base.HasLength`, `Base.HasShape`, `Base.IsInfinite`, and `Base.SizeUnknown` are all singleton types.

0.5.0

- In version 0.5 and later, each function is a singleton instance of a singleton type! Like any other singleton value, we can recover the function `sin`, for example, from `typeof(sin).instance`.

Because they contain nothing, singleton types are incredibly lightweight, and they can frequently be optimized away by the compiler to have no runtime overhead. Thus, they are perfect for traits, special tag values, and for things like functions that one would like to specialize on.

To define a singleton type,

```
julia> immutable MySingleton end
```

To define custom printing for the singleton type,

```
julia> Base.show(io::IO, ::MySingleton) = print(io, "sing")
```

To access the singleton instance,

```
julia> MySingleton.instance
MySingleton()
```

Often, one assigns this to a constant:

```
julia> const sing = MySingleton.instance
MySingleton()
```

## Wrapper types

If zero-field immutable types are interesting and useful, then perhaps one-field immutable types are even more useful. Such types are commonly called "wrapper types" because they wrap some underlying data, providing an alternative interface to said data. An example of a wrapper type in `Base` is `String`. We will define a similar type to `String`, named `MyString`. This type will be backed by a vector (one-dimensional array) of bytes (`UInt8`).

First, the type definition itself and some customized showing:

```
immutable MyString <: AbstractString
    data::Vector{UInt8}
end

function Base.show(io::IO, s::MyString)
    print(io, "MyString: ")
    write(io, s.data)
    return
end
```

Now our `MyString` type is ready for use! We can feed it some raw UTF-8 data, and it displays as we like it to:

```
julia> MyString([0x48,0x65,0x6c,0x6c,0x6f,0x2c,0x20,0x57,0x6f,0x72,0x6c,0x64,0x21])
MyString: Hello, World!
```

Obviously, this string type needs a lot of work before it becomes as usable as the `Base.String` type.

## True composite types

Perhaps most commonly, many immutable types contain more than one field. An example is the standard library `Rational{T}` type, which contains two fieds: a `num` field for the numerator, and a `den` field for the denominator. It is fairly straightforward to emulate this type design:

```
immutable MyRational{T}
    num::T
    den::T
    MyRational(n, d) = (g = gcd(n, d); new(n÷g, d÷g))
end
MyRational{T}(n::T, d::T) = MyRational{T}(n, d)
```

We have successfully implemented a constructor that simplifies our rational numbers:

```
julia> MyRational(10, 6)
MyRational{Int64}(5,3)
```

Read Types online: https://riptutorial.com/julia-lang/topic/5467/types

# Chapter 36: Unit Testing

## Syntax

- @test [expr]
- @test_throws [Exception] [expr]
- @testset "[name]" begin; [tests]; end
- Pkg.test([package])

## Remarks

The standard library documentation for `Base.Test` covers additional material beyond that shown in these examples.

## Examples

### Testing a Package

To run the unit tests for a package, use the `Pkg.test` function. For a package named `MyPackage`, the command would be

```
julia> Pkg.test("MyPackage")
```

An expected output would be similar to

```
INFO: Computing test dependencies for MyPackage...
INFO: Installing BaseTestNext v0.2.2
INFO: Testing MyPackage
Test Summary: | Pass  Total
  Data        |   66     66
Test Summary: | Pass  Total
  Monetary    |  107    107
Test Summary: | Pass  Total
  Basket      |   47     47
Test Summary: | Pass  Total
  Mixed       |   13     13
Test Summary: | Pass  Total
  Data Access |   35     35
INFO: MyPackage tests passed
INFO: Removing BaseTestNext v0.2.2
```

though obviously, one cannot expect it to match the above exactly, since different packages use different frameworks.

This command runs the package's `test/runtests.jl` file in a clean environment.

One can test all installed packages at once with

```
julia> Pkg.test()
```

but this usually takes a very long time.

## Writing a Simple Test

Unit tests are declared in the `test/runtests.jl` file in a package. Typically, this file begins

```
using MyModule
using Base.Test
```

The basic unit of testing is the `@test` macro. This macro is like an assertion of sorts. Any boolean expression can be tested in the `@test` macro:

```
@test 1 + 1 == 2
@test iseven(10)
@test 9 < 10 || 10 < 9
```

We can try out the `@test` macro in the REPL:

```
julia> using Base.Test

julia> @test 1 + 1 == 2
Test Passed
  Expression: 1 + 1 == 2
   Evaluated: 2 == 2

julia> @test 1 + 1 == 3
Test Failed
  Expression: 1 + 1 == 3
   Evaluated: 2 == 3
ERROR: There was an error during testing
 in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
 in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

The test macro can be used in just about anywhere, such as in loops or functions:

```
# For positive integers, a number's square is at least as large as the number
for i in 1:10
    @test i^2 ≥ i
end

# Test that no two of a, b, or c share a prime factor
function check_pairwise_coprime(a, b, c)
    @test gcd(a, b) == 1
    @test gcd(a, c) == 1
    @test gcd(b, c) == 1
end

check_pairwise_coprime(10, 23, 119)
```

## Writing a Test Set

0.5.0

In version v0.5, test sets are built into the standard library `Base.Test` module, and you don't have to do anything special (besides `using Base.Test`) to use them.

0.4.0

Test sets are not part of Julia v0.4's `Base.Test` library. Instead, you have to `REQUIRE` the `BaseTestNext` module, and add `using BaseTestNext` to your file. To support both version 0.4 and 0.5, you could use

```
if VERSION ≥ v"0.5.0-dev+7720"
    using Base.Test
else
    using BaseTestNext
    const Test = BaseTestNext
end
```

It is helpful to group related `@test`s together in a test set. In addition to clearer test organization, test sets offer better output and more customizability.

To define a test set, simply wrap any number of `@test`s with a `@testset` block:

```
@testset "+" begin
    @test 1 + 1 == 2
    @test 2 + 2 == 4
end

@testset "*" begin
    @test 1 * 1 == 1
    @test 2 * 2 == 4
end
```

Running these test sets prints the following output:

```
Test Summary: | Pass  Total
  +           |    2      2

Test Summary: | Pass  Total
  *           |    2      2
```

Even if a test set contains a failing test, the entire test set will be run to completion, and the failures will be recorded and reported:

```
@testset "-" begin
    @test 1 - 1 == 0
    @test 2 - 2 == 1
    @test 3 - () == 3
    @test 4 - 4 == 0
end
```

Running this test set results in

```
-: Test Failed
  Expression: 2 - 2 == 1
```

```
   Evaluated: 0 == 1
 in record(::Base.Test.DefaultTestSet, ::Base.Test.Fail) at ./test.jl:428
    ...
-: Error During Test
  Test threw an exception of type MethodError
  Expression: 3 - () == 3
  MethodError: no method matching -(::Int64, ::Tuple{})
    ...
Test Summary: | Pass  Fail  Error  Total
  -           |    2     1      1      4
ERROR: Some tests did not pass: 2 passed, 1 failed, 1 errored, 0 broken.
    ...
```

Test sets can be nested, allowing for arbitrarily deep organization

```
@testset "Int" begin
    @testset "+" begin
        @test 1 + 1 == 2
        @test 2 + 2 == 4
    end
    @testset "-" begin
        @test 1 - 1 == 0
    end
end
```

If the tests pass, then this will only show the results for the outermost test set:

```
Test Summary: | Pass  Total
  Int         |    3      3
```

But if the tests fail, then a drill-down into the exact test set and test causing the failure is reported.

The `@testset` macro can be used with a `for` loop to create many test sets at once:

```
@testset for i in 1:5
    @test 2i == i + i
    @test i^2 == i * i
    @test i ÷ i == 1
end
```

which reports

```
Test Summary: | Pass  Total
  i = 1       |    3      3
Test Summary: | Pass  Total
  i = 2       |    3      3
Test Summary: | Pass  Total
  i = 3       |    3      3
Test Summary: | Pass  Total
  i = 4       |    3      3
Test Summary: | Pass  Total
  i = 5       |    3      3
```

A common structure is to have outer test sets test components or types. Within these outer test sets, inner test sets test behaviour. For instance, suppose we created a type `UniversalSet` with a

singleton instance that contains everything. Before we even implement the type, we can use test-driven development principles and implement the tests:

```
@testset "UniversalSet" begin
    U = UniversalSet.instance
    @testset "egal/equal" begin
        @test U === U
        @test U == U
    end

    @testset "in" begin
        @test 1 in U
        @test "Hello World" in U
        @test Int in U
        @test U in U
    end

    @testset "subset" begin
        @test Set() ⊆ U
        @test Set(["Hello World"]) ⊆ U
        @test Set(1:10) ⊆ U
        @test Set([:a, 2.0, "w", Set()]) ⊆ U
        @test U ⊆ U
    end
end
```

We can then start implementing our functionality until it passes our tests. The first step is to define the type:

```
immutable UniversalSet <: Base.AbstractSet end
```

Only two of our tests pass right now. We can implement `in`:

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
```

This also makes some of our subset tests to pass. However, the `issubset` (⊆) fallback doesn't work for `UniversalSet`, because the fallback tries to iterate over elements, which we can't do. We can simply define a specialization that makes `issubset` return `true` for any set:

```
immutable UniversalSet <: Base.AbstractSet end
Base.in(x, ::UniversalSet) = true
Base.issubset(x::Base.AbstractSet, ::UniversalSet) = true
```

And now, all our tests pass!

## Testing Exceptions

Exceptions encountered while running a test will fail the test, and if the test is not in a test set, terminate the test engine. Usually, this is a good thing, because in most situations exceptions are not the desired result. But sometimes, one wants to test specifically that a certain exception is raised. The `@test_throws` macro facilitates this.

```
julia> @test_throws BoundsError [1, 2, 3][4]
Test Passed
  Expression: ([1,2,3])[4]
      Thrown: BoundsError
```

If the wrong exception is thrown, `@test_throws` will still fail:

```
julia> @test_throws TypeError [1, 2, 3][4]
Test Failed
  Expression: ([1,2,3])[4]
    Expected: TypeError
      Thrown: BoundsError
ERROR: There was an error during testing
 in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
 in do_test_throws(::Base.Test.Threw, ::Expr, ::Type{T}) at ./test.jl:329
```

and if no exception is thrown, `@test_throws` will fail also:

```
julia> @test_throws BoundsError [1, 2, 3, 4][4]
Test Failed
  Expression: ([1,2,3,4])[4]
    Expected: BoundsError
  No exception thrown
ERROR: There was an error during testing
 in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
 in do_test_throws(::Base.Test.Returned, ::Expr, ::Type{T}) at ./test.jl:329
```

## Testing Floating Point Approximate Equality

What's the deal with the following?

```
julia> @test 0.1 + 0.2 == 0.3
Test Failed
  Expression: 0.1 + 0.2 == 0.3
   Evaluated: 0.30000000000000004 == 0.3
ERROR: There was an error during testing
 in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
 in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

The error is caused by the fact that none of `0.1`, `0.2`, and `0.3` are represented in the computer as exactly those values — `1//10`, `2//10`, and `3//10`. Instead, they are approximated by values that are very close. But as seen in the test failure above, when adding two approximations together, the result can be a slightly worse approximation than is possible. There is much more to this subject that cannot be covered here.

But we aren't out of luck! To test that the combination of rounding to a floating point number and floating point arithmetic is *approximately* correct, even if not exact, we can use the `isapprox` function (which corresponds to operator `≈`). So we can rewrite our test as

```
julia> @test 0.1 + 0.2 ≈ 0.3
Test Passed
  Expression: 0.1 + 0.2 ≈ 0.3
   Evaluated: 0.30000000000000004 isapprox 0.3
```

Of course, if our code was entirely wrong, the test will still catch that:

```
julia> @test 0.1 + 0.2 ≈ 0.4
Test Failed
  Expression: 0.1 + 0.2 ≈ 0.4
   Evaluated: 0.30000000000000004 isapprox 0.4
ERROR: There was an error during testing
 in record(::Base.Test.FallbackTestSet, ::Base.Test.Fail) at ./test.jl:397
 in do_test(::Base.Test.Returned, ::Expr) at ./test.jl:281
```

The `isapprox` function uses heuristics based off the size of the numbers and the precision of the floating point type to determine the amount of error to be tolerated. It's not appropriate for all situations, but it works in most, and saves a lot of effort implementing one's own version of `isapprox`.

Read Unit Testing online: https://riptutorial.com/julia-lang/topic/5632/unit-testing

# Chapter 37: while Loops

## Syntax

- while cond; body; end
- break
- continue

## Remarks

The `while` loop does not have a value; although it can be used in expression position, its type is `Void` and the value obtained will be `nothing`.

## Examples

### Collatz sequence

The `while` loop runs its body as long as the condition holds. For instance, the following code computes and prints the Collatz sequence from a given number:

```
function collatz(n)
    while n ≠ 1
        println(n)
        n = iseven(n) ? n ÷ 2 : 3n + 1
    end
    println("1... and 4, 2, 1, 4, 2, 1 and so on")
end
```

Usage:

```
julia> collatz(10)
10
5
16
8
4
2
1... and 4, 2, 1, 4, 2, 1 and so on
```

It is possible to write any loop recursively, and for complex `while` loops, sometimes the recursive variant is more clear. However, in Julia, loops have some distinct advantages over recursion:

- Julia does not guarantee tail call elimination, so recursion uses additional memory and may cause stack overflow errors.
- And further, for the same reason, a loop can have decreased overhead and run faster.

### Run once before testing condition

Sometimes, one wants to run some initialization code once before testing a condition. In certain other languages, this kind of loop has special `do-while` syntax. However, this syntax can be replaced with a regular `while` loop and `break` statement, so Julia does not have specialized `do-while` syntax. Instead, one writes:

```
local name

# continue asking for input until satisfied
while true
    # read user input
    println("Type your name, without lowercase letters:")
    name = readline()

    # if there are no lowercase letters, we have our result!
    !any(islower, name) && break
end
```

Note that in some situations, such loops could be more clear with recursion:

```
function getname()
    println("Type your name, without lowercase letters:")
    name = readline()
    if any(islower, name)
        getname()  # this name is unacceptable; try again
    else
        name       # this name is good, return it
    end
end
```

## Breadth-first search

0.5.0

(Although this example is written using syntax introduced in version v0.5, it can work with few modifications on older versions also.)

This implementation of breadth-first search (BFS) on a graph represented with adjacency lists uses `while` loops and the `return` statement. The task we will solve is as follows: we have a sequence of people, and a sequence of friendships (friendships are mutual). We want to determine the degree of the connection between two people. That is, if two people are friends, we will return `1`; if one is a friend of a friend of the other, we will return `2`, and so on.

First, let's assume we already have an adjacency list: a `Dict` mapping `T` to `Array{T, 1}`, where the keys are people and the values are all the friends of that person. Here we can represent people with whatever type `T` we choose; in this example, we will use `Symbol`. In the BFS algorithm, we keep a queue of people to "visit", and mark their distance from the origin node.

```
function degree(adjlist, source, dest)
    distances = Dict(source => 0)
    queue = [source]

    # until the queue is empty, get elements and inspect their neighbours
```

```
    while !isempty(queue)
        # shift the first element off the queue
        current = shift!(queue)

        # base case: if this is the destination, just return the distance
        if current == dest
            return distances[dest]
        end

        # go through all the neighbours
        for neighbour in adjlist[current]
            # if their distance is not already known...
            if !haskey(distances, neighbour)
                # then set the distance
                distances[neighbour] = distances[current] + 1

                # and put into queue for later inspection
                push!(queue, neighbour)
            end
        end
    end

    # we could not find a valid path
    error("$source and $dest are not connected.")
end
```

Now, we will write a function to build an adjacency list given a sequence of people, and a sequence of `(person, person)` tuples:

```
function makeadjlist(people, friendships)
    # dictionary comprehension (with generator expression)
    result = Dict(p => eltype(people)[] for p in people)

    # deconstructing for; friendship is mutual
    for (a, b) in friendships
        push!(result[a], b)
        push!(result[b], a)
    end

    result
end
```

We can now define the original function:

```
degree(people, friendships, source, dest) =
    degree(makeadjlist(people, friendships), source, dest)
```

Now let's test our function on some data.

```
const people = [:jean, :javert, :cosette, :gavroche, :éponine, :marius]
const friendships = [
    (:jean, :cosette),
    (:jean, :marius),
    (:cosette, :éponine),
    (:cosette, :marius),
    (:gavroche, :éponine)
]
```

Jean is connected to himself in `0` steps:

```
julia> degree(people, friendships, :jean, :jean)
0
```

Jean and Cosette are friends, and so have degree `1`:

```
julia> degree(people, friendships, :jean, :cosette)
1
```

Jean and Gavroche are connected indirectly through Cosette and then Marius, so their degree is `3`:

```
julia> degree(people, friendships, :jean, :gavroche)
3
```

Javert and Marius are not connected through any chain, so an error is raised:

```
julia> degree(people, friendships, :javert, :marius)
ERROR: javert and marius are not connected.
 in degree(::Dict{Symbol,Array{Symbol,1}}, ::Symbol, ::Symbol) at ./REPL[28]:27
 in degree(::Array{Symbol,1}, ::Array{Tuple{Symbol,Symbol},1}, ::Symbol, ::Symbol) at
./REPL[30]:1
```

Read while Loops online: https://riptutorial.com/julia-lang/topic/5565/while-loops

# Credits

| S. No | Chapters | Contributors |
|---|---|---|
| 1 | Getting started with Julia Language | Andrew Piliser, becko, Community, Dawny33, Fengyang Wang, Kevin Montrose, prcastro |
| 2 | @goto and @label | Fengyang Wang |
| 3 | Arithmetic | Fengyang Wang |
| 4 | Arrays | Fengyang Wang, Michael Ohlrogge, prcastro |
| 5 | Closures | Fengyang Wang |
| 6 | Combinators | Fengyang Wang |
| 7 | Comparisons | Fengyang Wang |
| 8 | Comprehensions | 2Cubed, Fengyang Wang, zwlayer |
| 9 | Conditionals | Fengyang Wang, Michael Ohlrogge, prcastro |
| 10 | Cross-Version Compatibility | Fengyang Wang |
| 11 | Dictionaries | B Roy Dawson |
| 12 | Enums | Fengyang Wang |
| 13 | Expressions | Michael Ohlrogge |
| 14 | for Loops | Fengyang Wang, Michael Ohlrogge |
| 15 | Functions | Fengyang Wang, Harrison Grodin, Michael Ohlrogge, Sebastialonso |
| 16 | Higher-Order Functions | Fengyang Wang, mnoronha |
| 17 | Input | Fengyang Wang |
| 18 | Iterables | Fengyang Wang, prcastro |
| 19 | JSON | 4444, Fengyang Wang |
| 20 | Metaprogramming | Fengyang Wang, Ismael Venegas Castelló, P i, prcastro |

| | | |
|---|---|---|
| 21 | Modules | Fengyang Wang |
| 22 | Packages | Fengyang Wang |
| 23 | Parallel Processing | Fengyang Wang, Harrison Grodin, Michael Ohlrogge, prcastro |
| 24 | Reading a DataFrame from a file | Pranav Bhat |
| 25 | Regexes | Fengyang Wang |
| 26 | REPL | Fengyang Wang |
| 27 | Shell Scripting and Piping | 2Cubed, Fengyang Wang, mnoronha, prcastro |
| 28 | String Macros | Fengyang Wang |
| 29 | String Normalization | Fengyang Wang |
| 30 | Strings | Fengyang Wang, Michael Ohlrogge |
| 31 | sub2ind | Fengyang Wang, Gnimuc |
| 32 | Time | Fengyang Wang |
| 33 | Tuples | Fengyang Wang |
| 34 | Type Stability | Abhijith, Fengyang Wang |
| 35 | Types | Fengyang Wang, prcastro |
| 36 | Unit Testing | Fengyang Wang |
| 37 | while Loops | Fengyang Wang |