Functions

In Julia, a function is an object that maps a tuple of argument values to a return value. Julia functions are not pure mathematical functions, because they can alter and be affected by the global state of the program. The basic syntax for defining functions in Julia is:

```
julia> function f(x,y)
           x + y
       end
f (generic function with 1 method)
```

This function accepts two arguments `x` and `y` and returns the value of the last expression evaluated, which is `x + y`.

There is a second, more terse syntax for defining a function in Julia. The traditional function declaration syntax demonstrated above is equivalent to the following compact "assignment form":

```
julia> f(x,y) = x + y
f (generic function with 1 method)
```

In the assignment form, the body of the function must be a single expression, although it can be a compound expression (see Compound Expressions). Short, simple function definitions are common in Julia. The short function syntax is accordingly quite idiomatic, considerably reducing both typing and visual noise.

A function is called using the traditional parenthesis syntax:

```
julia> f(2,3)
5
```

Without parentheses, the expression `f` refers to the function object, and can be passed around like any other value:

```
julia> g = f;

julia> g(2,3)
5
```

As with variables, Unicode can also be used for function names:

```
julia> ∑(x,y) = x + y
∑ (generic function with 1 method)

julia> ∑(2, 3)
5
```

Argument Passing Behavior

Julia function arguments follow a convention sometimes called "pass-by-sharing", which means that values are not copied when they are passed to functions. Function arguments themselves act as new variable *bindings* (new locations that can refer to values), but the values they refer to are identical to the passed values. Modifications to mutable values (such as `Array`s) made within a function will be visible to the caller. This is the same behavior found in Scheme, most Lisps, Python, Ruby and Perl, among other dynamic languages.

The `return` Keyword

The value returned by a function is the value of the last expression evaluated, which, by default, is the last expression in the body of the function definition. In the example function, `f`, from the previous section this is the value of the expression `x + y`. As an alternative, as in many other languages, the `return` keyword causes a function to return immediately, providing an expression whose value is returned:

```
function g(x,y)
    return x * y
    x + y
end
```

Since function definitions can be entered into interactive sessions, it is easy to compare these definitions:

```
julia> f(x,y) = x + y
f (generic function with 1 method)

julia> function g(x,y)
           return x * y
           x + y
       end
```

```
g (generic function with 1 method)

julia> f(2,3)
5

julia> g(2,3)
6
```

Of course, in a purely linear function body like `g`, the usage of `return` is pointless since the expression `x + y` is never evaluated and we could simply make `x * y` the last expression in the function and omit the `return`. In conjunction with other control flow, however, `return` is of real use. Here, for example, is a function that computes the hypotenuse length of a right triangle with sides of length `x` and `y`, avoiding overflow:

```
julia> function hypot(x,y)
           x = abs(x)
           y = abs(y)
           if x > y
               r = y/x
               return x*sqrt(1+r*r)
           end
           if y == 0
               return zero(x)
           end
           r = x/y
           return y*sqrt(1+r*r)
       end
hypot (generic function with 1 method)

julia> hypot(3, 4)
5.0
```

There are three possible points of return from this function, returning the values of three different expressions, depending on the values of `x` and `y`. The `return` on the last line could be omitted since it is the last expression.

Return type

A return type can be specified in the function declaration using the `::` operator. This converts the return value to the specified type.

```
julia> function g(x, y)::Int8
```

```
            return x * y
        end;

julia> typeof(g(1, 2))
Int8
```

This function will always return an `Int8` regardless of the types of `x` and `y`. See Type Declarations for more on return types.

Returning nothing

For functions that do not need to return a value (functions used only for some side effects), the Julia convention is to return the value `nothing`:

```
function printx(x)
    println("x = $x")
    return nothing
end
```

This is a *convention* in the sense that `nothing` is not a Julia keyword but a only singleton object of type `Nothing`. Also, you may notice that the `printx` function example above is contrived, because `println` already returns `nothing`, so that the `return` line is redundant.

There are two possible shortened forms for the `return nothing` expression. On the one hand, the `return` keyword implicitly returns `nothing`, so it can be used alone. On the other hand, since functions implicitly return their last expression evaluated, `nothing` can be used alone when it's the last expression. The preference for the expression `return nothing` as opposed to `return` or `nothing` alone is a matter of coding style.

Operators Are Functions

In Julia, most operators are just functions with support for special syntax. (The exceptions are operators with special evaluation semantics like `&&` and `||`. These operators cannot be functions since Short-Circuit Evaluation requires that their operands are not evaluated before evaluation of the operator.) Accordingly, you can also apply them using parenthesized argument lists, just as you would any other function:

```
julia> 1 + 2 + 3
6

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

```
6
```

The infix form is exactly equivalent to the function application form – in fact the former is parsed to produce the function call internally. This also means that you can assign and pass around operators such as `+` and `*` just like you would with other function values:

```
julia> f = +;

julia> f(1,2,3)
6
```

Under the name `f`, the function does not support infix notation, however.

Operators With Special Names

A few special expressions correspond to calls to functions with non-obvious names. These are:

Expression	Calls
`[A B C ...]`	`hcat`
`[A; B; C; ...]`	`vcat`
`[A B; C D; ...]`	`hvcat`
`A'`	`adjoint`
`A[i]`	`getindex`
`A[i] = x`	`setindex!`
`A.n`	`getproperty`
`A.n = x`	`setproperty!`

Anonymous Functions

Functions in Julia are first-class objects: they can be assigned to variables, and called using the standard function call syntax from the variable they have been assigned to. They can be used as arguments, and they can be returned as values. They can also be created anonymously, without being given a name, using either of these syntaxes:

```
julia> x -> x^2 + 2x - 1
#1 (generic function with 1 method)

julia> function (x)
           x^2 + 2x - 1
       end
#3 (generic function with 1 method)
```

This creates a function taking one argument `x` and returning the value of the polynomial `x^2 + 2x - 1` at that value. Notice that the result is a generic function, but with a compiler-generated name based on consecutive numbering.

The primary use for anonymous functions is passing them to functions which take other functions as arguments. A classic example is `map`, which applies a function to each value of an array and returns a new array containing the resulting values:

```
julia> map(round, [1.2, 3.5, 1.7])
3-element Array{Float64,1}:
 1.0
 4.0
 2.0
```

This is fine if a named function effecting the transform already exists to pass as the first argument to `map`. Often, however, a ready-to-use, named function does not exist. In these situations, the anonymous function construct allows easy creation of a single-use function object without needing a name:

```
julia> map(x -> x^2 + 2x - 1, [1, 3, -1])
3-element Array{Int64,1}:
  2
 14
 -2
```

An anonymous function accepting multiple arguments can be written using the syntax `(x,y,z)->2x+y-z`. A zero-argument anonymous function is written as `()->3`. The idea of a function with no arguments may seem strange, but is useful for "delaying" a computation. In this usage, a block of code is wrapped in a zero-argument function, which is later invoked by calling it as `f`.

As an example, consider this call to `get`:

```
get(dict, key) do
    # default value calculated here
```

```
    time()
end
```

The code above is equivalent to calling `get` with an anonymous function containing the code enclosed between `do` and `end`, like so:

```
get(()->time(), dict, key)
```

The call to `time` is delayed by wrapping it in a 0-argument anonymous function that is called only when the requested key is absent from `dict`.

Tuples

Julia has a built-in data structure called a *tuple* that is closely related to function arguments and return values. A tuple is a fixed-length container that can hold any values, but cannot be modified (it is *immutable*). Tuples are constructed with commas and parentheses, and can be accessed via indexing:

```
julia> (1, 1+1)
(1, 2)

julia> (1,)
(1,)

julia> x = (0.0, "hello", 6*7)
(0.0, "hello", 42)

julia> x[2]
"hello"
```

Notice that a length-1 tuple must be written with a comma, `(1,)`, since `(1)` would just be a parenthesized value. `()` represents the empty (length-0) tuple.

Named Tuples

The components of tuples can optionally be named, in which case a *named tuple* is constructed:

```
julia> x = (a=2, b=1+2)
(a = 2, b = 3)

julia> x[1]
2
```

```
julia> x.a
2
```

Named tuples are very similar to tuples, except that fields can additionally be accessed by name using dot syntax (`x.a`) in addition to the regular indexing syntax (`x[1]`).

Multiple Return Values

In Julia, one returns a tuple of values to simulate returning multiple values. However, tuples can be created and destructured without needing parentheses, thereby providing an illusion that multiple values are being returned, rather than a single tuple value. For example, the following function returns a pair of values:

```
julia> function foo(a,b)
           a+b, a*b
       end
foo (generic function with 1 method)
```

If you call it in an interactive session without assigning the return value anywhere, you will see the tuple returned:

```
julia> foo(2,3)
(5, 6)
```

A typical usage of such a pair of return values, however, extracts each value into a variable. Julia supports simple tuple "destructuring" that facilitates this:

```
julia> x, y = foo(2,3)
(5, 6)

julia> x
5

julia> y
6
```

You can also return multiple values using the `return` keyword:

```
function foo(a,b)
    return a+b, a*b
```

```
end
```

This has the exact same effect as the previous definition of `foo`.

Argument destructuring

The destructuring feature can also be used within a function argument. If a function argument name is written as a tuple (e.g. `(x, y)`) instead of just a symbol, then an assignment `(x, y) = argument` will be inserted for you:

```
julia> minmax(x, y) = (y < x) ? (y, x) : (x, y)

julia> gap((min, max)) = max - min

julia> gap(minmax(10, 2))
8
```

Notice the extra set of parentheses in the definition of `gap`. Without those, `gap` would be a two-argument function, and this example would not work.

Varargs Functions

It is often convenient to be able to write functions taking an arbitrary number of arguments. Such functions are traditionally known as "varargs" functions, which is short for "variable number of arguments". You can define a varargs function by following the last positional argument with an ellipsis:

```
julia> bar(a,b,x...) = (a,b,x)
bar (generic function with 1 method)
```

The variables `a` and `b` are bound to the first two argument values as usual, and the variable `x` is bound to an iterable collection of the zero or more values passed to `bar` after its first two arguments:

```
julia> bar(1,2)
(1, 2, ())

julia> bar(1,2,3)
(1, 2, (3,))

julia> bar(1, 2, 3, 4)
(1, 2, (3, 4))
```

```
julia> bar(1,2,3,4,5,6)
(1, 2, (3, 4, 5, 6))
```

In all these cases, `x` is bound to a tuple of the trailing values passed to `bar`.

It is possible to constrain the number of values passed as a variable argument; this will be discussed later in Parametrically-constrained Varargs methods.

On the flip side, it is often handy to "splat" the values contained in an iterable collection into a function call as individual arguments. To do this, one also uses `...` but in the function call instead:

```
julia> x = (3, 4)
(3, 4)

julia> bar(1,2,x...)
(1, 2, (3, 4))
```

In this case a tuple of values is spliced into a varargs call precisely where the variable number of arguments go. This need not be the case, however:

```
julia> x = (2, 3, 4)
(2, 3, 4)

julia> bar(1,x...)
(1, 2, (3, 4))

julia> x = (1, 2, 3, 4)
(1, 2, 3, 4)

julia> bar(x...)
(1, 2, (3, 4))
```

Furthermore, the iterable object splatted into a function call need not be a tuple:

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

julia> bar(1,2,x...)
(1, 2, (3, 4))
```

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

julia> bar(x...)
(1, 2, (3, 4))
```

Also, the function that arguments are splatted into need not be a varargs function (although it often is):

```
julia> baz(a,b) = a + b;

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

julia> baz(args...)
3

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

julia> baz(args...)
ERROR: MethodError: no method matching baz(::Int64, ::Int64, ::Int64)
Closest candidates are:
  baz(::Any, ::Any) at none:1
```

As you can see, if the wrong number of elements are in the splatted container, then the function call will fail, just as it would if too many arguments were given explicitly.

Optional Arguments

It is often possible to provide sensible default values for function arguments. This can save users from having to pass every argument on every call. For example, the function `Date(y, [m, d])` from `Dates` module constructs a `Date` type for a given year `y`, month `m` and day `d`. However, `m` and `d` arguments are optional and their default value is `1`. This behavior can be expressed concisely as:

```
function Date(y::Int64, m::Int64=1, d::Int64=1)
    err = validargs(Date, y, m, d)
    err === nothing || throw(err)
    return Date(UTD(totaldays(y, m, d)))
end
```

Observe, that this definition calls another method of the `Date` function that takes one argument of type `UTInstant{Day}`.

With this definition, the function can be called with either one, two or three arguments, and `1` is automatically passed when only one or two of the arguments are specified:

```
julia> using Dates

julia> Date(2000, 12, 12)
2000-12-12

julia> Date(2000, 12)
2000-12-01

julia> Date(2000)
2000-01-01
```

Optional arguments are actually just a convenient syntax for writing multiple method definitions with different numbers of arguments (see Note on Optional and keyword Arguments). This can be checked for our `Date` function example by calling `methods` function.

Keyword Arguments

Some functions need a large number of arguments, or have a large number of behaviors. Remembering how to call such functions can be difficult. Keyword arguments can make these complex interfaces easier to use and extend by allowing arguments to be identified by name instead of only by position.

For example, consider a function `plot` that plots a line. This function might have many options, for controlling line style, width, color, and so on. If it accepts keyword arguments, a possible call might look like `plot(x, y, width=2)`, where we have chosen to specify only line width. Notice that this serves two purposes. The call is easier to read, since we can label an argument with its meaning. It also becomes possible to pass any subset of a large number of arguments, in any order.

Functions with keyword arguments are defined using a semicolon in the signature:

```
function plot(x, y; style="solid", width=1, color="black")
    ###
end
```

When the function is called, the semicolon is optional: one can either call `plot(x, y, width=2)` or `plot(x, y; width=2)`, but the former style is more common. An explicit semicolon is required only for passing varargs or computed keywords as described below.

Keyword argument default values are evaluated only when necessary (when a corresponding keyword argument is not passed), and in left-to-right order. Therefore default expressions may refer to prior keyword arguments.

The types of keyword arguments can be made explicit as follows:

```
function f(;x::Int=1)
    ###
end
```

Keyword arguments can also be used in varargs functions:

```
function plot(x...; style="solid")
    ###
end
```

Extra keyword arguments can be collected using `...`, as in varargs functions:

```
function f(x; y=0, kwargs...)
    ###
end
```

Inside `f`, `kwargs` will be a key-value iterator over a named tuple. Named tuples (as well as dictionaries with keys of `Symbol`) can be passed as keyword arguments using a semicolon in a call, e.g. `f(x, z=1; kwargs...)`.

If a keyword argument is not assigned a default value in the method definition, then it is *required*: an `UndefKeywordError` exception will be thrown if the caller does not assign it a value:

```
function f(x; y)
    ###
end
f(3, y=5) # ok, y is assigned
```

```
f(3)      # throws UndefKeywordError(:y)
```

One can also pass `key => value` expressions after a semicolon. For example, `plot(x, y; :width => 2)` is equivalent to `plot(x, y, width=2)`. This is useful in situations where the keyword name is computed at runtime.

When a bare identifier or dot expression occurs after a semicolon, the keyword argument name is implied by the identifier or field name. For example `plot(x, y; width)` is equivalent to `plot(x, y; width=width)` and `plot(x, y; options.width)` is equivalent to `plot(x, y; width=options.width)`.

The nature of keyword arguments makes it possible to specify the same argument more than once. For example, in the call `plot(x, y; options..., width=2)` it is possible that the `options` structure also contains a value for `width`. In such a case the rightmost occurrence takes precedence; in this example, `width` is certain to have the value `2`. However, explicitly specifying the same keyword argument multiple times, for example `plot(x, y, width=2, width=3)`, is not allowed and results in a syntax error.

Evaluation Scope of Default Values

When optional and keyword argument default expressions are evaluated, only *previous* arguments are in scope. For example, given this definition:

```
function f(x, a=b, b=1)
    ###
end
```

the `b` in `a=b` refers to a `b` in an outer scope, not the subsequent argument `b`.

Do-Block Syntax for Function Arguments

Passing functions as arguments to other functions is a powerful technique, but the syntax for it is not always convenient. Such calls are especially awkward to write when the function argument requires multiple lines. As an example, consider calling `map` on a function with several cases:

```
map(x->begin
           if x < 0 && iseven(x)
               return 0
           elseif x == 0
               return 1
```

```
            else
                return x
            end
        end,
    [A, B, C])
```

Julia provides a reserved word `do` for rewriting this code more clearly:

```
map([A, B, C]) do x
    if x < 0 && iseven(x)
        return 0
    elseif x == 0
        return 1
    else
        return x
    end
end
```

The `do x` syntax creates an anonymous function with argument `x` and passes it as the first argument to `map`. Similarly, `do a,b` would create a two-argument anonymous function, and a plain `do` would declare that what follows is an anonymous function of the form `() -> ...`.

How these arguments are initialized depends on the "outer" function; here, `map` will sequentially set `x` to `A`, `B`, `C`, calling the anonymous function on each, just as would happen in the syntax `map(func, [A, B, C])`.

This syntax makes it easier to use functions to effectively extend the language, since calls look like normal code blocks. There are many possible uses quite different from `map`, such as managing system state. For example, there is a version of `open` that runs code ensuring that the opened file is eventually closed:

```
open("outfile", "w") do io
    write(io, data)
end
```

This is accomplished by the following definition:

```
function open(f::Function, args...)
    io = open(args...)
    try
        f(io)
    finally
```

```
        close(io)
    end
end
```

Here, `open` first opens the file for writing and then passes the resulting output stream to the anonymous function you defined in the `do ... end` block. After your function exits, `open` will make sure that the stream is properly closed, regardless of whether your function exited normally or threw an exception. (The `try/finally` construct will be described in Control Flow.)

With the `do` block syntax, it helps to check the documentation or implementation to know how the arguments of the user function are initialized.

A `do` block, like any other inner function, can "capture" variables from its enclosing scope. For example, the variable `data` in the above example of `open...do` is captured from the outer scope. Captured variables can create performance challenges as discussed in performance tips.

Function composition and piping

Functions in Julia can be combined by composing or piping (chaining) them together.

Function composition is when you combine functions together and apply the resulting composition to arguments. You use the function composition operator (`∘`) to compose the functions, so `(f ∘ g)(args...)` is the same as `f(g(args...))`.

You can type the composition operator at the REPL and suitably-configured editors using `\circ<tab>`.

For example, the `sqrt` and `+` functions can be composed like this:

```
julia> (sqrt ∘ +)(3, 6)
3.0
```

This adds the numbers first, then finds the square root of the result.

The next example composes three functions and maps the result over an array of strings:

```
julia> map(first ∘ reverse ∘ uppercase, split("you can compose functions like this"))
6-element Array{Char,1}:
 'U': ASCII/Unicode U+0055 (category Lu: Letter, uppercase)
 'N': ASCII/Unicode U+004E (category Lu: Letter, uppercase)
 'E': ASCII/Unicode U+0045 (category Lu: Letter, uppercase)
 'S': ASCII/Unicode U+0053 (category Lu: Letter, uppercase)
 'E': ASCII/Unicode U+0045 (category Lu: Letter, uppercase)
```

```
 'S': ASCII/Unicode U+0053 (category Lu: Letter, uppercase)
```

Function chaining (sometimes called "piping" or "using a pipe" to send data to a subsequent function) is when you apply a function to the previous function's output:

```
julia> 1:10 |> sum |> sqrt
7.416198487095663
```

Here, the total produced by `sum` is passed to the `sqrt` function. The equivalent composition would be:

```
julia> (sqrt ∘ sum)(1:10)
7.416198487095663
```

The pipe operator can also be used with broadcasting, as `.|>`, to provide a useful combination of the chaining/piping and dot vectorization syntax (described next).

```
julia> ["a", "list", "of", "strings"] .|> [uppercase, reverse, titlecase, length]
4-element Array{Any,1}:
  "A"
  "tsil"
  "Of"
 7
```

Dot Syntax for Vectorizing Functions

In technical-computing languages, it is common to have "vectorized" versions of functions, which simply apply a given function `f(x)` to each element of an array `A` to yield a new array via `f(A)`. This kind of syntax is convenient for data processing, but in other languages vectorization is also often required for performance: if loops are slow, the "vectorized" version of a function can call fast library code written in a low-level language. In Julia, vectorized functions are *not* required for performance, and indeed it is often beneficial to write your own loops (see Performance Tips), but they can still be convenient. Therefore, *any* Julia function `f` can be applied elementwise to any array (or other collection) with the syntax `f.(A)`. For example, `sin` can be applied to all elements in the vector `A` like so:

```
julia> A = [1.0, 2.0, 3.0]
3-element Array{Float64,1}:
 1.0
 2.0
 3.0
```

```
julia> sin.(A)
3-element Array{Float64,1}:
 0.8414709848078965
 0.9092974268256817
 0.1411200080598672
```

Of course, you can omit the dot if you write a specialized "vector" method of `f`, e.g. via `f(A::AbstractArray) = map(f, A)`, and this is just as efficient as `f.(A)`. The advantage of the `f.(A)` syntax is that which functions are vectorizable need not be decided upon in advance by the library writer.

More generally, `f.(args...)` is actually equivalent to `broadcast(f, args...)`, which allows you to operate on multiple arrays (even of different shapes), or a mix of arrays and scalars (see Broadcasting). For example, if you have `f(x,y) = 3x + 4y`, then `f.(pi,A)` will return a new array consisting of `f(pi,a)` for each `a` in `A`, and `f.(vector1,vector2)` will return a new vector consisting of `f(vector1[i],vector2[i])` for each index `i` (throwing an exception if the vectors have different length).

```
julia> f(x,y) = 3x + 4y;

julia> A = [1.0, 2.0, 3.0];

julia> B = [4.0, 5.0, 6.0];

julia> f.(pi, A)
3-element Array{Float64,1}:
 13.42477796076938
 17.42477796076938
 21.42477796076938

julia> f.(A, B)
3-element Array{Float64,1}:
 19.0
 26.0
 33.0
```

Moreover, *nested* `f.(args...)` calls are *fused* into a single `broadcast` loop. For example, `sin.(cos.(X))` is equivalent to `broadcast(x -> sin(cos(x)), X)`, similar to `[sin(cos(x)) for x in X]`: there is only a single loop over `X`, and a single array is allocated for the result. [In contrast, `sin(cos(X))` in a typical "vectorized" language would first allocate one temporary array for `tmp=cos(X)`, and then compute `sin(tmp)` in a separate loop, allocating a second array.] This loop fusion is not a compiler optimization that may or may not occur, it is a *syntactic guarantee* whenever nested

`f.(args...)` calls are encountered. Technically, the fusion stops as soon as a "non-dot" function call is encountered; for example, in `sin.(sort(cos.(X)))` the `sin` and `cos` loops cannot be merged because of the intervening `sort` function.

Finally, the maximum efficiency is typically achieved when the output array of a vectorized operation is *pre-allocated*, so that repeated calls do not allocate new arrays over and over again for the results (see Pre-allocating outputs). A convenient syntax for this is `X .= ...`, which is equivalent to `broadcast!(identity, X, ...)` except that, as above, the `broadcast!` loop is fused with any nested "dot" calls. For example, `X .= sin.(Y)` is equivalent to `broadcast!(sin, X, Y)`, overwriting `X` with `sin.(Y)` in-place. If the left-hand side is an array-indexing expression, e.g. `X[begin+1:end] .= sin.(Y)`, then it translates to `broadcast!` on a `view`, e.g. `broadcast!(sin, view(X, firstindex(X)+1:lastindex(X)), Y)`, so that the left-hand side is updated in-place.

Since adding dots to many operations and function calls in an expression can be tedious and lead to code that is difficult to read, the macro `@.` is provided to convert *every* function call, operation, and assignment in an expression into the "dotted" version.

```
julia> Y = [1.0, 2.0, 3.0, 4.0];

julia> X = similar(Y); # pre-allocate output array

julia> @. X = sin(cos(Y)) # equivalent to X .= sin.(cos.(Y))
4-element Array{Float64,1}:
  0.5143952585235492
 -0.4042391538522658
 -0.8360218615377305
 -0.6080830096407656
```

Binary (or unary) operators like `.+` are handled with the same mechanism: they are equivalent to `broadcast` calls and are fused with other nested "dot" calls. `X .+= Y` etcetera is equivalent to `X .= X .+ Y` and results in a fused in-place assignment; see also dot operators.

You can also combine dot operations with function chaining using `|>`, as in this example:

```
julia> [1:5;] .|> [x->x^2, inv, x->2*x, -, isodd]
5-element Array{Real,1}:
    1
    0.5
    6
   -4
 true
```

Further Reading

We should mention here that this is far from a complete picture of defining functions. Julia has a sophisticated type system and allows multiple dispatch on argument types. None of the examples given here provide any type annotations on their arguments, meaning that they are applicable to all types of arguments. The type system is described in Types and defining a function in terms of methods chosen by multiple dispatch on run-time argument types is described in Methods.

« Strings Control Flow »