

# A Short and Incomplete Introduction to Julia

## Part 2: Functions

**Riccardo Murri** <[riccardo.murri@uzh.ch](mailto:riccardo.murri@uzh.ch)>

S3IT: Services and Support for Science IT,  
University of Zurich

# Functions

# Functions, I

Functions are called by postfixing the function name with a parenthesized argument list.

```
julia>    string(42)
"42"
julia>    string("x=", 42)
"x=42"
julia>    log10(1e42)
42.0
julia>    √(1764)    # \sqrt + TAB
42.0
```

## Functions, II

The `|>` operator allows an alternate syntax for function calls:

```
julia> 42 |> string  
"42"
```

This makes chained function calls more readable; compare:

```
julia> 42 |> string |> typeof  
String  
julia> typeof(string(42))  
String
```

## Functions, III

Some functions can take a variable number of arguments. For instance:

`max( $x_0, \dots, x_n$ )` Return the maximum of  $\{x_0, \dots, x_n\}$

`min( $x_0, \dots, x_n$ )` Return the minimum of  $\{x_0, \dots, x_n\}$

`string( $x_0, \dots, x_n$ )` Return the concatenation of printed representation of all arguments.

Examples:

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

```
julia> string("x=", 1, 2, 3)
"x=123"
```

# How to define new functions

A function definition is enclosed by the keyword **function** and a matching **end**.

```
"""  
A friendly function.  
"""  
  
function greet(name)  
    println("Hello, $(name)!")  
end  
  
# the customary greeting  
greet("world")
```

(This is a comment. It is  
ignored by Julia, just like  
blank lines.)

```
"""  
A friendly function.  
"""  
  
function greet(name)  
    println("Hello, $(name)!")  
end  
  
# the customary greeting  
greet("world")
```

This calls the function  
just defined.

```
"""  
A friendly function.  
"""  
function greet(name)  
    println("Hello, $(name)!")  
end  
  
# the customary greeting  
greet("world")
```



What is this? The answer  
in the next exercise!

```
"""  
A friendly function.  
"""  
function greet(name)  
    println("Hello, $(name)!")  
end  
  
# the customary greeting  
greet("world")
```

**Exercise 2.A:** Type and run the code on the previous page at the interactive prompt.

What's the result of evaluating the function  
`greet("world")`?

What does `?greet` output?

## Default values

Function arguments can  
have default values.

```
julia>
```

```
function greet( name="world" )  
    println("Hello, $(name)!")  
end
```

```
greet (generic function with 2 methods)
```

```
julia>    greet()  
Hello, world!
```

## The return value of functions

A function call returns the value of the expression evaluated last:

```
julia> function f()  
    1  
    2  
    3  
end  
f (generic function with 1 method)
```

```
julia> f()  
3
```

## The **return** keyword

The **return** keyword ends execution of a function.

The expression following 'return' sets the return value of the function being called.

```
julia> function f()
    println("entering f()...")
    return 2
    println("f() is done by now")
end
f (generic function with 1 method)
```

```
julia> f()
entering f()...
2
```

## The special value 'nothing'

If a `return` keyword is *not* followed by any expression, then the function returns a special value called 'nothing':

```
julia>    function noval()  
           return; #no expression to evaluate  
        end  
noval (generic function with 1 method)
```

The constant 'nothing' has its own special type 'Nothing':

```
julia>    typeof(noval())  
Nothing
```

## Shorthand definitions

Functions whose definition fits in a single expression can be created with a shorthand syntax:

```
julia> F(x, y) = (a-x)^2 + b*(y-x^2)^2
```

```
F (generic function with 1 method)
```

```
julia> F(-1,-1) # with a=1 b=2
```

```
12
```

This compact form is especially useful for mathematical functions but is not limited to them.

## Basic control flow



# Conditionals

Conditional execution uses the `if` statement:

```
if test-expr  
    # evaluate these expressions  
elseif other-test-expr  
    # evaluate these  
else  
    # executed if none of the above matched  
end
```

The `elseif` can be repeated, with different conditions, or left out entirely.

Also the `else` clause is optional.

## Truth values, I

Test expressions used in `if` and `elseif` clauses *must* evaluate to a boolean value: one of the two constants `true` or `false`.

All relational and comparison operators return boolean values:

```
julia> if 1 > 0
    println("A truism.")
else
    print("It's false!!")
end
A truism.
```

```
julia> if NaN == NaN
    println("A truism.")
else
    print("It's false!!")
end
It's false!
```

## Truth values, II

Test expressions used in `if` and `elseif` clauses *must* evaluate to a boolean value: one of the two constants `true` or `false`.

Integers, strings, or other types cannot be used in `if`'s test expression:

```
julia> if 1 #Python would allow this!
println("A truism.")
else
    print("It's false!!")
end
```

```
ERROR: TypeError: non-boolean (Int64) used
in boolean context
```

## The ternary operator

All code is an expression in Julia, so an ‘if’ block returns the value of the last expression evaluated:

```
julia> x = if (name == "Julia"); '♥' else '♠' end
'♠': Unicode U+2660 (category So: Symbol, other)
```

Testing and selecting one of two values is so common, it has an abbreviation:

```
julia> x = (name == "Julia") ? '♥' : '♠'
'♠': Unicode U+2660 (category So: Symbol, other)
```

Note that ‘?’ must be preceded by a space.

## while-loops

Conditional looping uses the `while` statement:

```
while test-expr  
    # expressions  
end
```

To break out of a `while` loop, use the `break` statement.

Use the `continue` statement anywhere in the expression block to jump back to the `while` statement.

Again, all code is an expression in Julia, so also a ‘`while`’ block returns the value of the last expression evaluated.

**Exercise 2.B:** Modify the `greet()` function to print '♥'\* if the argument `name` is the string `'Julia'`.

**Exercise 2.C:** Write a function `greetmany()` that repeatedly:

1. Asks the user for a name;
2. Prints a greeting to the person by that name.

The function should keep asking until the user enters the single letter 'q' as the name.

Julia's built-in function `readline()` can be used to input a string.

---

\*Use `\heartsuit` followed by the TAB key.

# Packages

## Using functions defined in another file, I

Julia code is organized into modules, files, and packages.

You can make functions from other packages usable from the current module with the `using` keyword:

```
julia> @sprintf("working?")
```

```
ERROR: LoadError: UndefVarError: @sprintf not defined
```

```
julia> using Printf
```

```
julia> @sprintf("working!")  
"working!"
```



## Using functions defined in another file, II

Using a package will *not* overwrite functions:

```
julia> function red(c)
           "Defined in Main"
       end
red (generic function with 1 method)
```

```
julia> using Colors
```

```
julia> red == Colors.red
false
```

```
julia> green == Colors.green
true
```

## Using functions defined in another file, III

To know in what module a function is defined, use the macro `@which`:

```
julia> @which red
Main
```

```
julia> @which green
ColorTypes
```

(Main is the module where Julia loads definitions by default.)

To list all symbols (functions and variables) defined in a module, use the function `names(ModuleName)`:

```
julia> names(Printf)
3-element Array{Symbol,1}:
 Symbol("@printf")
 Symbol("@sprintf")
  :Printf
```

# Installing packages

The standard package `Pkg` can be used to install new packages:

```
julia> using Pkg
```

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

```
Updating registry at `~/.julia/registries/General`
```

```
...
```

```
[no changes]
```

Note that `Pkg` does not export its functions, so to call them one has to prefix the name with `Pkg.`

For a better understanding of Julia's package system and its (many) nuances, see:

- ▶ [https://en.wikibooks.org/wiki/Introducing\\_Julia/Modules\\_and\\_packages](https://en.wikibooks.org/wiki/Introducing_Julia/Modules_and_packages) Wikibook “Introducing Julia” for a practical summary of all the options.
- ▶ <https://docs.julialang.org/en/v1/manual/modules/> Julia's manual section on “Modules” for details of the relationship between modules, files, and packages.
- ▶ <https://docs.julialang.org/en/v1/manual/code-loading/> Julia's manual section on “Code loading” for details on how files are organized in packages and how to define local packages.