Introducing Julia

en.wikibooks.org

July 25, 2015

Contents

0.1 Introduction

Julia is a recent arrival to the world of programming languages, and has yet to accumulate a profusion of introductory texts. This will surely change. Soon we might see:

- Julia in 24 hours
- Learn Julia the Hard Way
- Julia for Dummies
- Julia for Fun and Profit

and many others. But until then, here's a collection of notes and introductory paragraphs that comprise a gentle introduction to the Julia programming language, in the form of a wikibook.

The advantage and disadvantage of Wikibooks, apart from being free and open, is that anyone can edit anything at any time. In theory, a wikibook can only get better as more people add improvements and corrections. In practice, a wikibook may lose focus and consistency as it gains accuracy and coverage. But, because the Julia community has established a good ethos of encouraging participation in the development of the language, it's right that this wikibook is freely editable by everyone.

The official Julia documentation[1] is excellent, although aimed more at the early adopters, developers, and more experienced programmers. Refer to it as often as possible.

Much of the text in this wikibook should work with the current version of Julia, which is, as of September 2014, version 0.3.

1 `http://docs.julialang.org/en/release-0.3/`

1 Getting started

To install Julia on your computer, visit `http://julialang.org/downloads/` and follow instructions. If you'd rather use it online in a browser, visit `http://forio.com/julia/repl/` or `http://www.compileonline.com/execute_julia_online.php`, although the online Julia sites aren't always running at full efficiency.

Currently in beta and invite-only, the JuliaBox[1] looks a very promising environment for running Julia on a remote machine.

If you're familiar with coding in IPython[2] notebooks, you can use IJulia[3], a version of IPython that allows Julia notebooks. And keep an eye on Jupyter[4], a version of iPython that lets you write notebooks in Python, Julia, and R.

1.1 On Mac OS X

On a Mac, you download the Julia DMG, double-click to open it, and drag the icon to the Applications folder. To run Julia, you double-click the icon of the Julia package in the /Applications folder. This opens the terminal application, and starts a new window. This is the REPL:

```
$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "help()" to list help topics
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.3.0-rc1+60 (2014-07-17 19:50 UTC)
 _/ |\__'_|_|_|\__'_|  |  Commit a327b47* (36 days old master)
|__/                   |  x86_64-apple-darwin13.3.0

julia>
```

Alternatively, you can type, in a terminal:

```
$ /Applications/Julia-0.3.9.app/Contents/Resources/julia/bin/julia
```

— here you're specifying the path name of the Julia binary executable that lives inside the Julia application bundle. The current version is 0.3.9 — this may change in the next few weeks.

1 `http://juliabox.org`
2 `http://ipython.org`
3 `http://github.com/JuliaLang/IJulia.jl`
4 `http://jupyter.org`

1.1.1 Running directly from terminal

Typically, Julia is installed in `/Applications` , which isn't included in your PATH, and so the shell won't find it when you type `julia` on the command line.

But there are clever things you can do with paths and profiles, so that you can log in to a terminal and type `julia` with immediate success.

For example, after you find out the location of the Julia binary executable file, you can define the following alias:

```
alias julia="/Applications/Julia-0.3.9.app/Contents/Resources/julia/bin/julia"
```

Obviously this will have to be updated when the version number changes.

As an alternative, you could add the `/Applications/...` path to the PATH variable:

```
PATH="/Applications/Julia-0.3.9.app/Contents/Resources/julia/bin/:${PATH}"
export PATH
```

A different approach is to create a link to the executable and put it into the `/usr/local/bin` directory (which should be in your path), so that typing `julia` is the exact equivalent of typing `/Applications/Julia/.../julia` . This command does that:

```
ln -fs "/Applications/Julia-0.3.9.app/Contents/Resources/julia/bin/julia"
 /usr/local/bin/julia
```

Whichever method you choose, you can add the relevant command to your `~/.bash_profile` file to run every time you create a new shell.

You can use the shebang line at the top of a text file ('script') so that the shell can find Julia and execute the file:

```
#!/usr/bin/env julia
```

This also works in text editors, so that you can choose Run to run the file. This works if the editor reads the user's environment variables before running the file. (Not all do.)

1.1.2 Running a script with Julia

If you want to write Julia code in an editor and run it, in true scripting-language fashion, you can. At the top of the script file, add a line like the following:

```
#!/Applications/Julia-0.3.0.app/Contents/Resources/julia/bin/julia
```

where the pathname points to the right place on your system, somewhere inside the relevant Julia application bundle. This is the **shebang** line.

Now you can run the script inside the editor in the same way that you'd run a Perl script.

1.2 On Windows

On a Windows machine, you download the Julia Self-Extracting Archive (.exe) 32-bit or 64-bit. Double-click to start the installation process.

By default, it will install to your **AppData** folder. You may keep the default or choose your own directory (eg. C:\Julia).

After the installation has finished, you should create a System Environment variable called **JULIA_HOME** and set its value to the **\bin** directory under the folder where you installed Julia.

It is important to point **JULIA_HOME** to the /bin directory instead of the JULIA directory.

Then you can append **;%JULIA_HOME%** to your **PATH** System Environment variable, so you can run scripts from any directory.

1.3 On Linux

1.3.1 Installing from package

This is the easiest way to install Julia if you using Linux distributions based on RedHat, Fedora, Debian or Ubuntu. To install, download the respective package from the website, and install using your favorite way (double-clicking on the package file usually works). After doing this, Julia will be availabe from command line. On a terminal you can do:

```
$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "help()" to list help topics
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.3.0-rc1+60 (2014-07-17 19:50 UTC)
 _/ |\__'_|_|_|\__'_|  |  Commit a327b47* (36 days old master)
|__/                   |  x86_64-apple-darwin13.3.0

julia>
```

Installing from PPA (Ubuntu and derivatives)

On Ubuntu (and other distributions based on Ubuntu, like elementary or Linux Mint) you can install Julia even easier. You just have to add a repository and install julia from the command line terminal:

```
$ sudo add-apt-repository ppa:staticfloat/juliareleases
$ sudo apt-get install julia
```

With this, Julia will be updated together with the other software on your machine. Notice that, if you are logged as *root* , you don't have to use *sudo* on the commands.

To remove the package, run:

```
$ sudo apt-get purge julia
```

And to remove the repository, run:

```
$ sudo add-apt-repository --remove ppa:staticfloat/juliareleases
```

Make sure you remove the package before removing the repository.

Arch Linux

On Arch Linux, Julia is available from *community* repository, and can be installed running:

```
$ sudo pacman -S julia
```

To remove Julia package and it's dependencies (if not used by any other software on your system), you can run:

```
$ sudo pacman -Rsn julia
```

1.3.2 Using Binaries

You can use Julia direct from the binaries, without installing it on your machine. This is usefull if you have old Linux distributions or if you don't have administrator's access to the machine. Just download the binaries from the website, extract to a directory. While in this directory, enter the `bin` folder and run:

```
$ ./julia
```

If the program doesn't have permission to run, use the following command to give this permission:

```
$ chmod +x julia
```

In principle, this method could be used on any Linux distribution.

1.3.3 Running a script with Julia

To tell your operating system that it should run the script using Julia, you can use what is called the *sheebang* syntax. To do this, just use the following line on the very top of your script:

```
#!/usr/bin/env julia
```

With this as the first line of the script, the OS will search for ”julia” on the path, and use it to run the script.

2 The REPL

2.0.4 The REPL

The REPL is the Read/Evaluate/Print/Loop part of the language. It's an interactive command-line program that lets you type expressions in Julia code and see the results of the evaluation printed on the screen immediately. It:

- Reads what you type
- Evaluates it
- Prints out the return value, then
- Loops back and does it all over again

To be honest, it's not the best environment to do serious programming work of any scale in — for that, a text editor, or interactive notebook environment (e.g. IJulia/Jupyter) is a better choice. But there are advantages to using the REPL: it's simple, and should work without any installation or configuration. There's a bit of built-in help, too.

Using the REPL

You type some Julia code and then press Return/Enter. Julia evaluates what you typed and returns the result:

```
julia> 42 <Return/Enter>
42

julia>
```

If you don't want to see the result of the expression printed, use a semicolon at the end of the expression:

```
julia> 42;

julia>
```

Also, if you want to access the value of the last expression you typed on the REPL, it's stored within the variable `ans` :

```
julia> ans
42
```

If you don't complete the expression on the first line, continue typing until you finish. For example:

```
julia> 2 + <Return/Enter>
```

— now Julia waits patiently until you finish the expression:

```
2 <Return/Enter>
```

and then you'll see the answer:

```
4

julia>
```

Help within the REPL

Type a question mark `?`

```
julia> ?
```

and you'll immediately switch to Help mode, and the prompt changes to yellow:

```
help?>
```

Now you can type the name of something (function names should be written without parentheses):

```
help?> quit
Base.quit()

Quit the program indicating that the processes completed
succesfully. This function calls "exit(0)" (see "exit()").

julia>
```

You can also use the `help()` function, and supply the name of another function between the parentheses:

```
julia> help(quit)
```

You can try searching for anything using the `apropos()` function, and see a list of matches (but use quotes because you're searching for any string, not a function):

```
julia> apropos("quit")
Base.quit()
Base.exit([code])
Base.less(file::String[, line])
Base.less(function[, types])
Base.edit(file::String[, line])
Base.edit(function[, types])
```

The names or descriptions of all these functions contains the word "quit", even though some won't be applicable or useful right now.

Shell mode

If you type a semicolon

```
julia> ;
```

you immediately switch to shell mode, and the prompt changes to red:

```
shell>
```

In shell mode you can type any shell (ie non-Julia) command and see the result:

```
shell> ls
file.txt   executable.exe   directory file2.txt

julia>
```

then the prompt switches back to `julia>` , so you have to type a semicolon every time you want to give a shell command. The commands available within this mode are the ones used by your system's shell.

Orientation

Here are some other useful interactive functions and macros available at the REPL prompt:

- `whos()` - print information about the current global symbols
- `@which` - tells you which method will be called for a function and particular arguments:

```
julia> @which sin(3)
sin(x::Real) at math.jl:124
```

- `versioninfo()` - Julia version and platform information
- `edit("filename-in-current-directory")` - edit a file
- `@edit rand()` - edit the definition of the (built-in) function
- `less("filename-in-current-directory")` - displays a file

- `clipboard("stuff")` copies ”stuff” to the system clipboard
- `clipboard()` pastes the contents of the keyboard into the current REPL line
- `dump()` displays information about a Julia object on the screen
- `names()` Get an array of the names exported by a module or of the fields of a data type
- `workspace()` replace the top-level module (Main) with a new one and clean workspace

The `<TAB>` key: autocompletion

The TAB key is usually able to complete — or suggest a completion for — something whose name you start typing. For example, if I type w and then press the TAB key, all the functions that are currently available beginning with ’w’ are listed:

```
julia> w <TAB>
w
wait            whos                  with_rounding
writedlm
warn             widemul               workers
writemime
watch_file        widen                workspace
writesto
which            with_bigfloat_precision  write
wstring
while            with_bigfloat_rounding   writecsv
```

This works both for Julia entities and in shell mode. Here, for example, is how I can navigate to a directory from inside Julia:

```
shell> cd ~
/Users/me

shell> cd Doc <TAB>
shell> cd Documents/

shell> ls
...
```

Remember you can get help about these utility functions using `help()` .

History

You can look back through a record of your previous commands using the Up and Down arrow keys (and you can quit and restart without erasing that history). So you don’t have to type a long multi-line expression again, because you can just recall it from history. And if you’ve typed loads of expressions, you can search through them, both back and forwards, by pressing Ctrl-R and Ctrl-S.

Scope and performance

One warning about the REPL. The REPL operates at the global scope level of Julia. Usually, when writing longer code, you would put your code inside a function, and organise functions into modules and packages. Julia's compiler works much more effectively when your code is organized into functions, and your code will run much faster as a result. There are also some things that you can't do at the top level — such as specify types for the values of variables.

Changing the prompt

You can change the prompt from `julia>` to something else, such as `>` . Type:

```
julia> Base.active_repl.interface.modes[1].prompt="> "
"> "

>
```

2.0.5 Using Julia as a calculator

You can use Julia as a powerful calculator, using the REPL. It's good practice, too. (This is a tradition in introductions to interactive programming languages, and it's a good way to meet the language.)

```
julia> 2 + 2
4

julia> 2 + 3 + 4
9
```

An equivalent form for adding numbers is:

```
julia> +(2, 2)
4
```

The operators that you usually use between values are ordinary Julia functions, and can be used in the same way as other functions. Similarly:

```
julia> 2 + 3 + 4
9
```

can be written as

```
julia> +(2, 3, 4)
9
```

and

```
julia> 2 * 3 * 4
24
```

can be written as

```
julia> *(2,3,4)
24
```

Some maths constants are provided:

```
julia> pi
π = 3.1415926535897...

julia> golden
φ = 1.6180339887498...

julia> e
e = 2.7182818284590...
```

All the usual operators are provided:

```
julia> 2 + 3 - 4 * 5 / 6 % 7
1.6666666666666665
```

Notice the precedence of the operators. In this case it looks to be:

```
(2 + 3) - (4 * 5 / 6 % 7)
```

You'll sometimes need parentheses to control the evaluation order:

```
julia> (1 + sqrt(5)) / 2
1.618033988749895
```

Some others to watch out for include:

- `^` power
- `%` remainder

To make rational numbers, use two slashes (`//`):

```
julia> x = 666//999
2//3
```

There's also reverse division "\ ", so that `x/y` = `y\x` .

The standard arithmetic operators also have special updating versions, which you can use to update variables quickly:

- `+=`
- `-=`
- `*=`
- `/=`
- `\=`
- `%=`
- `^=`

For example, after defining a variable x:

```
julia> x = 5
5
```

you can add 2 to it like this:

```
julia> x += 2
7
```

multiply it by 100 like this:

```
julia> x *= 100
700
```

and find its modulus 11 value:

```
julia> x %= 11
7
```

There are element-wise operators which work on arrays. This means that you can multiply two arrays element by element:

```
julia> [2,4] .* [10, 20]
2-element Array{Int64,1}:
 20
 80
```

Arrays are fundamental to Julia, and so have their own chapter in this book.

2.0.6 Number bases

These handy utility functions might come in useful when using the REPL as a calculator.

The `bits()` function shows the literal binary representation of a number:

```
julia> bits(20.0)
”0100000000110100000000000000000000000000000000000000000000000000”

julia> bits(20)
”0000000000000000000000000000000000000000000000000000000000010100”
```

For working in bases other than the default 10, try `hex()` and `oct()` :

```
julia> hex(65535)
”ffff”
```

```
julia> oct(64)
”100”
```

and there are `base()` and `digits()` functions. The function `base(base, number)` converts a number to a string in the given base:

```
julia> base(16,255)
”ff”
```

Whereas `digits(number, base)` returns an array of the digits of number in the given base:

```
julia> digits(255,16)
2-element Array{Int64,1}:
 15
 15

julia>
```

Here’s a good place to mention `num2hex()` and `hex2num()` , functions used to convert hexadecimal strings to floating-point numbers and vice-versa.

2.0.7 Variables

In this expression:

```
julia> x = 3
```

`x` is a variable, a named storage location for a data object. In Julia, variables can be named pretty much how you like, although don’t start variable names with numbers or punctuation. You can use Unicode characters, if you want.

To assign a value, use a single equals sign.

```
julia> a = 1
1
```

```
julia> b = 2
2

julia> c = 3
3

julia> ✎= 3
3

julia> ✎
3
```

To test equality, one should use the `==` operator or `isequal()` function.

Julia's very good at spotting variable names and numbers in expressions:

```
julia> x = 2
2

julia> x^2+3x-1
9
```

You should avoid using names that Julia has already taken. For example, words like mean and median are used for imported function names, but Julia doesn't stop you redefining them — although you'll get a message warning you of possible problems ahead.

```
julia> mean = 0
Warning: imported binding for mean overwritten in module Main
0
```

If you make this mistake, you can restore the original meaning like this, if `mean` is a function in the Base module:

```
julia> mean = Base.mean
mean (generic function with 3 methods)
```

In Julia, you can also assign multiple variables at the same time:

```
julia> a, b = 5, 3
(5,3)
```

Notice that the return value of this expression is a parenthesis-bounded comma-separated ordered list of elements, or tuple for short.

```
julia> a
5

julia> b
3
```

2.0.8 Special characters

The Julia REPL provides easy access to special characters, such as Greek alphabetic characters, subscripts, and special maths symbols. If you type a backslash, you can then type a string (usually the equivalent LATEX string) to insert the corresponding Unicode character. For example, if you type:

```
julia> \sqrt <TAB>
```

Julia replaces the \sqrt with a square root symbol:

```
julia> √
```

Some other examples:

- \Gamma Γ
- \mercury ☿
- \degree °
- \cdot ·
- \in ∈

There's a full list in the Julia source code, which you can find at `julia/latex_symbols.jl` . As a general principle, in Julia you're encouraged to look at the source code, so there are useful built-in functions for looking at Julia source files:

```
julia> less("/Applications/Julia-0.3.0.app/Contents/Resources/julia/share/julia/base/latex_symbols.jl")
```

runs the file through a pager (ie the `less` command in Unix). If you're brave, try using `edit()` rather than `less()` .

2.0.9 Maths functions

Because Julia is particularly suited for scientific and technical computing, there are many mathematical functions that you can use immediately — you don't have to import them or use prefixes.

The trigonometry functions expect values in radians:

```
julia> sin(pi / 2)
1.0
```

but there are degree-based versions too: `sind(90)` finds the sine of 90 degrees. Use `deg2rad()` and `rad2deg()` to convert between degrees and radians.

There are also lots of log functions:

```
julia> log(12)
2.4849066497880004
```

and the accurate `hypot()` function:

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

The `norm()` function returns the "p"-norm of a vector or the operator norm of a matrix. Here's `divrem()` :

```
julia> divrem(13,3)   # returns the division and the remainder
(4,1)
```

There are dozens of others. Special functions include `erf()` , `dawson()` , `eta()` , `zeta()` , a full collection of Bessel functions, and so on.

In the statistics realm, all the basic statistics functions are available. For example:

```
julia> mean([1,2,3,4,5,6,7,8])
4.5
```

There's a system-wide variable called ans that remembers the most recent result, so that you can use it in the next expression.

```
julia> 1*2*3*4*5
120

julia> ans/10
12.0
```

2.0.10 Exercise

Guess, then find out using `help()` , what `isprime()` and `isapprox()` do.

Descriptions of all the functions provided as standard with Julia are described here: `http://julia.readthedocs.org/en/latest/stdlib/base/#the-standard-library`

2.0.11 Random numbers

`rand()` - one random Float64 between 0 and 1

```
julia> rand()
0.11258244478647295
```

`rand(2,2)` - an array of Float64s with dimensions 2,2

`rand(type, 2, 2)` - an array of values of this type with dims 2, 2

`rand(range, dims)` - array of numbers in a range (including both ends) with specified dimensions:

```
julia> rand(0:10, 6)
6-element Array{Int64,1}:
 6
 7
 9
 6
 3
 10
```

The `randbool()` function generates one or more trues and falses:

```
julia> randbool()
false

julia> randbool(20)
20-element BitArray{1}:
 false
 true
 false
 false
 false
 true
 true
 false
 false
 false
 false
 false
 false
 false
 true
 true
 false
 true
 true
 false
```

Random numbers in a distribution

`randn()` gives you one random number in a normal distribution with mean 0 and standard deviation 1. `randn(n)` gives you n such numbers:

```
julia> randn()
0.8060073309441075

julia> randn(10),
([1.31598,
1.55126,-1.14605,-0.562148,0.69725,0.468769,-1.58275,0.238471,2.72857,1.11561],)
```

(the comma after randn(10) is just intended for line visualization)

If you've installed the Gadfly plotting package, you can plot this:

```
julia> using Gadfly

julia> p = plot(x=randn(2000), Geom.histogram(bincount=100))
```

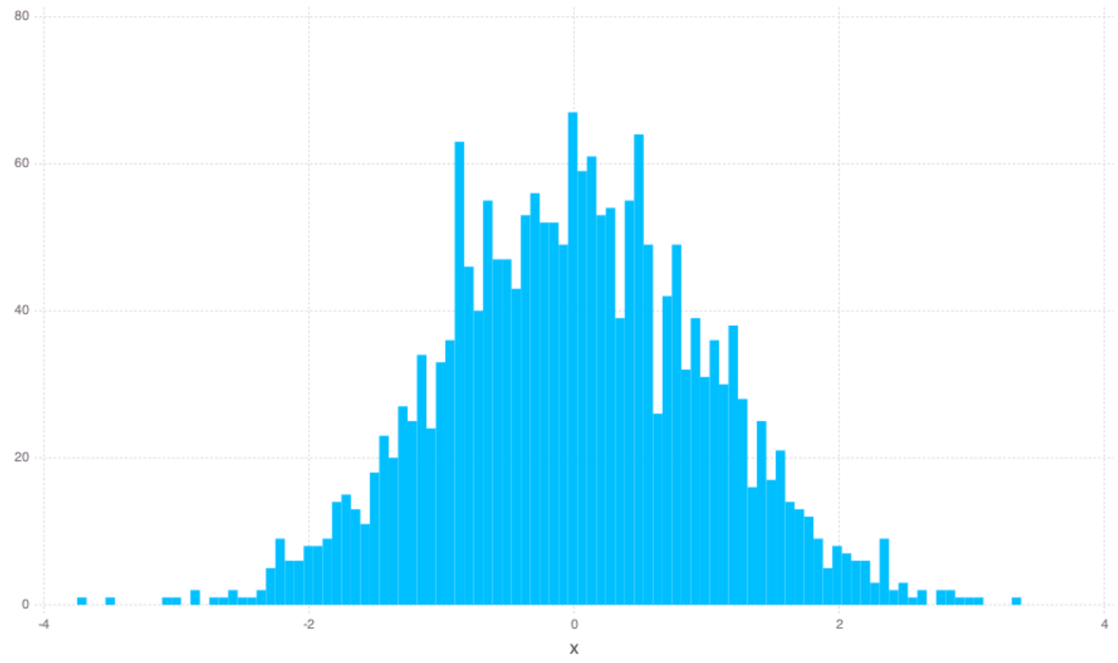

Figure 1 histogram plot created in Julia using Gadfly

2.0.12 Seeding the random number generator

Before you use random numbers, you can seed the random number generator with a specific value. This ensures that subsequent random numbers will follow the same sequence, if they start from the same seed. You can seed the generator using the `srand()` or `MersenneTwister()` functions.

```
julia> srand(10)

julia> rand(0:10, 6)
6-element Array{Int64,1}:
 6
 5
 9
 1
 1
 0
```

```
julia> rand(0:10, 6)
6-element Array{Int64,1}:
 10
 3
 6
 8
 0
 1
```

After restarting Julia, the same seed guarantees the same random numbers:

```
julia> exit()
$ julia
_
_       _ _(_)_     |  A fresh approach to technical computing
(_)     | (_) (_)    |  Documentation: http://docs.julialang.org
_ _   _| |_  __ _   |  Type "help()" for help.
| | | | | | |/ _` |  |
| | |_| | | | (_| |  |  Version 0.3.0 (2014-08-20 20:43 UTC)
_/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-apple-darwin13.3.0

julia> srand(10)

julia> rand(0:10, 6)
6-element Array{Int64,1}:
 6
 5
 9
 1
 1
 0

julia> rand(0:10, 6)
6-element Array{Int64,1}:
 10
 3
 6
 8
 0
 1
```

3 Arrays and Tuples

3.1 Storage: Arrays and Tuples

In Julia, groups of items are stored in arrays, tuples, or dictionaries.

3.1.1 Arrays

An array is an ordered collection of elements. It's indicated with square brackets or — occasionally — curly braces. You can create arrays that are full or empty, and arrays that hold values of any type or restricted to values of a specific type.

In Julia, arrays are used for lists, vectors, and matrices. A one-dimensional array is a vector or list. A 2-D array can be thought of as a matrix. And 3-D and more-D arrays are similarly thought of as multi-dimensional matrices.

3.1.2 Creating simple arrays

Here's how to create a simple one dimensional array:

```
julia> a = [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

Julia informs you that you've created a one-dimensional array with 5 elements, each of which is a 64-bit integer, and bound the variable a to it. Notice that intelligence is applied to the process: if one of the elements looks like a floating-point number, for example, you'll get an array of Float64s:

```
julia>a1 = [1, 2, 3.0, 4, 5]
5-element Array{Float64,1}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

Similarly for strings:

```
julia> s = ["this", "is", "an", "array", "of", "strings"]
6-element Array{ASCIIString,1}:
 "this"
 "is"
 "an"
 "array"
 "of"
 "strings"
```

returns an array of ASCII strings, and:

```
julia> trigfuns = [sin, cos, tan]
3-element Array{Function,1}:
 sin
 cos
 tan
```

returns an array of Julia functions.

You can specify both the type and the dimensions with the `Array()` function (notice that upper-case "A"):

```
julia> array = Array(Int64,5)
5-element Array{Int64,1}:
 0
 0
 0
 0
 0

julia> array3 = Array(Int64,2,2,2)
2x2x2 Array{Int64,3}:
[:, :, 1] =
 0  0
 0  0

[:, :, 2] =
 0  0
 0  0
```

It's possible to create arrays with elements of different types:

```
julia> [1,"2", 3.0, sin, pi]
5-element Array{Any, 1}:
     1
      "2"
     3.0
      sin
 π = 3.1415926535897...
```

Here, the array has five elements, but they're an odd mixture: numbers, strings, functions, constants — so Julia creates an array of type Any:

```
julia> typeof(ans)
Array{Any,1}
```

To create an array of a specific type, use the type definition and square brackets:

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

If you think you can fool Julia by sneaking in a value of the wrong type while declaring a typed array, you will get caught out:

```
julia> Int64[1, 2, 3, 4, 5, 6, 7, 8,  9, 10.0]
ERROR: InexactError()
 in setindex! at array.jl:307
 in getindex at array.jl:121
```

You can use any type to create empty arrays:

```
julia> b = Int64[]
0-element Array{Int64,1}

julia> b = Int64[10, 20, 30]
3-element Array{Int64,1}:
 10
 20
 30

julia> b = String[]
0-element Array{String,1}

julia> b = Float64[]
0-element Array{Float64,1}
```

Row vectors

If you leave out the commas when defining an array, Julia creates a single row, multi-column array — also called a row vector:

```
julia> [1 2 3 4]
1x4 Array{Int64,2}:
 1  2  3  4
```

and you can use a semicolon to add another row:

```
julia> [1 2 3 4 ; 5 6 7 8]
2x4 Array{Int64,2}:
 1  2  3  4
 5  6  7  8
```

Notice the `,2}` in the first row of the response.

There are a number of functions which let you create and fill an array in one go. See Creating and filling an array[1].

3.1.3 Range objects

In Julia, the colon (`:`) has a number of uses. One use is to define ranges and sequences of integers. You can create a range object by typing it directly:

```
julia> 1:20
1:20
```

1 Chapter 3.1.8 on page 29

Or you can use the `range()` function to make a range object:

```
julia> range(1,10)
1:10
```

It may not look very useful in that form, but it provides the raw material for any job in Julia that needs a range or sequence of numbers. For example, you can build an array consisting of those numbers, just by enclosing the range in square brackets:

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

Or you can use it in a loop expression:

```
julia> for n in 1:10 print(n) end
12345678910
```

You don't have to start and finish on an integer either:

```
julia> [3.5:9.5]
7-element Array{Float64,1}:
 3.5
 4.5
 5.5
 6.5
 7.5
 8.5
 9.5
```

There's also a three-piece version of a range object which lets you specify a step size other than 1. This builds an array with elements that go from 0 to 100 in steps of 10:

```
julia> [0:10:100]
11-element Array{Int64,1}:
   0
  10
  20
  30
  40
  50
  60
  70
  80
  90
 100
```

To go down instead of up, you'll have to use the middle step value:

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

Collecting up the values in a range

If you're not using your range object in a for loop, you can use `collect()` to obtain all the values from a range object directly:

```
julia> collect(0:5:100)
21-element Array{Int64,1}:
   0
   5
  10
  15
  20
  25
  30
  35
  40
  45
  50
  55
  60
  65
  70
  75
  80
  85
  90
  95
 100
```

(Although you could have used square brackets to build an array, in this case.)

3.1.4 Custom range objects

Another useful function is `linrange()` , which constructs a range object that goes from a start value to an end value taking a specific number of steps. You don't have to calculate the increment, Julia calculates the step size for you. For example, to go from 1 to 100 in exactly 12 steps:

```
julia> linrange(1,100,12)
1.0:9.0:100.0
```

You can use this range object to build an array:

```
julia> [linrange(1,100,12)]
12-element Array{Float64,1}:
   1.0
  10.0
  19.0
  28.0
  37.0
  46.0
  55.0
  64.0
  73.0
  82.0
  91.0
 100.0
```

`collect()` could work here too. Notice that it provided you with a Float64 array, rather than an Integer array. Similar functions are `linspace()` , which creates the array directly, and a logarithmic version called `logspace()` , here going from 10^1 to 10^2 in five steps:

```
julia> logspace(1, 2, 5)
5-element Array{Float64,1}:
  10.0
  17.7828
  31.6228
  56.2341
 100.0
```

3.1.5 Arrays, vectors, lists, and matrices

Compare these two: `[1,2,3,4,5]` and `[1 2 3 4 5]` .

With the commas, this array is a column vector, with 5 rows and 1 column:

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

With the spaces, this array is a row vector, with 1 row and 5 columns:

```
julia> [1 2 3 4 5]
1x5 Array{Int64,2}:
 1  2  3  4  5
```

With 2 dimensions, you can create matrices:

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

And of course you can create matrices with 3 or more dimensions.

3.1.6 Curly brace syntax

Note: This syntax has been deprecated in favor of Any[].

You can also use the curly braces to create arrays.

They're like square brackets, but the resulting array is of type Any.

Compare:

```
julia> aany = {1,2,3,4}
4-element Array{Any,1}:
 1
 2
 3
 4
```

with

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

The curly braces created an array of Any type, so you can change a value to another type without problems:

```
julia> aany[1] = "One"
"One"
julia> aany
4-element Array{Any,1}:
  "One"
 2
 3
 4
```

But you can't do this for the square-bracket-created array:

```
julia> aarr[1] = "One"
ERROR: `convert` has no method matching convert(::Type{Int64}, ::ASCIIString)
 in setindex! at array.jl:307
```

3.1.7 Creating 2D arrays

You **can't** create a 2D array by typing this:

```
julia> [[1,2,3], [4,5,6]]
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

because Julia massages it into a 6 element array. Instead, you can create a 2-D array by doing one of the following:

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

Here, you use spaces instead of commas, and semicolons to indicate the start of a new row.

Or you can specify each column in turn, like this:

```
julia> a = [[1,2,3] [4,5,6]]
3x2 Array{Int64,2}:
 1  4
 2  5
 3  6
```

The `Array` constructor constructs an array of arrays:

```
julia> Array[1:3, 4:6]
2-element Array{Array{T,N},1}:
 [1,2,3]
 [4,5,6]
```

The `{Array(T,N}` indicates that the types of the arrays haven't been specified.

Notice the difference between this Array constructor and the function `Array()` , which does this:

```
julia> Array(Int64, 3,2)
3x2 Array{Int64,2}:
 0  0
 0  0
 0  0
```

Since there's a useful function for reshaping arrays, you can of course create a simple array and then change its shape:

```
julia> reshape([1,2,3,4,5,6,7,8], 2, 4)
2x4 Array{Int64,2}:
 1  3  5  7
 2  4  6  8
```

The same techniques can be used to create 3D arrays. Here's a 3D array of strings:

```
julia> Array(String,2,3,4)
2x3x4 Array{String,3}:
[:, :, 1] =
 #undef  #undef  #undef
 #undef  #undef  #undef

[:, :, 2] =
 #undef  #undef  #undef
 #undef  #undef  #undef

[:, :, 3] =
 #undef  #undef  #undef
 #undef  #undef  #undef

[:, :, 4] =
 #undef  #undef  #undef
 #undef  #undef  #undef
```

Each element is `#undef` .

If you have an existing array and want to copy it, you can use the `similar()` function:

```
julia> a = [1:10]
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
julia> b = similar(a)
10-element Array{Int64,1}:
 0
 0
 0
 0
 0
 0
 0
 0
 0
 0
```

You can, though, change the type and dimensions anyway, so they don't have to be that similar:

```
julia> c = similar(b, String, (2, 2))
2x2 Array{String,2}:
 #undef  #undef
 #undef  #undef
```

3.1.8 Creating and filling an array

There are a number of functions that let you create arrays with specific contents. We've seen that you can enclose a range object (start:step:stop) in square brackets:

```
julia> [0:10:100]
11-element Array{Int64,1}:
   0
  10
  20
  30
  40
  50
  60
  70
  80
  90
 100
```

Some functions that you can use to make arrays with ready-to-use contents include `zeros()` , `ones()` , `trues()` , `falses()` , `fill()` , and `fill!()` :

```
julia> zeros(2, 2)
2x2 Array{Float64,2}:
 0.0  0.0
 0.0  0.0

julia> ones(Int64, (3, 3))
3x3 Array{Int64,2}:
 1  1  1
 1  1  1
 1  1  1
```

The `trues()` and `falses()` functions do a similar job with the Boolean values true and false.

```
julia> trues(3, 4)
3x4 BitArray{2}:
 true  true  true  true
 true  true  true  true
 true  true  true  true
```

You can use `fill()` to create an array with a specific value, i.e. an array of repeating duplicates:

```
julia> fill(42, 42),
([42,42,42,42,42,42,42,42,42,42  …  42,42,42,42,42,42,42,42,42,42],)  # 42 42s

julia> fill("hi", 2, 2)
2x2 Array{ASCIIString,2}:
 "hi"  "hi"
 "hi"  "hi"
```

With `fill!()` , the exclamation mark (`!`) or "bang" is to warn you that you're about to change the contents of an existing array (a useful indication that's adopted throughout Julia). Let's change an array of falses to trues:

```
julia> trueArray = falses(3,3)
3x3 BitArray{2}:
 false  false  false
 false  false  false
 false  false  false
julia> fill!(trueArray, true)
3x3 BitArray{2}:
 true  true  true
 true  true  true
 true  true  true
julia> trueArray
3x3 BitArray{2}:
 true  true  true
 true  true  true
 true  true  true
```

If you'd prefer a random number in each cell of a new array, use `rand()` followed by the required dimensions:

```
julia> rand(2,3)
2x3 Array{Float64,2}:
 0.662843  0.445892  0.439851
 0.121989  0.048896  0.870163
```

or use `randn()` for normally distributed numbers:

```
julia> randn(20,3)
20x3 Array{Float64,2}:
 -0.728709    -0.391909  -1.13824
  1.2293      -0.829367  -1.06711
  2.26544     -1.5573    -0.444163
  0.00163927  -0.987814  -0.887405
 -0.595679     1.45497   -0.651611
 -0.690074    -1.96845   -2.57402
 -0.705386     1.0319    -0.884835
  0.0640004    0.622827  -1.10468
  0.640549    -1.98737   -0.586064
 -2.14224     -0.129388  -1.50473
 -0.0834891    0.600715   3.00035
  0.059163    -0.310474  -0.84879
  0.471043    -0.168212   2.16704
  0.748387     1.06909    0.381324
  1.64173     -0.327076   0.489524
  0.872271    -1.00717   -1.56758
 -1.88053      0.462356   0.452252
 -0.318148    -1.08098   -0.00325938
 -1.23269     -0.126621  -0.300601
 -1.34337     -1.50194    0.0724766
```

Here's the identity matrix, I (called "eye", to avoid complexity):

```
julia> eye(3, 3)
3x3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

You can use `linspace()` and `logspace()` functions to create vector-like arrays, then use `reshape()` to make them into 2D arrays. `linspace(start, end, steps)` goes from start to end in steps steps.

```
julia> reshape(linspace(0, 100, 30), 10, 3)
10x3 Array{Float64,2}:
  0.0       34.4828   68.9655
  3.44828   37.931    72.4138
  6.89655   41.3793   75.8621
 10.3448    44.8276   79.3103
 13.7931    48.2759   82.7586
 17.2414    51.7241   86.2069
 20.6897    55.1724   89.6552
 24.1379    58.6207   93.1034
 27.5862    62.069    96.5517
 31.0345    65.5172  100.0
```

So this gives a 10 by 3 array featuring evenly spaced numbers between 0 and 100.

As mentioned above, the `logspace(a, b, n)` creates a logarithmically-spaced array of n numbers between 10^a and 10^b. Here, the `int()` function converts the numbers in the 5 by 4 array to integers.

```
julia> int(reshape(logspace(0, 10, 20), 5, 4))
5x4 Array{Int64,2}:
   1    428    183298     78475997
   3   1438    615848    263665090
  11   4833   2069138    885866790
  38  16238   6951928   2976351442
 127  54556  23357215  10000000000
```

You can make a diagonal matrix and put 1s in every diagonal slot, using `diagm()` . For example, put 1 into the diagonal of a 6 by 6 matrix:

```
julia> diagm([1,1,1,1,1,1])
6x6 Array{Int64,2}:
 1  0  0  0  0  0
 0  1  0  0  0  0
 0  0  1  0  0  0
 0  0  0  1  0  0
 0  0  0  0  1  0
 0  0  0  0  0  1
```

Using comprehensions

A useful way of creating arrays is to use comprehensions (described in Comprehensions[2]).

```
julia> [r * c for r in 1:5, c in 1:5]
5x5 Array{Int64,2}:
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
```

2 Chapter 5.1.4 on page 69

```
 5  10  15  20  25
```

Curly braces

Using curly braces instead of square brackets is, currently, a shorthand notation for an array of type Any.

3.1.9 Accessing the contents of an array

To access the elements of an array, type the name of the array, followed by the element number in square brackets. Here's a 1D array:

```
julia> a = [10,20,30,40,50,60,70,80,90,100]
```

The fifth element:

```
julia> a[5]
50
```

The first element is index number 1. Julia is one of the languages that starts indexing elements in lists and arrays starting at 1, rather than 0. (And thus it's in the elite company of Matlab, Mathematica, Fortran, Lua, and Smalltalk, while most of the other programming languages are firmly in the opposite camp of 0-based indexers.)

Here's a 2D array:

```
julia> a2 = [1 2 3; 4 5 6; 7 8 9]
3x3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9
julia> a2[1]
1
```

If you just ask for one element of a 2D array, you'll receive the second element as if the array is unwound column by column, i.e. down first, then across:

```
julia> a2[2]
4
```

Asking for row and column works as you expect:

```
julia> a2[1,2]
2
```

which is row 1, column 2. Here's row 1, column 3:

```
julia> a2[1,3]
3
```

but don't get the row/column indices wrong:

```
julia> a2[1,4]
ERROR: BoundsError()
```

```
 in getindex at array.jl:247
```

The error message here is another reminder that indexing using square brackets is an alternative to calling the `getindex()` function:

```
julia> getindex(a2, 1,3)
3
julia> getindex(a2, 1,4)
ERROR: BoundsError()
 in getindex at array.jl:247
```

Rows and Columns

With a 2D array, you use brackets, colons, and commas to extract individual rows and columns or ranges of rows and columns:

```
julia> table = [r * c for r in 1:5, c in 1:5]
5x5 Array{Int64,2}:
 1   2   3   4   5
 2   4   6   8  10
 3   6   9  12  15
 4   8  12  16  20
 5  10  15  20  25
```

You can find a single row using the following (notice the comma):

```
julia> table[1,:]
1x5 Array{Int64,2}:
 1  2  3  4  5
```

And you can get a range of rows with a range followed by a comma and a colon:

```
julia> table[2:3,:]
2x5 Array{Int64,2}:
 2  4  6   8  10
 3  6  9  12  15
```

For columns, start with a colon followed by a comma:

```
julia> table[:,2]
5-element Array{Int64,1}:
  2
  4
  6
  8
 10
```

On its own, the colon accesses the entire array:

```
julia> table[:]
25-element Array{Int64,1}:
  1
  2
  3
  4
  5
  2
  4
  6
  8
```

```
 10
  3
  6
  9
 12
 15
  4
  8
 12
 16
 20
  5
 10
 15
 20
 25
```

To extract a range of columns:

```
julia> table[:,2:3]
5x2 Array{Int64,2}:
  2   3
  4   6
  6   9
  8  12
 10  15
```

3.1.10 Setting the contents of arrays

To set the contents of an array, specify the indices on the right hand side of an assignment expression:

```
julia> a = [1:10];
julia> a[9]= -9
-9
```

To check if the array has really changed:

```
julia> print(a)
[1,2,3,4,5,6,7,8,-9,10]
```

You can set a bunch of elements at the same time:

```
julia> a[3:6] = -5
-5
julia> print(a)
[1,2,-5,-5,-5,-5,7,8,-9,10]
```

And you can assign a sequence of elements to a sequence of values:

```
julia> a[3:9] = [9:-1:3]
7-element Array{Int64,1}:
 9
 8
 7
 6
 5
 4
 3
```

Notice here that, although Julia shows the 7 element slice as the return value, in fact the whole array has been modified:

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

You can set ranges to a single value:

```
julia> a[1:10] = -1
-1
julia> print(a)
[-1,-1,-1,-1,-1,-1,-1,-1,-1,-1]
```

By the way, there's a function call version that does the same job of setting array contents, `setindex!()` :

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

You can refer to the entire contents of an array using the colon separator without start and end index numbers, i.e. `[:]` . For example, after creating the array `a` :

```
julia> a = [1:10];
```

we can refer to the contents of this array `a` using `a[:]` :

```
julia> b = a[:]
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
julia> b[3:6]
4-element Array{Int64,1}:
 3
 4
 5
 6
```

3.1.11 Filling arrays

`fill!()` can fill arrays with a single value:

```
julia> fill!(a, 0);
julia> print(a)
[0,0,0,0,0,0,0,0,0,0]
```

3.1.12 Finding items in arrays

If you want to know whether an array contains an item, use the `in()` function, which can be called in two ways:

```
julia> a = [1:10];
julia> 3 in a
true
```

This can also be phrased as a function call:

```
julia> in(3, a)
true
```

There's a set of functions starting with **find** — such as `find()` , `findfirst()` , and `findnext()` — that you can use to get the index or indices of array cells that match a specific value, or pass a test. Each of these has two or more more forms.

Here's an array of primes (including 1, because it looks better):

```
julia> primes = [1,2,3,5,7,11,13,17,19,23];
```

To find the first occurrence of a number, and obtain its index, you can use the following method of `findfirst()` :

```
julia> findfirst(primes,13)
7
```

so the seventh cell of the array equals 13:

```
julia> primes[7]
13
```

There's another version of `findfirst()` that lets you use a function that tests each element and returns the index of the first one that passes the test. The two arguments are a function and the array.

```
julia> findfirst(x -> x == 13, primes)
7
```

The `find()` function returns an array of indices, pointing to every element where the function returns true when applied to the value:

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

julia> find(iseven, primes)
1-element Array{Int64,1}:
 2
```

Remember that these are arrays of indexes, not the actual cell values. The indexes can be used to extract the corresponding values using the standard square bracket syntax:

```
julia> primes[find(isodd,primes)]
9-element Array{Int64,1}:
  1
  3
  5
  7
 11
 13
 17
 19
 23
```

The `findfirst()` version returns a single number — the index of the matching cell:

```
julia> findfirst(iseven, primes)
2

julia> primes[findfirst(iseven, primes)]
2
```

The `findnext()` function is very similar to the `find()` and `findfirst()` functions, but accepts an additional number that tells the functions to start the search from somewhere in the middle of the array, rather than from the beginning. For example, if `find-first(primes,13)` finds the index of the first occurrence of the number 13 in the array, we can continue the search from there by using this value in `findnext()` :

```
julia> findnext(isodd, primes, 1 + findfirst(primes,13))
8

julia> primes[ans]
17
```

The `findin(A, B)` function returns the indices of the elements in array A where the elements of array B can be found:

```
julia> findin(primes, [11,5])
2-element Array{Int64,1}:
 4
 6

julia> primes[4]
5

julia> primes[6]
11
```

Notice the order in which the indices are returned.

3.1.13 Filtering

There's a set of related functions that let you work on an array's elements. `filter()` finds and keeps elements if they pass a test. Here, use the `isprime()` function (as a named function without parentheses, rather than a function call with parentheses) to filter (keep) everything in the array that's prime.

```
julia> filter(isprime, [1:100])
25-element Array{Int64,1}:
  2
  3
  5
  7
 11
 13
 17
 19
 23
 29
 31
 37
 41
 43
 47
 53
 59
 61
 67
 71
 73
 79
 83
 89
 97
```

Like many Julia functions, there's a version which changes the array. So `filter()` returns a modified copy of the original, but `filter!()` changes the array.

The `count()` function is like `filter()` , but just counts the number of elements that satisfy the condition:

```
julia> count(isprime, [1:100])
25
```

Also, the `any()` function just tells you whether any of the elements satisfy the condition:

```
julia> any(isprime, [1:100])
true
```

and the `all()` function tells you whether all of the elements satisfy the condition. Here, `all()` checks to see whether `filter()` did the job properly.

```
julia> all(isprime, filter(isprime, [1:100]))
true
```

3.1.14 Other tests

The following functions are useful for working on numeric arrays:

```
julia> maximum(a)
10
julia> minimum(a)
1
julia> extrema(a)
(1,10)
```

`findmax()` finds the maximum element and returns it and its index in a tuple:

```
julia> findmax(a)
(10,10)
```

You can calculate the sum of elements:

```
julia> sum(a)
55
```

and the product of elements:

```
julia> prod(a)
3628800
```

3.1.15 Joining arrays

`union()` builds the union of two arrays.

```
julia> odds = [1:2:10]
5-element Array{Int64,1}:
 1
 3
 5
 7
 9

julia> evens = [2:2:10]
5-element Array{Int64,1}:
  2
  4
  6
  8
 10

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

`intersect()` finds the intersection of two arrays:

```
julia> intersect([1:10], [5:15])
6-element Array{Int64,1}:
  5
  6
  7
  8
  9
 10
```

3.1.16 Finding out about an array

With our 2D array:

```
julia> a2 = [1 2 3; 4 5 6; 7 8 9]
3x3 Array{Int64,2}:
 1  2  3
 4  5  6
 7  8  9
```

we can find out more about it using the following functions:

- `ndims()`
- `size()`
- `length()`
- `countnz()`

`ndims()` returns the number of dimensions, ie 1 for a vector, 2 for a table, and so on:

```
julia> ndims(a2)
2
```

`size()` returns the row and column count of the array, in the form of a tuple:

```
julia> size(a2)
(3,3)
```

`length()` tells you how many elements the array contains:

```
julia> length(a2)
9
```

`countnz()` tells you how many non-zero elements there are:

```
julia> countnz(a2)
9
```

There are two related functions for converting between row/column numbers and array index numbers, `ind2sub()` and `sub2ind()` . Row 1, Column 1 is easy, of course - it's element 1, But Row 3, Column 7 is harder to work out. `ind2sub()` takes the total rows and columns, and a element index. For example, `ind2sub(size(a2), 5)` returns the row and column for the fifth element, in the form of a tuple:

```
julia> ind2sub(size(a2), 5)
(2,2)
```

With a loop, you could look at the row/column numbers of every element in an array:

```
julia> for i in 1:length(a2)
       println(ind2sub(size(a2), i), " ", a2[i])
       end
(1,1)     1
(2,1) 4
(3,1)     7
(1,2)     2
(2,2)     5
(3,2)     8
(1,3)     3
(2,3)     6
(3,3)     9
```

To go in the reverse direction, use sub2ind().

```
julia> a2[sub2ind((3,3), 2, 1)]
```

finds you the element at row 2, column 1, for an array with dimensions (3,3).

3.1.17 Modifying arrays

To add an item at the end of an array, use `push!()` :

```
julia> push!(a, 20)
11-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 20
```

As usual, the exclamation mark reminds you that this function will change the array.

To add an item at the front, use the oddly-named `unshift!()` :

```
julia> unshift!(a, 0)
12-element Array{Int64,1}:
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 20
```

To remove the last item:

```
julia> pop!(a)
20
```

and the first:

```
julia> shift!(a)
0
```

More aggressive modification of arrays (and similar data structures) can be made with functions such as `deleteat!()` and `splice!()` . `deleteat!()` deletes an element:

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

To insert an element into an array at a given index, use the `splice!()` function. For example, here's a list of numbers with an obvious omission:

```
julia> a = [1,2,3,5,6,7,8,9]
8-element Array{Int64,1}:
 1
 2
 3
 5
 6
 7
 8
 9
```

Use `splice!()` to insert a sequence at a specific index. Julia returns the values that were replaced. Let's insert the numbers 4:6 at index position 4, currently occupied by the number 5:

```
julia> splice!(a, 4:5, [4:6])
2-element Array{Int64,1}:
 5
 6
```

and you can check that the new values were inserted correctly:

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

Without a replacement, `splice!()` just removes elements and moves the rest of them along.

```
julia> a = [1:10]; splice!(a,5); a
9-element Array{Int64,1}:
  1
  2
  3
  4
  6
  7
  8
  9
 10
```

Here are even more array-modifying functions:

- `resize!()`
- `append!()`
- `prepend!()`
- `empty!(a)`
- `rot90(a)` to rotate an array 90 degrees clockwise:

```
julia> rotr90([1 2 3;4 5 6])
3x2 Array{Int64,2}:
 4  1
 5  2
 6  3
```

If you want to do something to an array, there's probably a function to do it, and sometimes with an exclamation mark to remind you of the potential consequences.

Passing arrays to functions

A function can't modify a variable passed to it as an argument, but it can change the contents of a container passed to it.

For example, here's a function that changes its argument to 5:

```
julia> function set_to_5(x)
           x = 5
       end
set_to_5 (generic function with 1 method)

julia> x = 3
3

julia> set_to_5(x)
5

julia> x
3
```

Although the `x` inside the function is changed, the `x` outside the function isn't. Variable names in functions are local to the function.

But you can modify the contents of a container, such as an array. This function uses the `[:]` syntax to access the **contents** of the container `x` , rather than change the value of the variable `x` :

```
julia> function fill_with_5(x)
           x[:] = 5
       end
fill_with_5 (generic function with 1 method)
```

```
julia> x = [1:10]
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10

julia> fill_with_5(x)
5

julia> x
10-element Array{Int64,1}:
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
```

If, instead of accessing the container variable's contents, you try to change the variable itself, it won't work. For example, this function definition creates an array of 5s in `temp` and then attempts to change the argument `x` to be `temp` .

```
julia> function fail_to_fill_with_5(x)
          temp = similar(x)
          for i in 1:length(x)
           temp[i] = 5
          end
          x = temp
       end
fail_to_fill_with_5 (generic function with 1 method)

julia> x = [1:10]
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10

julia> fail_to_fill_with_5(x)
10-element Array{Int64,1}:
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5

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

You can change elements of the array, but you can't change the variable so that it points to a different array. In other words, your function isn't allowed to change the binding between the argument and the array that was passed to it.

Julia's way of handling function arguments is described as "pass-by-sharing". An array isn't copied when you pass it to a function (that would be very inefficient for large arrays).

Elementwise and vectorized operations

Many Julia functions and operators are designed specifically to work with arrays. This means that you don't always have to work through each element of an array and process it individually.

A simple example is the use of the basic arithmetic operators. These can be used directly on an array if the other argument is a single value:

```
julia> a = [1:10];
julia> a * 2
10-element Array{Int64,1}:
  2
  4
  6
  8
 10
 12
 14
 16
 18
 20
```

and every element is multiplied by 2.

```
julia> a / 100
10-element Array{Float64,1}:
 0.01
 0.02
 0.03
 0.04
 0.05
 0.06
 0.07
 0.08
 0.09
 0.1
```

and every element is divided by 100.

These operations are described as operating **elementwise** .

For a particular group of operations in Julia, there is, for each operator, an elementwise version, which starts with a dot. These versions are the same as their non-dotted versions, and work on the arrays element by element. For example, the counterpart of the multiply function (*) has an elementwise version (.*). This lets you multiply arrays together element by element:

```
julia> n1 = [1:6]
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
julia> n2= [100:100:600]
6-element Array{Int64,1}:
 100
 200
 300
 400
 500
 600
julia> n1 .* n2
6-element Array{Int64,1}:
  100
  400
  900
 1600
 2500
 3600
```

and the first element of the result is what you get by multiplying the first elements of the two arrays.

As well as the arithmetic operators, some of the comparison operators also have elementwise versions. For example, instead of using == in a loop to compare two arrays, use **.==** . Here are two arrays of ten numbers, one sequential, the other disordered, and an elementwise comparison to see how many elements of array b happened to end up in the same place as array a:

```
julia> a = [1:10]; b=rand(1:10, 10); a .== b
10-element BitArray{1}:
  true
 false
  true
 false
 false
 false
 false
 false
 false
 false
```

Quite a few of the mathematics functions can also be used elementwise on an array. For example, here's `sind()` finding the sine of 24 angles from 0 to 360°:

```
julia> sind([0:15:360])
25-element Array{Any,1}:
  0.0
  0.258819
  0.5
  0.707107
  0.866025
  0.965926
  1.0
  0.965926
  0.866025
  0.707107
  0.5
  0.258819
  0.0
 -0.258819
 -0.5
 -0.707107
 -0.866025
 -0.965926
 -1.0
 -0.965926
 -0.866025
 -0.707107
 -0.5
 -0.258819
  0.0
```

Watch out for `max()` and `min()` . You might think that `max()` can be used on an array, like this, to find the largest element:

```
julia> r = rand(0:10, 10)
10-element Array{Int64,1}:
  3
  8
  4
  3
  2
  5
  7
  3
 10
 10
```

but no.

```
julia> max(r)
ERROR: `max` has no method matching max(::Array{Int64,1})
```

For this task you'll have to use the related function `maximum()` :

```
julia> maximum®
10
```

You use `max()` on two or more arrays to carry out an elementwise examination, returning another array containing the maximum values:

```
julia> r = rand(0:10, 10); s = rand(0:10, 10); t = rand(0:10,10);
julia> max(r, s, t)
10-element Array{Int64,1}:
  8
  9
  7
  5
  8
  9
  6
 10
 9
 9
```

`min()` and `minimum()` behave in the same way.

You can test each value of an array and change it in a single operation, using element-wise operators.

Here's an array of random integers from 0 to 10:

```
julia> a = rand(0:10,10, 10)
10x10 Array{Int64,2}:
 10   5   3   4  7   9  5   8  10   2
  6  10   3   4  6   1  2   2   5  10
  7   0   3   4  1  10  7   7   0   2
  4   9   5   2  4   2  1   6   1   9
  0   0   6   4  1   4  8  10   1   4
 10   4   0   5  1   0  4   4   9   2
  9   4  10   9  6   9  4   5   1   1
  1   9  10  10  1   9  3   2   3  10
  4   6   3   2  7   7  5   4   6   8
  3   8   0   7  1   0  1   9   7   5
```

Now you can test each value for being equal to 0, then set those elements to 11, like this:

```
julia> a[a .== 0] = 11;

julia> a
10x10 Array{Int64,2}:
 10   5   3   4  7   9  5   8  10   2
  6  10   3   4  6   1  2   2   5  10
  7  11   3   4  1  10  7   7  11   2
  4   9   5   2  4   2  1   6   1   9
 11  11   6   4  1   4  8  10   1   4
 10   4  11   5  1  11  4   4   9   2
  9   4  10   9  6   9  4   5   1   1
  1   9  10  10  1   9  3   2   3  10
  4   6   3   2  7   7  5   4   6   8
  3   8  11   7  1  11  1   9   7   5
```

3.2 Tuples

A tuple is an ordered sequence of elements, like an array. A tuple is represented by parentheses and commas, rather than the square brackets used by arrays. The important difference between arrays and tuples is that tuples are immutable. You can't change a tuple once you've got one.

Tuples are mostly good for small fixed-length collections — they're used everywhere in Julia, for example as for argument lists and for returning multiple values from functions.

Other than setting the contents, tuples work in much the same way as arrays.

```
julia> t = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
(1,2,3,4,5,6,7,8,9,10)
julia> t
(1,2,3,4,5,6,7,8,9,10)
julia> t[1:3]
(1,2,3)
```

But you can't do this:

```
julia> t[1] = 0
ERROR: `setindex!` has no method matching
 setindex!(::(Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64,Int64),
 ::Int64, ::Int64)
```

And, because you can't modify tuples, you can't use any of the functions like `push!()` that you use with arrays:

```
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> t = (1,2,3)
(1,2,3)
julia> push!(t,4)
ERROR: `push!` has no method matching push!(::(Int64,Int64,Int64), ::Int64)
```

3.3 Sorting arrays

Julia has a flexible `sort()` function that returns a sorted copy of an array, and a companion `sort!()` version that changes the array so that it's sorted.

You can usually use `sort()` without options and obtain the results you want:

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

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

If you need more than the default sort, use the `by` and `lt` keywords and provide your own functions for processing and comparing elements during the sort.

The `by` function processes each element before comparison and provides the 'key' for the sort. For example, suppose you wanted a case-insensitive sort of an array of characters (the default is case-sensitive, with uppercase letters coming before lowercase letters). This can be achieved by passing the `lowercase()` function to `by` :

```
julia>chars = collect("Mr. Jock, TV quiz PhD, bags few lynx.");
julia>sort(chars, by = lowercase) |> join
"       ,,..abcDefghiJklMnoPqrsTuVwxyz"
```

and most characters occur in ASCII-table order, except "D", "J", "M", "P", "T", and "V". Notice that the 'by' function supplies the sort key, but the original elements appear in the final result.

Anonymous functions[3] can be useful when sorting arrays. Here's a 10 rows by 2 columns array of tuples:

```
julia> table = collect(enumerate(rand(1:100, 10)))
10-element Array{(Int64,Int64),1}:
 (1,86)
 (2,25)
 (3,3)
 (4,97)
 (5,89)
 (6,58)
 (7,27)
 (8,93)
 (9,98)
 (10,12)
```

You can sort this by the second element of each tuple, not the first, by supplying an anonymous function to `by` that points to the second element of each, The anonymous function says, given an object `x` to sort, sort by the second element of `x` :

```
julia> sort(table, by= x -> x[2])
10-element Array{(Int64,Int64),1}:
 (3,3)
 (10,12)
 (2,25)
 (7,27)
 (6,58)
 (1,86)
 (5,89)
 (8,93)
 (4,97)
 (9,98)
```

By default, sorting uses the built-in `isless()` function when comparing elements. You can change this behaviour by passing a different function to the `lt` keyword. This function should compare two elements and return true if they're sorted. The sorting process compares pairs of elements repeatedly until every element of the array is in the right place.

3 Chapter 6.1.8 on page 86

For example, suppose you want to sort an array of words according to the number of vowels in each word; i.e. the more vowels a word has, the earlier in the sorted results it occurs. First we'll need a function that counts vowels:

```
vowelcount(string) = count(c -> (c in "aeiou"), lowercase(string))
```

Now you can pass an anonymous function to `sort()` that compares the vowel count of two elements and returns the element with a higher count in each case:

```
sentence = split("Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed
 do eiusmod tempor incididunt ut labore et dolore magna aliqua.");

sort(sentence, lt = (x,y) -> vowelcount(x) > vowelcount(y))
```

The result is that the word with the most vowels appears first:

```
19-element Array{SubString{ASCIIString},1}:
 "adipisicing"
 "consectetur"
 "eiusmod"
 "incididunt"
 "aliqua."
 "labore"
 "dolore"
 "Lorem"
 "ipsum"
 "dolor"
 "amet,"
 "elit,"
 "tempor"
 "magna"
 "sit"
 "sed"
 "do"
 "ut"
 "et"
```

The `sort()` function also lets you specify a reverse sort - after the `by` and `lt` functions (if used) have done their work, you can pass a true value to `rev` , to reverse the array.

To sort arrays with two or more dimensions, including matrices, you should use `sortrows()` and `sortcolumns()` .

Here's a 10 by 10 array.

```
julia> table = rand(1:4, 10, 10)
10x10 Array{Int64,2}:
 1  1  4  1  3  1  3  2  2  2
 2  3  2  3  3  4  3  4  1  4
 1  1  1  2  2  3  3  2  1  3
 2  2  3  1  4  2  1  1  3  2
 3  4  2  1  4  4  1  1  3  3
 4  1  2  1  4  3  4  1  4  1
 1  1  3  2  4  3  4  3  3  1
 2  2  3  3  1  4  3  4  3  3
 3  3  2  4  3  2  2  2  3  4
 4  4  3  1  2  3  3  3  3  3
```

By default, `sortrows()` just sorts the array by the first element in each row:

```
julia> sortrows(table)
10x10 Array{Int64,2}:
 1  1  4  1  3  1  3  2  2  2
 1  1  1  2  2  3  3  2  1  3
 1  1  3  2  4  3  4  3  3  1
 2  3  2  3  3  4  3  4  1  4
 2  2  3  1  4  2  1  1  3  2
 2  2  3  3  1  4  3  4  3  3
 3  4  2  1  4  4  1  1  3  3
 3  3  2  4  3  2  2  2  3  4
 4  1  2  1  4  3  4  1  4  1
 4  4  3  1  2  3  3  3  3  3
```

— notice that `1 1 4` comes before `1 1 1` . But, as with `sort()` , `sortrows()` lets you specify the keys, and you can provide a tuple of column indicators, such as (x[1], x[2], x[3]), which sorts the array first by the element in column 1, then by the element in column 2, then by column 3.

```
julia> sortrows(table, by = x -> (x[1], x[2], x[3]))
10x10 Array{Int64,2}:
 1  1  1  2  2  3  3  2  1  3
 1  1  3  2  4  3  4  3  3  1
 1  1  4  1  3  1  3  2  2  2
 2  2  3  1  4  2  1  1  3  2
 2  2  3  3  1  4  3  4  3  3
 2  3  2  3  3  4  3  4  1  4
 3  3  2  4  3  2  2  2  3  4
 3  4  2  1  4  4  1  1  3  3
 4  1  2  1  4  3  4  1  4  1
 4  4  3  1  2  3  3  3  3  3
```

The `sortcols()` function does a similar job, sorting by column rather than row.

4 Types

4.1 Types

This section, on types, and the next section, on functions and methods, should ideally be read at the same time, because the two topics are so closely connected.

4.1.1 Types of type

Data elements come in different shapes and sizes, which are called **types** .

Consider the following numeric values: a floating point number, a rational number, and an integer:

```
0.5  1/2  1
```

It's easy for us humans to add these numbers without much thought, but a computer won't be able to use a simple addition routine to add all three values, because the types are different. Code for adding rational numbers has to consider numerators and denominators, whereas code for adding integers won't. The computer will probably convert two of these values to be the same type as the third — typically the integer and the rational will first be converted to floating-point — then the three floating-point numbers will be added together.

This business of converting types obviously takes time. So, to write really fast code, you want to make sure that you don't make the computer waste time by continually converting values from one type to another. When Julia compiles your source code (which happens every time you evaluate a function for the first time), any type indications you've provided allow the compiler to produce more efficient executable code.

Another issue with converting types is that in some cases you'll be losing precision — converting a rational number to a floating-point number is likely to lose some precision.

The official word from the designers of Julia is that types are optional. In other words, if you don't want to worry about types (and if you don't mind your code running slower than it might), then you can ignore them. But you'll encounter them in error messages and the documentation, so you will eventually have to tackle them...

A compromise is to write your top-level code without worrying about types, but, when you want to speed up your code, find out the bottlenecks where your program spends the most time, and clean up the types in that area.

4.1.2 The type system

There's a lot to know about Julia's type system, so the official documentation is really the place to go. But here's a brief overview.

Type hierarchy

In Julia types are organized in an hierarchical way, and this hierarchy has a tree structure. At the tree's root, we have a special type called `Any` , and all other types are connected to it directly or indirectly. Informally, we can say that the type `Any` has children. Its children are called `Any` 's **subtypes** . Alternatively, we say that a child's **supertype** is `Any` .

An example of that is the type `Number` , a direct child of `Any` . To see what `Number` 's supertype is, we can use the `super()` function:

```
julia> super(Number)
 Any
```

But if we were curious, we could also try to find `Number` 's subtypes (Number's children, therefore Any's grandchildren). To do this, we can use the function `subtypes()` :

```
julia> subtypes(Number)
2-element Array{Any,1}:
 Complex{T<:Real}
 Real
```

Despite the syntactical junk, we can observe that we have two subtypes of `Number` : `Complex` and `Real` . If we knew maths, we could check that, for mathematicians, real and complex are both, indeed, numbers. Here is a general rule: Julia type's hierarchy reflect the real world's hierarchy.

As another example, if both `Cat` and `Lion` were Julia types, it would natural if their supertype were `Feline` . We would have:

```
julia> abstract Feline
julia> type Jaguar <: Feline end
julia> type Lion <: Feline end
julia> subtypes(Feline)
2-element Array{Any,1}:
 Jaguar
 Lion
```

Concrete and abstract types

Each object in Julia (informally, this mean everything you can put into a variable in Julia) has a type. But not all types can have a respective object (instances of that type). The only ones that can have instances are called **concrete types** . These types cannot have any subtypes. The types that can have subtypes (e.g. `Any` , `Number`) are called **abstract**

types . Therefore we cannot have a object of type `Number` , since it's an abstract type. In other words, only the leaves of the type tree are concrete types and can be instantiated.

If we can't create objects of abstract types, why are they useful? With them, we can write code that generalizes for any of its subtypes. For instance, suppose we write a function that expects a variable of the type `Number` :

```
#this function gets a number, and returns the same number plus one
function plus_one(n::Number)
 return n + 1
end
```

In this example, the function expects a variable `n` . The type of `n` must be subtype of `Number` (directly or indirectly) as indicated with the :: syntax (but don't worry about the syntax yet). What does this mean? No matter if `n` 's type is `Int` (Integer number) or `Float64` (floating-point number), the function `plus_one()` will work correctly. Furthermore, `plus_one()` will not work with any types that are not subtypes of `Number` (e.g. text strings, arrays).

We can divide concrete types into two categories: primitive (or basic), and complex (or composite). Primitive types are the building blocks, usually hardcoded into Julia's heart, whereas composite types group many other types to represent higher-level data structures.

You'll probably see the following primitive types:

- the basic integer and float types (signed and unsigned): `Int8` , `Uint8` , `Int16` , `Uint16` , `Int32` , `Uint32` , `Int64` , `Uint64` , `Int128` , `Uint128` , `Float16` , `Float32` , and `Float64`
- more advanced numeric types: `BigFloat` , `BigInt`
- Boolean and character types: `Bool` and `Char`
- Text string types: `String`

A simple example of a composite type is `Rational` , used to represent fractions. It is composed of two pieces, a numerator and a denominator, both integers (of type `Int`).

4.1.3 Investigating types

Julia provides two functions for navigating the type hierarchy: `subtypes()` and `super()` .

```
julia> subtypes(Integer)
5-element Array{Any,1}:
 BigInt
 Bool
 Char
 Signed
 Unsigned

julia> super(Float64)
FloatingPoint
```

The `sizeof()` function tells you how many bytes an item of this type occupies:

```
julia> sizeof(BigFloat)
 32

julia> sizeof(Char)
 4
```

If you want to know how big a number you can fit into a particular type, these two functions are useful:

```
julia> typemax(Int64)
 9223372036854775807

julia> typemin(Int32)
 -2147483648
```

There are over 340 types in the base Julia system. You can investigate the type hierarchy with the following (not very elegant) code:

```
level = 0
function showtypetree(subtype)
    global level
    subtypelist = filter(asubtype -> asubtype != Any, subtypes(subtype))
    if length(subtypelist) > 0
         println("\t" ^ level, subtype)
         level += 1
         map(showtypetree, subtypelist)
         level -= 1
    else
         println("\t" ^ level, subtype)
    end
end

showtypetree(Number)
```

It produces something like this for the different Number types:

```
      Number
        Complex{T<:Real}
        Real
            FloatingPoint
                BigFloat
                Float16
                Float32
                Float64
            Integer
                BigInt
                Bool
                Char
                Signed
                    Int128
                    Int16
                    Int32
                    Int64
                    Int8
                Unsigned
                    Uint128
                    Uint16
                    Uint32
                    Uint64
                    Uint8
            MathConst{sym}
            Rational{T<:Integer}
```

This shows, for example, the four main subtypes of `Real` number: `FloatingPoint` , `Integer` , `Rational` , and `MathConstant` (mathematical constant).

4.1.4 Specifying the type of variables

We've already seen that Julia likes to guess at the types of things you put in your code, if you don't specify them:

```
julia> [1:10]
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
julia> [1.0:10]
10-element Array{Float64,1}:
  1.0
  2.0
  3.0
  4.0
  5.0
  6.0
  7.0
  8.0
  9.0
 10.0
```

And we've also seen that you can specify the type for a new array:

```
julia> fill!(Array(String, 3), "Julia")
3-element Array{String,1}:
 "Julia"
 "Julia"
 "Julia"
```

For variables, you can specify the type that its value must have. For technical reasons, you can't do this at the top level, in the REPL — you can only do it inside a definition. The syntax uses the `::` syntax, which means "is of type". So:

```
function f(x::Int64)
```

means that the function `f` accepts an argument `x` which must be an Integer64.

Type stability

Here's an example of how the performance of Julia code is affected by the choice of type for variables. Can you spot the difference between these two functions:

```
julia> function t1(n)
          s  = 0
          for i in 1:n
              s += s/i
          end
       end
t1 (generic function with 1 method)

julia> function t2(n)
          s  = 0.0
          for i in 1:n
              s += s/i
          end
       end
t2 (generic function with 1 method)
```

These two function definitions are almost identical, except for the type of `s` at the start of each one. After running them both, the timing results are noteworthy:

```
julia> @time t1(10000000)
elapsed time: 0.907372058 seconds (479991904 bytes allocated, 32.97% gc time)

julia> @time t2(10000000)
elapsed time: 2.421e-6 seconds (80 bytes allocated)
```

The performance of `t1()` is significantly worse than that of `t2()` . The reason is that `s` is declared as an integer, but inside the loop it's assigned to hold the result of `s/i` , which is a floating-point value: it has to be converted from integer to floating-point to match. So the function isn't **type stable** — the Julia compiler is unable to make assumptions about its contents, so it can't produce pure integer code or pure floating-point code. As a result, the code it ends up producing isn't as fast as it could be. (Run the commands `@code_llvm t1(100)` and `@code_llvm t2(100)` to see how much more work you've made the Julia compiler do...)

4.2 Creating types

In Julia, it's very easy for the programmer create new types, benefiting from the same performance and language-wise integration that the native types (those made by Julia's creators) have.

Abstract Types

Suppose we want to create a abstract type. To do this, we use Julia's keyword `abstract` followed by the name of the type you want to create:

```
abstract MyAbstractType
```

By default, the type you create is a direct subtype of `Any` :

```
julia> super(MyAbstractType)
 Any
```

You can change this using the `<:` operator. If you want your new abstract type to be a subtype of `Number` , for example, you can declare:

```
abstract MyAbstractType2 <: Number
```

Now, we get:

```
julia> super(MyAbstractType2)
 Number
```

Notice that in the same Julia session (without exiting the REPL or ending the script) it's impossible to redefine a type. That's why we had to create a type called `MyAbstractType2` . Another workaround is to restart Julia workspace using the `workspace()` function. This will start fresh the REPL section, but you will lose all libraries you imported and variables you stored.

Concrete Types

You can create new concrete types. To do this, use the `type` keyword, which has the same syntax as declaring the supertype. Also, the new type may contain multiple fields, where the object stores values. As an example, let's define a concrete type that is a subtype of `MyAbstractType` :

```
type MyType <: MyAbstractType
   foo
   bar::Int
end
```

We just created a type called `MyType` , a subtype of `MyAbstractType` , with two fields: `foo` that can be of any type, and `bar` , that is of type `Int` .

How do we create an object of `MyType` ? By default, Julia creates a **constructor** , a function that returns an object of that type. The function has the same name of the type, and each argument of the function correspond to each field. In this example, we can create a new object by typing:

```
julia> x = MyType("Hello World!", 10)
 MyType("Hello World!", 10)
```

This creates a `MyType` object, assigning *"Hello World!"* to the `foo` field and 10 to the `bar` field. We can access `x` 's fields by using the **dot** notation:

```
julia> x.foo
 "Hello World!"

julia> x.bar
 10
```

Also, we can change the field's values of the object easily:

```
julia> x.foo = 3.0
 3.0

julia> x.foo
 3.0
```

Notice that, since we didn't specify `foo` 's type when we created the type definition, we can change its type at any time. This is different when we try to change the type of `x.bar` (which we specified as being an `Int` according to `MyType` 's definition):

```
julia> x.bar = "Hello World!"
 ERROR: `convert` has no method matching convert(::Type{Int64}, ::ASCIIString)
in convert at base.jl:13
```

The error message tells us that Julia couldn't change `x.bar` 's type. This ensures type-stable code, and can provide better performance when programming. As a performance tip, specifying field's types when possible is usually good practice.

The default constructor is used for simple cases, where you type something like **type-name(field1, field2)** to produce a new instance of the type. But sometimes you want to do more when you construct a new instance, such as checking the incoming values. For this you can use an inner constructor, a function inside the type definition. The next section shows a practical example.

4.2.1 Example: British Currency

Here's an example of creating a simple composite type that can handle the old-fashioned British currency. Before Britain saw the light and introduced a decimal currency, the monetary system used pounds, shillings, and pence, where a pound consisted of 20 shillings, and a shilling consisted of 12 pence. This was called the £sd or LSD system (Latin for Librae, Solidii, Denarii, because the system originated in the Roman empire).

To define a suitable type, start with the type declaration:

```
 type LSD
```

To contain a price in pounds, shillings, and pence, this new type should contain three fields, pounds, shillings, and pence:

```
   pounds::Int
   shillings::Int
   pence::Int
```

The important task is to create a **constructor function** . This has the same name as the type, and accepts three values as arguments. After a few checks for invalid values, the special `new()` function creates a new object with the passed-in values. Remember we're still inside the `type` definition — this is an *inner* constructor.

```
   function LSD(a,b,c)
    if a < 0 || b < 0
      error("no negative numbers")
    end
    if c > 12 || b > 20
      error("too many pence or shillings")
    end
    new(a, b, c)
   end
```

Now we can finish the type definition:

```
end
```

Here's the complete type definition again:

```
type LSD
   pounds::Int
   shillings::Int
   pence::Int

   function LSD(a,b,c)
    if a < 0 || b < 0
      error("no negative numbers")
    end
    if c > 12 || b > 20
      error("too many pence or shillings")
    end
    new(a, b, c)
   end
end
```

It's now possible to create new objects that store old-fashioned British prices:

```
julia>price1 = LSD(5,10,6)
LSD(5,10,6)

julia>price2 = LSD(1,6,8)
LSD(1,6,8)
```

And you can't create bad prices, because of the simple checks added to the constructor function:

```
julia> price = LSD(1,0,13)
ERROR: too many pence or shillings
 in LSD at none:11
```

If you inspect one of the price objects we've created:

```
julia> names(price1)
3-element Array{Symbol,1}:
 :pounds
 :shillings
 :pence
```

you can see the three field symbols, and these are storing the values:

```
julia> price1.pounds
5
julia> price1.shillings
10
julia> price1.pence
6
```

The next task is to make this new type behave in the same way as other Julia objects. For example, we can't add two prices:

```
julia> price1 + price2
ERROR: `+` has no method matching +(::LSD, ::LSD)
```

and the output looks poor:

```
LSD(5,10,6)
```

Julia has the addition function (+) with methods defined for many types of object. The following code adds another method that can handle two LSD objects:

```
function +(a::LSD, b::LSD)
  newpence = a.pence + b.pence
  newshillings = a.shillings + b.shillings
  newpounds = a.pounds + b.pounds
  subtotal = newpence + newshillings * 12 + newpounds * 240
  (pounds, balance) = divrem(subtotal, 240)
  (shillings, pence) = divrem(balance, 12)
  LSD(pounds, shillings, pence)
end
```

The first two lines and the last line are the key to adding new behavior to Julia. This definition adds a new method to the + function, one that accepts two LSD objects, adds them together, and produces a new LSD object containing the sum.

Now you can add two prices:

```
julia> price1 + price2
LSD(6,17,2)
```

which is indeed the result of adding LSD(5,10,6) and LSD(1,6,8).

The next problem to address is the unattractive presentation of LSD objects. This is fixed in exactly the same way, by adding a new method, but this time to the `show()` function, which belongs to the Base environment:

```
function Base.show(io::IO, money::LSD)
    print(io, "£$(money.pounds).$(money.shillings)s.$(money.pence)d")
end
```

Here, the `io` is the output channel currently used by all `show()` methods. We've added a simple print expression that displays the field values with appropriate punctuation and separators. The `show()` method is automatically called:

```
julia> price1 + price2
£6.17s.2d

julia> price1 + price2 + LSD(0,19,11) + LSD(19,19,6)
£27.16s.7d
```

You can add one or more aliases, which are alternative names for a particular type. Since `Price` is a better way of saying `LSD` , we'll create an alias:

```
julia> typealias Price LSD
LSD (constructor with 1 method)

julia> Price(1, 19, 11)
£1.19s.11d
```

So far, so good, but these LSD objects are still not yet fully developed. If you want to do subtraction, multiplication, and division, you have to define additional methods for these functions for handling LSDs. Subtraction is easy enough, just requiring some fiddling with shillings and pence, so we'll leave that for now, but what about multiplication? Multiplying a price by a number involves two types of object, one a Price/LSD object, the other - well, any positive real number should be possible:

```
function *(a::LSD, b::Real)
    if b < 0
        error("Cannot multiply by a negative number")
    end

    totalpence = b * (a.pence + a.shillings * 12 + a.pounds * 240)
    (pounds, balance) = divrem(totalpence, 240)
    (shillings, pence) = divrem(balance, 12)
    LSD(pounds, shillings, pence)
end
```

Like the + method, this new `*` method is defined specifically to multiply a price by a number. It works surprisingly well for a first attempt:

```
julia> price1 * 2
£11.1s.0d
julia> price1 * 3
£16.11s.6d
julia> price1 * 10
£55.5s.0d
julia> price1 * 1.5
£8.5s.9d
julia> price3 = Price(0,6,5)
£0.6s.5d
julia> price3 * 1//7
£0.0s.11d
```

However, some failures are to be expected,. We didn't allow for the really old-fashioned fractions of a penny: the halfpenny, and the farthing:

```
julia> price1 * 0.25
ERROR: InexactError()
```

(The answer should be £1.7s.7$\frac{1}{2}$d. Our LSD objects don't allow fractions of a penny.)

But there's another, more pressing, problem. At the moment you have to give the price followed by the multiplier; the other way round fails:

```
julia> 2 * price1
ERROR: `*` has no method matching *(::Int64, ::LSD)
```

This is because, although Julia can find a method that matches `(a::LSD, b::Number)` , it can't find it the other way round: `(a::Number, b::LSD)` . But adding it isn't too difficult:

```
function *(a::Number, b::LSD)
  b * a
end
```

which adds yet another method to the `*` function.

```
julia> price1 * 2
£11.1s.0d

julia> 2 * price1
£11.1s.0d

julia> for i in 1:10
           println(price1 * i)
       end
£5.10s.6d
£11.1s.0d
£16.11s.6d
£22.2s.0d
£27.12s.6d
£33.3s.0d
£38.13s.6d
£44.4s.0d
£49.14s.6d
£55.5s.0d
```

The prices are now looking like an old British shop from the 19th century, forsooth!

It's still possible to add and improve the type — it would depend on how you envisage yourself or others using it. For example, you might want to add division and modulo methods, and to act intelligently about negative monetary values.

5 Controlling the Flow

5.1 Control flow

Typically each line of a Julia program is evaluated in turn. There are various ways to control and modify the flow of evaluation. These correspond with the constructs used in other languages:

- **ternary** and **compound** expressions
- **Boolean** switching expressions
- **if elseif else end** - conditional evaluation
- **for end** - iterative evaluation
- **while end** - iterative conditional evaluation
- **try catch error throw** exception handling
- **do** blocks

5.1.1 Ternary expressions

Often you'll want to do job A (or call function A) if some condition is true, or job B (function B) if it isn't. The quickest way to write this is using the ternary operator ("?" and ":"):

```
julia> x = 1
1
julia> x > 3 ? "yes" : "no"
"no"
julia> x = 5
5
julia> x > 3 ? "yes" : "no"
"yes"
```

Here's another example:

```
julia> x = 0.3
0.3
julia> x < 0.5 ? sin(x) : cos(x)
0.29552020666133955
```

and Julia returned the value of `sin(x)` , because x was less than 0.5. `cos(x)` wasn't evaluated at all.

5.1.2 Boolean switching expressions

Boolean operators let you evaluate an expression if a condition is true. You can combine the condition and expression using `&&` or `||` . `&&` means "and", and `||` means "or". Since Julia evaluates expressions one by one, you can easily arrange for an expression to be evaluated if — or if not — a previous condition is true or false.

The following example uses a Julia function that returns true or false depending on whether the number is prime: `isprime(n)` .

With `&&` , both parts have to be true, so we can write this:

```
julia> n = 1000003
1000003
julia> isprime(n) && "It's a prime!"
"It's a prime!"
julia> n = 1000004
1000004
julia> isprime(n) && "It's a prime!"
false
```

If the first condition (primality) is true, the second expression is evaluated. If not, it isn't, and just the condition is returned.

With the `||` operator, on the other hand:

```
julia> n = 1000003
1000003
juli> isprime(n) || "It's not a prime!"
true
julia> n = 1000004
1000003
julia> isprime(n) || "It's not a prime!"
"It's not a prime!"
```

If the condition is true, there's no need to evaluate the second expression, since we already have the one truth value we need for "or". If it's false, the second one is evaluated, because it might turn out to be true.

5.1.3 If and Else

For a more general — and traditional — approach to conditional execution, you can use `if` , `elseif` , and `else` . If you're used to other languages, don't worry about white space, braces, indentation, brackets, semicolons, or anything like that, but remember to finish the conditional construction with end.

```
julia> name = "Brinkley"
"Brinkley"
julia> if name == "Jeeves"
           println("Very Good Jeeves")
       elseif name == "Brinkley"
           println("Thank you, Brinkley")
           println("and shut the door behind you")
       else
           println("Fine, just ignore me")
       end
Thank you, Brinkley
and shut the door behind you
```

The `elseif` and `else` parts are optional too:

```
julia> name = "Jeeves"
julia> if name == "Jeeves"
```

```
          println("Very Good Jeeves")
       end
Very Good Jeeves
```

Just don't forget the `end` !

Switch and case? You don't have to learn the syntax for a Julia switch or case statement, because there isn't such a thing.

5.1.4 For loops and iteration

Working through a list or a set of values or from a start value to a finish value are all examples of iteration, and the `for` ... `end` construction can let you iterate through a number of different types of object, including ranges, arrays, sets, dictionaries, and strings.

Here's the standard syntax for a simple iteration through a range of values:

```
julia> for i in 0:10:100
            println(i)
       end
0
10
20
30
40
50
60
70
80
90
100
```

The variable `i` takes the value of each element in the array (which is built from a range object) in turn — here stepping from 0 to 100 in steps of 10.

```
julia> for color in ["red", "green", "blue"] # an array
           print(color, " ")
       end
red green blue
```

```
julia> for letter in "julia" # a string
           print(letter, " ")
       end
j u l i a
```

```
julia> for element in (1, 2, 4, 8, 16, 32) # a tuple
           print(element, " ")
       end
1 2 4 8 16 32
```

```
julia> for element in {1, 2, "pardon", "my", sin} # one of those {} things
           println(typeof(element), " " , element, " ")
       end
Int64 1
Int64 2
ASCIIString pardon
ASCIIString my
Function sin
```

We haven't yet met sets and dictionaries, but iterating through them is exactly the same.

You can iterate through a 2D array, stepping "down" through column 1 from top to bottom, then through column 2, and so on:

```
julia> a = reshape(1:100, (10, 10))
10x10 Array{Int64,2}:
  1  11  21  31  41  51  61  71  81   91
  2  12  22  32  42  52  62  72  82   92
  3  13  23  33  43  53  63  73  83   93
  4  14  24  34  44  54  64  74  84   94
  5  15  25  35  45  55  65  75  85   95
  6  16  26  36  46  56  66  76  86   96
  7  17  27  37  47  57  67  77  87   97
  8  18  28  38  48  58  68  78  88   98
  9  19  29  39  49  59  69  79  89   99
 10  20  30  40  50  60  70  80  90  100

julia> for n in a
       print(n, " ")
       end
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57
 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100
```

You can use `=` instead of `in` .

Loop variables and scope

The variable that steps through each item — the 'loop variable' — can be either a variable that already exists, in which case it stays around after the loop finishes, or a variable that doesn't currently exist (in the current scope), in which case it disappears when the loop finishes.

Here's a comparison:

```
julia> v = 0
0
julia> for v in 1:5
       println(v)
       end
1
2
3
4
5
julia> v
5
```

`v` existed before the loop, and exists after the loop — its value was changed by the iteration process. However:

```
julia> w
ERROR: w not defined
julia> for w in 1:5
       println(w)
       end
1
2
3
4
```

```
5

julia> w
ERROR: w not defined
```

Here, `w` didn't exist before the loop, and doesn't exist after the loop.

The difference between the two is worth noting. The advantage of the first approach is that you can find out the final value of the loop variable after the loop's finished.

Fine tuning the loop: Continue

Sometimes on a particular iteration you might want to skip to the next value. You can use `continue` to skip the rest of the code inside the loop and start the loop again with the next value.

```
julia> for i in 1:10
           if i % 3 == 0
              continue
           end
           println(i) # this and subsequent lines are
                    # skipped if i is a multiple of 3
       end
1
2
4
5
7
8
10
```

Comprehensions

This oddly-named concept is simply a way of generating and collecting items. In mathematical circles you'd say something like:

```
"Let S be the set of all elements n where n is equal to or greater than 1 and
less than or equal to 10".
```

In Julia, you can write this as:

```
julia> S = [n for n in 1:10]
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
```

and this is called **array comprehension** or **list comprehension** . ('Comprehension' in the sense of 'getting everything' rather than 'understanding', perhaps?) The outer brackets collect together the elements generated by evaluating the expression placed before the `for` . Instead of `end` , use a square bracket to finish.

```
julia> [i^2 for i in 1:10]
10-element Array{Int64,1}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

Here's another example, using curly braces instead of braces to make an array of Any type:

```
julia> {i^2 for i in 1:10}
10-element Array{Any,1}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

Next, two iterators in a comprehension, separated with a comma, makes generating tables very easy. Here we're making a table of tuples:

```
julia> [(r,c) for r in 1:5, c in 1:5]
5x5 Array{(Int64,Int64),2}:
 (1,1)  (1,2)  (1,3)  (1,4)  (1,5)
 (2,1)  (2,2)  (2,3)  (2,4)  (2,5)
 (3,1)  (3,2)  (3,3)  (3,4)  (3,5)
 (4,1)  (4,2)  (4,3)  (4,4)  (4,5)
 (5,1)  (5,2)  (5,3)  (5,4)  (5,5)
```

`c` goes through five cycles, once for every value of `r` .

Enumerating arrays

Often you want to go through an array element by element while also keeping track of the index number of each element. The `enumerate()` function gives you an **iterable object** that produces both an index number and the value of an array at each index number:

```
julia> m = rand(0:9, 3, 3)
3x3 Array{Int64,2}:
 9  2  9
 3  4  3
 6  1  1

julia> [i for i in enumerate(m)]
9-element Array{(Int64,Any),1}:
 (1,9)
 (2,3)
 (3,6)
 (4,2)
 (5,4)
 (6,1)
 (7,9)
 (8,3)
 (9,1)
```

Zipping arrays

Sometimes you want to work through two or more arrays at the same time, taking the first element of each array first, then the second, and so on. This is possible using the `zip()` function:

```
julia> for i in zip(0:10, 100:110, 200:210)
           println(i)
          end
(0,100,200)
(1,101,201)
(2,102,202)
(3,103,203)
(4,104,204)
(5,105,205)
(6,106,206)
(7,107,207)
(8,108,208)
(9,109,209)
(10,110,210)
```

You'd think it would all go wrong if the arrays were different sizes. What if the third array is too big?

```
julia> for i in zip(0:10, 100:110, 200:215)
        println(i)
    end
(0,100,200)
(1,101,201)
(2,102,202)
(3,103,203)
(4,104,204)
(5,105,205)
(6,106,206)
(7,107,207)
(8,108,208)
(9,109,209)
(10,110,210)
```

but Julia isn't fooled — any oversupply in any one of the arrays is handled gracefully.

```
julia> for i in zip(0:15, 100:110, 200:210)
        println(i)
    end
(0,100,200)
(1,101,201)
(2,102,202)
(3,103,203)
(4,104,204)
(5,105,205)
(6,106,206)
(7,107,207)
```

```
(8,108,208)
(9,109,209)
(10,110,210)
```

Iterable objects

The ”for something in something” construction is the same for every other type of object that you can iterate through: arrays, dictionaries, sets, ranges, and so on. In Julia this is a general principle: there are a number of ways in which you can create an ”iterable object”, an object that is designed to be used as part of the iteration process that provides the elements one at a time.

The most obvious example we’ve already met is the range object. It doesn’t look much when you type it into the REPL:

```
julia> 0:2:100
0:2:100
```

But it yields the numbers on demand, when you start iterating:

```
julia> [i for i in 0:2:100]
51-element Array{Int64,1}:
   0
   2
   4
   6
   8
  10
  12
  14
  16
  18
  20
  22
  24
  26
  28
   ⋮
  74
  76
  78
  80
  82
  84
  86
  88
  90
  92
  94
  96
  98
 100
```

Should you want the actual numbers from a range (or other iterable object), you can use `collect()` :

```
julia> collect(0:25:100)
5-element Array{Int64,1}:
   0
  25
  50
  75
 100
```

This comes in useful when you have iterable objects created by other Julia functions. For example, `permutations()` creates an iterable object containing all the permutations of an array. You could use `collect()` to grab them and make a new array.

```
julia> collect(permutations([1:4]))
24-element Array{Array{Int64,1},1}:
 [1,2,3,4]
 [1,2,4,3]
 …
 [4,3,2,1]
```

Of course, there's a good reason why iterator objects don't produce all the values from the iteration at the same time: memory and performance. A range object doesn't take up much room, even if iterating over it might take ages (depending on how big the range is). If you generate all the numbers at once, rather than only producing them when they're needed, they all have to be stored somewhere...

Julia provides iterable objects for working with other types of data. For example, when you're working with files, you can treat an open file as an iterable object:

```
for line in eachline(filehandle)
    println(length(line), line)
end
```

Making your own iterable objects

It's possible to design your own iterable objects. What you do is create a new type that has `start()` , `next()` , and `done()` methods. Then, a `for` .. `end` loop can work through the components of your object by calling the methods automatically as necessary.

This example shows how you can create an iterable object that iterates through prime numbers, starting at 1.

First, you define a type:

```
type PrimeIterator
    n::Integer
end
```

You will later create an object of this type using the code:

```
primes = PrimeIterator(20)
```

The number you supply is passed to the type constructor, and stored in `n` . The iterator will be expected to generate every prime up to this number.

Next, you must define three methods that the iteration process will call. First, you want to add a `start()` method for starting the iteration process off. This function already exists in Julia (that's why you can iterate over all the basic data objects), so the `Base` prefix is required, because you're adding a new method to the system's `start()` function, one which is designed to handle these special objects.

```
function Base.start(::PrimeIterator)
  1
end
```

The `next()` function returns a tuple of two values. The first item in the tuple returned, whether returned by the first part of the `if` clause or the `else` clause, is the next value of the iterator. The last item is the state that will be preserved and passed on to the next invocation of `next()` . In this particular case, the code just runs up to find the next prime number above `state` :

```
function Base.next(::PrimeIterator,state)
    if state == 1
        return (1, 2)          # first return possibility
    else
        while ! isprime(state)
            state += 1
        end
        (state, state += 1)  # second return possibility
    end
end
```

The `done()` function is required so that the iteration knows when to stop. This is simple, because the construction of the PrimeIterator object expected a number, which acts as the obvious stop point. This function is the only one which refers to the original number which was passed to the constructor. This number was called `n` in the type definition, so it can be accessed using the "." field accessor.

```
function Base.done(piter::PrimeIterator, state)
    state >= piter.n
end
```

So, to run through the primes from 1 to 20:

```
primes = PrimeIterator(20)
for i in primes
           println(i)
end
1
2
3
5
7
11
13
17
19
```

(Code inspired by Carl Vogel posts at `http://slendermeans.org`.)

Nested loops

If you want to nest one loop inside another, you don't have to duplicate the `for` and `end` keywords. Just use a comma:

```
julia> for x in 1:10, y in 1:10
   println("x is $x, y is $y")
end

x is 1, y is 1
x is 1, y is 2
x is 1, y is 3
```

```
x is 1, y is 4
x is 1, y is 5
x is 1, y is 6
x is 1, y is 7
x is 1, y is 8
x is 1, y is 9
x is 1, y is 10
x is 2, y is 1
...
x is 10, y is 1
x is 10, y is 2
x is 10, y is 3
x is 10, y is 4
x is 10, y is 5
x is 10, y is 6
x is 10, y is 7
x is 10, y is 8
x is 10, y is 9
x is 10, y is 10
```

One difference between the shorter and longer forms of nesting loops is the behaviour of `break`:

```
julia> for x in 1:10
   for y in 1:10
       println("x is $x, y is $y")
       if y % 3 == 0
          break
       end
   end
end

x is 1, y is 1
x is 1, y is 2
x is 1, y is 3
x is 2, y is 1
x is 2, y is 2
x is 2, y is 3
x is 3, y is 1
x is 3, y is 2
x is 3, y is 3
...
x is 9, y is 3
x is 10, y is 1
x is 10, y is 2
x is 10, y is 3

julia> for x in 1:10, y in 1:10
 println("x is $x, y is $y")
 if y % 3 == 0
    break
  end
end

x is 1, y is 1
x is 1, y is 2
x is 1, y is 3

julia>
```

Notice that `break` breaks out of both inner and outer loops in the shorter form, but only out of the inner loop in the longer form.

5.1.5 While loops

To repeat some expressions while a condition is true, use the `while` ... `end` construction.

```
julia> x = 0
0
julia> while x < 4
    println(x)
    x += 1
end

0
1
2
3
```

If you want the condition to be tested after the statements, rather than before, producing a ”do .. until” form, use the following construction:

```
while true
   println(x)
   x += 1
   x >= 4 && break
end

0
1
2
3
```

Here we’re using a Boolean switch rather than an `if` ... `end` statement.

5.1.6 Exceptions

If you want to write code that checks for errors and handles them gracefully, use the `try` ... `catch` construction:

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

5.1.7 Do block

Finally, let’s look at a `do` block, which is another syntax form that, like the list comprehension, looks at first sight to be a bit backwards (i.e. it can perhaps be better understood by starting at the end and working towards the beginning).

A `do` block is a way of passing an anonymous function[1] and its argument (between the `do` and `end` keywords) to a function that lets you use anonymous functions and apply them to arguments.

1 Chapter 6.1.8 on page 86

Remember the `find()` example from earlier[2]?

```
julia> primes
10-element Array{Int64,1}:
  1
  2
  3
  5
  7
 11
 13
 17
 19
 23
julia> find(x -> x == 13, primes)
1-element Array{Int64,1}:
 7
```

but the anonymous function `find(x -> x == 13, primes)` can be written as a `do` block:

```
julia> find(primes) do x
       x == 13
       end
1-element Array{Int64,1}:
 7
```

You just lose the arrow and change the order, putting the`find()` function and its array argument first, then adding the anonymous function on one or more lines at the end, after the `do` (keep the anonymous function's arguments on the same line as the `do`), instead of squeezing everything in as the first argument of `find()` .

The idea is that it's easier to write a longer anonymous function on multiple lines at the end of the form, rather than wedged in as the first argument.

2 Chapter 3.1.12 on page 36

6 Functions

6.1 Functions

Functions are the building blocks of Julia code, acting as the subroutines, procedures, blocks, and similar structural concepts found in other programming languages.

A function's job is to take a tuple of values as an argument list and return a value. If the arguments contain mutable values like arrays, the array can be modified inside the function. By convention, an exclamation mark (!) at the end of a function's name indicates that the function may modify its arguments.

There are various syntaxes for defining functions:

- when the function contains a single expression
- when the function contain multiple expressions
- when the function doesn't need a name

6.1.1 Single expression functions

To define a simple function, all you need to do is provide the function name and argument on the left and an expression on the right of an equals sign. These are like mathematical functions:

```
julia> f(x) = x * x
f (generic function with 1 method)

julia> f(2)
4


julia> g(x, y) = sqrt(x^2 + y^2)
g (generic function with 1 method)

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

6.1.2 Functions with multiple expressions

The syntax for defining a function with more than one expression is like this:

```
function functionname(arg1, arg2)
  an expression
  another expression
  more expressions
  the final expression
end
```

Here's a typical function that calls two other functions and then ends.

```
julia> function breakfast()
   maketoast()
   brewcoffee()
end

breakfast (generic function with 1 method)
```

Whatever the value returned by the final expression — here, the `brewcoffee()` function — that value is also returned by the `breakfast()` function.

You can use the `return` keyword to indicate a specific value to be returned:

```
julia> function payBills(bankBalance)
   if bankBalance < 0
       return false
   else
      return true
   end
end
payBills (generic function with 1 method)
julia> payBills(20)
true
julia> payBills(-10)
false
```

Later we'll learn how to make sure that the function doesn't go adrift if you call it with the wrong type of argument.

Returning more than one value from a function

To return more than one value from a function, use a tuple.

```
julia> function doublesix()
           (6,6)
       end
doublesix (generic function with 1 method)
julia> doublesix()
(6,6)
```

6.1.3 Optional arguments and variable number of arguments

You can define functions with optional arguments, so that the function can use sensible defaults if specific values aren't supplied. You provide a default symbol and value in the argument list:

```
julia> function xyzpos(x, y, z=0)
          println("$x, $y, $z")
       end
xyzpos (generic function with 2 methods)
```

And when you call this function, if you don't provide a third value, the variable `z` defaults to 0 and uses that value inside the function.

```
julia> xyzpos(1,2)
1, 2, 0
```

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

6.1.4 Keyword and positional arguments

The problem with writing functions like this:

```
function f(p, q, r, s, t, u)
...
end
```

is that, sooner or later, you'll forget the order in which you have to supply the arguments. Was it:

```
 f("42", -2.123, atan2, "obliquity", 42, 'x')
```

or

```
 f(-2.123, 42, 'x', "42", "obliquity", atan2)
```

You can avoid this problem by using keywords to label arguments. Use a semicolon after the function's unlabelled arguments, and follow it with one or more `keyword=value` pairs:

```
julia> function f(p, q ; r = 4, s = "hello")
  println("p is $p")
  println("q is $q")
  return ["r"=>r,"s"=>s]
end
f (generic function with 1 method)
```

When called, this function expects two arguments, and will also accept a number and a string, labelled `r` and `s` . If you don't supply them, the default values are used:

```
julia> f(1,2)
p is 1
q is 2
Dict{ASCIIString,Any} with 2 entries:
  "r" => 4
  "s" => "hello"

julia> f("a", "b", r=pi, s=22//7)
p is a
q is b
Dict{ASCIIString,Any} with 2 entries:
  "r" => π = 3.1415926535897...
  "s" => 22//7
```

If you supply a keyword argument, it can be anywhere in the argument list, not just at the end or in the matching place. And you don't use the semicolon when calling it.

```
julia> f(r=999, 1, 2)
p is 1
q is 2
Dict{ASCIIString,Any} with 2 entries:
  "r" => 999
  "s" => "hello"
julia> f(s="hello world", r=999, 1, 2)
p is 1
q is 2
Dict{ASCIIString,Any} with 2 entries:
  "r" => 999
  "s" => "hello world"
julia>
```

When defining the function, remember to insert a semicolon before the keyword/value pairs.

Here's another example, from the Julia manual. The `rtol` keyword can appear anywhere in the list of arguments (and it can be omitted altogether):

```
julia> isapprox(3.0, 3.01, rtol=0.1)
true
julia> isapprox(rtol=0.1, 3.0, 3.01)
true
julia> isapprox(3.0, 3.00001)
true
```

A function definition can combine all the different kinds of arguments. Here's one with normal, optional, and keyword arguments:

```
julia> function f(a1, opta2=2; key="foo")
   println("normal argument: $a1")
   println("optional argument: $opta2")
   println("keyword argument: $key")
end
f (generic function with 2 methods)

julia> f(1)
normal argument: 1
optional argument: 2
keyword argument: foo

julia> f(key=3, 1)
normal argument: 1
optional argument: 2
keyword argument: 3

julia> f(key=3, 2, 1)
normal argument: 2
optional argument: 1
keyword argument: 3
```

6.1.5 Functions with variable number of arguments

Functions can be defined so that they can accept any number of arguments:

```
function fvar(args...)
    println("you supplied $(length(args)) arguments")
    for arg in args
       println(" argument ", arg)
    end
end
```

The three dots indicate the famous (or infamous) **splat** . Here it means 'any', including 'none'.

You can call this function with any number of arguments:

```
julia> fvar()
you supplied 0 arguments
julia> fvar(64)
you supplied 1 arguments
```

```
 argument 64
julia> fvar(64,65)
you supplied 2 arguments
 argument 64
 argument 65
julia> fvar(64,65,66)
you supplied 3 arguments
 argument 64
 argument 65
 argument 66
```

and so on. The splat could also be called 'ellipsis'.

Here's another example. Suppose you define a function that accepts two arguments:

```
julia> function test(x, y)
   println("x $x y $y")
end
```

You can call this in the usual way:

```
julia> test(12, 34)
x 12 y 34
```

So what do you do if you have the two numbers, but in a tuple. How do you supply a single tuple of numbers to this two argument function? The answer is to use the ellipsis — splat.

```
julia> test((12, 34) ...)
x 12 y 34
```

6.1.6 Local variables and changing the values of arguments

Any variable you define inside a function will be forgotten when the function finishes.

```
julia>function test(a,b,c)
    subtotal = a + b +c
end

test (generic function with 1 method)
julia> test(1,2,3)
6
julia> subtotal
ERROR: subtotal not defined
```

If you want to keep values around across function calls, then you can think about using global variables[1].

A function can't modify an existing variable passed to it as an argument, but it can change the contents of a container passed to it.

For example, here's a function that changes its argument to 5:

```
julia> function set_to_5(x)
         x = 5
       end
```

1 http://en.wikibooks.org/wiki/Introducing_Julia%2FModules_and_packages%23Global_variables_and_scope%20

```
set_to_5 (generic function with 1 method)

julia> x = 3
3

julia> set_to_5(x)
5

julia> x
3
```

Although the `x` inside the function is changed, the `x` outside the function isn't. Variable names in functions are local to the function.

But you can modify the contents of a container, such as an array. This function uses the `[:]` syntax to access the **contents** of the container `x` , rather than change the value of the variable `x` :

```
julia> function fill_with_5(x)
           x[:] = 5
       end
fill_with_5 (generic function with 1 method)

julia> x = [1:10];
julia> fill_with_5(x)
5

julia> x
10-element Array{Int64,1}:
 5
 5
 5
 5
 5
 5
 5
 5
 5
 5
```

You can change elements of the array, but you can't change the variable so that it points to a different array. In other words, your function isn't allowed to change the binding of the argument.

6.1.7 Map and apply

If you already have a function and an array, you can call the function for each element of the array by using `map()` . This calls the function on each element in turn and returns the result in another array. This process is called "mapping":

```
julia> map(sin, a)
10-element Array{Float64,1}:
  0.841471
  0.909297
  0.14112
 -0.756802
 -0.958924
 -0.279415
  0.656987
  0.989358
  0.412118
 -0.544021
```

This returns a copy of the array `a` , and, as usual, there's a corresponding function `map!` that modifies the contents of the array.

Often, you don't have to use `map()` to apply a function like `sin()` to every member of an array, because many functions automatically operate "element-wise". In fact, if you compare the timings of the `map()` version to the element-wise approach, you'll see that `map()` isn't as quick:

```
julia> @time map(sin,[1:10000]);
elapsed time: 0.023442818 seconds (472120 bytes allocated, 97.43% gc time)

julia> @time sin([1:10000]);
elapsed time: 0.000293766 seconds (160200 bytes allocated)
```

There's an `apply()` function, which takes a list of arguments and a function, and calls the function with those arguments. Consider this simple function, `f` , which takes two arguments:

```
julia> f(x,y) = x^2 + y^2
f (generic function with 1 method)

julia> f(3,4)
25
```

Now suppose you have the two values already bound together in an array or tuple, e.g. stored as `[3,4]` or `(3,4)` . It looks at first like you can't apply the function to the arguments:

`julia> f([3,4]) ERROR: `f` has no method matching f(::Array{Int64,1})`

But with `apply()` you apply the function to the values like this:

```
julia> apply(f,[3,4])
25
```

Another way of doing this is to use the ellipsis or splat:

```
julia> f([3,4]...)
25
```

which is sometimes referred to as 'splicing' the arguments.

Map with multiple arrays

You can use `map()` with more than one array. The function is applied to the first element of each of the arrays, then to the second, and so on. The arrays must be of the same length (unlike the `zip()` function, which is more tolerant).

Here's an example which generates an array of imperial (non-metric) spanner/socket sizes. The second array is just a bunch of repeated 32s to match the integers from 5 to 24 in the first array. Julia simplifies the rationals for us:

```
julia> map(//, [5:24], fill(32,20))
20-element Array{Rational{Int64},1}:
  5//32
  3//16
  7//32
  1//4
  9//32
  5//16
 11//32
  3//8
 13//32
  7//16
 15//32
  1//2
 17//32
  9//16
 19//32
  5//8
 21//32
 11//16
 23//32
  3//4
```

(In reality, an imperial spanner set won't contain some of these strange sizes - I've never seen an old 17/32" spanner.)

6.1.8 Anonymous functions

Sometimes you don't want to worry about thinking up a cool name for a function. Anonymous functions — functions with no name — can be used in a number of places in Julia, such as with `map()` , and in list comprehensions.

The syntax uses `->` , like this:

```
x -> x^2 + 2x - 1
```

which defines a nameless function that takes a argument, calls it `x` , and produces `x^2 + 2x - 1` .

For the `map()` function, for example, the first argument is a function, and you can define a one-off function that exists just for this particular `map()` operation:

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

After the `map()` finishes, the function, and the argument `x` has disappeared:

```
julia> x
ERROR: x not defined
```

If you want an anonymous function that accepts more than one argument, provide the arguments as a tuple:

```
julia> map((x,y,z) -> x + y + z, [1,2,3], [4, 5, 6], [7, 8, 9])
3-element Array{Int64,1}:
 12
 15
 18
```

Notice that the results are 12, 15, 18, rather than 6, 15, and 24. The anonymous function takes the first value of each of the three arrays and adds them, followed by the second, then the third.

Also, anonymous functions can have zero arguments, if you use an 'empty' tuple`()` :

```
julia> random = () -> rand(0:10)
(anonymous function)
julia> random()
3
julia> random()
1
```

6.1.9 Methods

A function can have one or more different methods for doing the same job. Each method concentrates on doing the job for a particular type of argument.

When you enter this code in the REPL:

```
julia> function firstpaybills(bankBalance)
       if bankBalance < 0
           return false
       else
          return true
       end
       end
firstpaybills (generic function with 1 method)
```

the message ("generic function with 1 method") tells you that there is currently one way you can call the `firstpaybills()` function. If you call this function and supply a number, it works fine. But what happens when you try to pass a string as the argument, such as a string "now" (or "pay it now"):

```
julia> firstpaybills("now")
ERROR: `isless` has no method matching isless(::ASCIIString, ::Int64)
 in firstpaybills at none:2
```

The error tells us that the function can't carry on, because the concept of less than (<), which we're using inside our function, makes no sense if the first argument is a string and the second argument is an integer. Strings aren't really less than or greater than integers, they're two separate things, so the function fails at that point.

Notice that the `firstpaybills()` function did start executing, though. The argument `bankBalance` could have been anything - a string, a floating point number, an integer, symbol, or even an array. There are many ways for this function to fail. This is not the best way to write code!

Let's start again. The first improvement is to specify that the `bankBalance` argument must be some kind of number:

```
function paybills(bankBalance::Number)
   if bankBalance < 0
      return false
   else
      return true
   end
```

```
end
paybills (generic function with 1 method)
```

Now the function doesn't run if you try to call it with arguments that aren't numbers, and the problem of calling `<` on non-numeric values is avoided. If you use a type such as Number, you can call this particular method with any argument, provided that it's a kind of number. We can use the `applicable()` function to test it. `applicable()` lets you know whether you can apply a function to an argument — i.e. whether there's an available method for the function for arguments with that type:

```
julia> applicable(paybills, 3)
true

julia> applicable(paybills, 3.14)
true

julia> applicable(paybills, pi)
true

julia> applicable(paybills, 22//7)
true

julia> applicable(paybills, -5)
true

julia> applicable(paybills, "now")
false
```

and the `false` indicates that you can't pass a string value to the `paybills()` function because there's isn't a method that accepts strings:

```
julia> paybills("now")
ERROR: `paybills` has no method matching paybills(::ASCIIString)
```

Now the body of the function isn't even looked at — Julia doesn't know a method for calling `paybills()` function with a string argument, only a method that accepts numeric arguments.

The next step perhaps is to define another method for the `paybills()` function, only this time one that accepts a string argument. In this way, functions can be given a number of separate methods: one for numeric arguments, one for string arguments, and so on. Julia selects and runs one of the available methods depending on what types you're dealing with. This is the idea of **multiple dispatch** .

```
julia> function paybills(action::String)
          if action == "now"
             return "make sure you have enough money in the bank"
          elseif action == "later"
             return "OK, worry it about them tomorrow"
          end
       end
paybills (generic function with 2 methods)
```

Now the `paybills()` function has two methods. The code that runs depends on the type of the argument you provide to the function.

You can use the built-in `methods()` function to find out how many methods you've defined for a particular function. A good example is to see how many different ways you can use the `+` function in Julia:

```
julia> methods(+)
```

which prints a list of every method for the `+` function. There are over 100 different types of thing that you can add together, including arrays and matrices.

Just one more observation about our first `paybills()` method. We specified the type Number, but the Number type includes Complex numbers:

```
julia> subtypes(Number)
2-element Array{Any,1}:
 Complex{T<:Real}
 Real
```

But our `paybills( )` method will fail if it's given a complex number as a bank balance value (as might be expected):

```
julia> paybills(complex(1))
ERROR: `isless` has no method matching isless(::Complex{Int64}, ::Int64)
 in paybills at none:2
```

Perhaps we should have defined our method to accept values of type Real rather than of type Number. Then, the method wouldn't be called at all.

If you decide to add a method to handle Real arguments in the REPL, you'll end up with **three** methods:

```
julia> methods(paybills)
# 3 methods for generic function "paybills":
paybills(bankBalance::Real) at none:2
paybills(bankBalance::Number) at none:2
paybills(action::String) at none:2
```

The obvious question to ask is, which method is called for numbers that aren't complex, since there are apparently two methods which would apply to most numbers: 1 is both Real and Number.

The answer, according to the official documentation[2], is that Julia chooses "the most specific method". Here, because Real is a subtype of Number, the Real method is chosen whenever possible. Only if you choose Complex does Julia choose the Number version.

6.1.10 Functions that return functions

You can treat Julia functions in the same way as any other Julia object, particularly when it comes to returning them as the result of other functions.

For example, let's create a function-making function. Inside this function, a function called `newfunction` is created, and this will raise its argument (y) to the number that was originally passed in as the argument x. This new function is returned as the value of the `create_exponent_function()` function.

2 Chapter 0.1 on page 1

```
function create_exponent_function(x)
    newfunction = function (y) return y^x end
    return newfunction
end
create_exponent_function (generic function with 1 method)
```

Now we can construct lots of exponent functions. First, let's build a `squarer()` function:

```
julia> squarer = create_exponent_function(2)
(anonymous function)
```

and a `cuber()` function:

```
julia> cuber = create_exponent_function(3)
(anonymous function)
```

While we're at it, let's do a "raise to the power of 4" function (called `quader` , although I'm starting to struggle with the Latin and Greek naming):

```
julia> quader = create_exponent_function(4)
(anonymous function)
```

These are ordinary Julia functions:

```
julia> squarer(4)
16
julia> cuber(5)
125
julia> quader(6)
1296
```

The definition of the `create_exponent_function()` above is perfectly valid Julia code, but it's not idiomatic. For one thing, the return value doesn't always need to be provided explicitly — the final evaluation is returned if `return` isn't used. Also, in this case, the full form of the function definition can be replaced with the shorter one-line version. This gives the concise version:

```
function create_exponent_function(x)
   y -> y^x
end
create_exponent_function (generic function with 1 method)
```

which acts in the same way.

Here's another example of making functions:

```
make_counter = function()
     count = 0
     function()
       so_far = count
       count += 1
       so_far
     end
end
(anonymous function)

julia> a = make_counter()
(anonymous function)
julia> b = make_counter()
(anonymous function)
julia> a()
```

```
0
julia> a()
1
julia> a()
2
```

6.2 Type parameters in method definitions

It's possible to include type information in method definitions. You can provide one or more variables and types, in curly braces, preceding the tuple of arguments. Here's a simple example:

```
julia> function test{T <: Real}(a::T)
           println("$a is a $T")
       end
test (generic function with 3 methods)

julia> test(2.3)
2.3 is a Float64

julia> test(2)
2 is a Int64

julia> test(.02)
0.02 is a Float64

julia> test(pi)
π = 3.1415926535897... is a MathConst{:π}

julia> test(22//7)
22//7 is a Rational{Int64}
```

The `test()` method automatically extracts the type of the single argument `a` passed to it and stores it in the 'variable' `T` . For this function, `T` must be a subtype of the Real type (so it can be any real number, but not a complex number). 'T' can be used like any other variable — in this method it's just printed out using string interpolation.

This mechanism is useful when you want to constrain the arguments of a particular method definition to be of a particular type. For example, the type of argument `a` must belong to the Real number super type, so this `test()` method doesn't apply when `a` is an array or a string, because then the type of the argument isn't a subtype of Real:

```
julia> test("str")
ERROR: `test` has no method matching test(::ASCIIString)

julia> test([1:3])
ERROR: `test` has no method matching test(::Array{Int64,1})
```

Here's an example where you might want to write a method definition that applies to all one-dimensional integer arrays. It finds all the prime numbers in an array:

```
function findprimes{T<:Integer}(a::Array{T,1})
   find(isprime, a)
end
findprimes (generic function with 1 method)

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

but can't be used for arrays of real numbers:

```
julia> findprimes([1.0:20])
ERROR: `findprimes` has no method matching findprimes(::Array{Float64,1})
```

Note that, in this simple example, because you're not using the type information inside the method definition, you might be better off sticking to the simpler way of defining types, by adding qualifiers to the arguments:

```
function findprimes1(a::Array{Int64,1})
   find(isprime, a)
end
```

But if you wanted to do things inside the method that depended on the types of the arguments, then the type parameters approach will be useful.

7 Dictionaries and Sets

7.1 Dictionaries and sets

7.1.1 Dictionaries

Many of the functions introduced so far have been shown working on arrays (and tuples). But arrays are just one type of collection. Julia has others.

A simple look-up table is a useful way of organizing many types of data: given a single piece of information, such as a number or a string, called the **key** , what is the corresponding data **value** ? For this purpose, Julia provides the Dictionary object, called Dict for short. It's an "associative collection".

7.1.2 Creating dictionaries

In Julia version 0.3, you can create simple dictionaries using the curly braces and arrows syntax. For example:

```
julia> dict = {"a" => 1, "b" => 2, "c" => 3}
Dict{Any,Any} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 1
```

`dict` is now a dictionary. The keys in this particular example, "a", "b", and "c", are strings, but could be any data type - strings, symbols, or even numbers. The corresponding values are 1, 2, and 3, so "a" is associated with 1, "b" with 2, and so on. Keys are unique — you can't have two or more keys with the same value.

This declaration, using curly braces, created a dictionary of type Any. But, as with arrays, you can use square brackets to create a dictionary of a specific type.

You can also create dictionaries using the comprehension syntax:

```
[i => sind(i) for i = 0:5:360]
```

and use the following syntax to create an empty dictionary:

```
julia> (String => Any)[]
Dict{String,Any} with 0 entries
```

⚠ **Warning**

All this changes in version 0.4 of Julia:

```
dict = Dict("a" => 1, "b" => 2, "c" => 3)
```

creates a dictionary, inferring the type of keys and values if possible.

```
dict = Dict{Any,Any}(a=>1, "b" => 2)
```

constructs an untyped dictionary, where the keys and values can be strings, symbols, or numbers.

```
dict = Dict{ASCIIString, Int64}()
```

constructs a empty dictionary with the specified types for keys and values.

The syntax for creating dictionary using comprehension will be:

```
Dict([i => sind(i) for i = 0:5:360])
```

7.1.3 Looking things up

To get a value, given a key:

```
julia> dict = ["a"=> 1 ,"b"=> 2,"c"=>3 ,"d"=>4 ,"e"=>5]
julia> dict["a"]
1
```

if the keys are strings. Or, if the keys are symbols:

```
julia> symdict = [:x => 1, :y => 3, :z => 6]
Dict{Symbol,Int64} with 3 entries:
  :z => 6
  :x => 1
  :y => 3

julia> symdict[:x]
1
```

You can also use the `get()` function, and provide a default value if there's no value for that particular key:

```
julia> get(dict, "a", 0)
1
julia> get(dict, "1", 0)
0
```

If you don't use a default value as a safety precaution, you'll get an error if there's no key:

```
julia> dict = {"a"=> 1 ,"b"=> 2,"c"=>3 ,"d"=>4 ,"e"=>5 }
Dict{Any,Any} with 5 entries:
  "c" => 3
  "e" => 5
  "b" => 2
  "a" => 1
  "d" => 4

julia> get(dict, "w", 0)
0
julia> get(dict, "w")
ERROR: `get` has no method matching get(::Dict{Any,Any}, ::ASCIIString)
```

If you don't want use `get()` to provide a default value, use a `try` ...`catch` block:

```
try
    dict["f"]
    catch error
       if isa(error, KeyError)
           println("sorry, I couldn't find anything")
       end
end

sorry, I couldn't find anything
```

To change a value assigned to an existing key (or assign a value to a hitherto unseen key):

```
julia> dict["a"] = 10
10
```

(Remember that keys must be unique for a dictionary. There's always only one key called `a` in this dictionary, so when you assign a value to a key that already exists, you're not creating a new one, just modifying an existing one.)

To see if the dictionary contains a key, use `haskey()` :

```
julia> haskey(dict, "d")
false
```

To check for the existence of a key/value pair:

```
julia> in(("b",2),dict)
true
```

To add a new key and value to a dictionary, use this:

```
julia> dict["d"] = 4
4
```

You can delete a key from the dictionary, using `delete!()` :

```
julia> delete!(dict, "d")
Dict{Any,Any} with 3 entries:
  "c" => 3
  "b" => 2
  "a" => 10
```

You'll notice that the dictionary doesn't seem to be sorted in any way — the keys are in no particular order. This is due to the way they're stored, and you can't sort them in place. (But see Sorting, below.)

To get all keys, use the `keys()` function:

```
julia> keys(dict)
KeyIterator for a Dict{Any,Any} with 3 entries. Keys:
  "c"
  "b"
  "a"
```

The result is a Key iterator, that, as its names suggests, is ideally suited for iterating through a dictionary key by key:

```
julia> [key for key in  keys(dict)]
6-element Array{Any,1}:
 "f"
 "c"
 "e"
 "b"
 "a"
 "d"
```

To get all values, use the `values()` function:

```
julia> values(dict)
ValueIterator for a Dict{Any,Any} with 3 entries. Values:
  3
  2
  10
```

If you want to go through a dictionary and process each key/value, you can make use the fact that dictionaries themselves are iterable objects:

```
julia> for e in dict
    println(e)
end
("c",3)
("b",2)
("a",5)
```

where `e` is a tuple for each key/value pair. Or you could do:

```
julia> for k in keys(dict)
    println(k, " ==> ", dict[k])
end
c ==> 3
b ==> 2
a ==> 5
```

Even better, you can use a key/value tuple to simplify the iteration even more:

```
julia> for (key,value) in dict
              println(key, " ==> ", value)
end
c ==> 3
b ==> 2
a ==> 5
```

Here's another example:

```
for tuple in ["1"=>"Hydrogen", "2"=>"Helium", "3"=>"Lithium"]
    println("Element $(tuple[1]) is  $(tuple[2])")
end
Element 1 is  Hydrogen
Element 2 is  Helium
Element 3 is  Lithium
```

(Notice the string interpolation operator, `$` . This allows you to use a variable's name in a string and get the variable's value when the string is printed. You can include any Julia expression in a string using `$()` .)

You don't always have to use strings as dictionary keys. Here's an example where integers are keys:

```
julia> nn = Dict{Int64, ASCIIString}()
Dict{Int64,ASCIIString} with 0 entries

julia> nn[1] = "One"
"One"

julia> nn[2] = "Two"
"Two"

julia> nn[3] = "Three"
"Three"

julia> dump(nn)
Dict{Int64,ASCIIString} len 3
  2: ASCIIString "Two"
  3: ASCIIString "Three"
  1: ASCIIString "One"
```

You can also use other data types as dictionary keys.

7.1.4 Sorting a dictionary

Because dictionaries aren't sorted, you have to do a little more work to get a sorted dictionary:

```
julia> dict = {"a" => 1,"b" =>2 ,"c" =>3 ,"d" =>4 ,"e" =>5 ,"f" =>6 }
Dict{Any,Any} with 6 entries:
  "f" => 6
  "c" => 3
  "e" => 5
  "b" => 2
  "a" => 1
  "d" => 4

julia> for key in sort(collect(keys(dict)))
           println("$key => $(dict[key])")
       end
a => 1
b => 2
c => 3
d => 4
e => 5
f => 6
```

7.1.5 Simple example: counting words

A simple application of a dictionary is to count how many times each word appears in a piece of text. Each word is a key, and the value of the key is the number of times that word appears in the text.

Let's count the words in the Sherlock Holmes stories. I've downloaded the text into the file ”sherlock-holmes-the-complete-canon”:

```
julia> canon = open("sherlock-holmes-the-complete-canon")
```

To create the list of words, it suffices to split the text using a regular expression, after converting the text to lower case:

```
julia> wordlist = split(lowercase(readall(canon)), r"\W", false)

669336-element Array{SubString{UTF8String},1}:
 "the"
 "complete"
 "sherlock"
 "holmes"
 "arthur"
 "conan"
 "doyle"
 "table"
 "of"
 "contents"
 "a"
 "study"
 "in"
 "scarlet"
 ⋮
 ”well”
 ”mackinnon”
 ”is”
 ”a”
 ”good”
 ”fellow”
 ”said”
 ”holmes”
 ”with”
 ”a”
 ”tolerant”
 ”smile”
 ”you”
 ”can”
 ”file”
 ”it”
 ”in”
 ”our”
 ”archives”
 ”watson”
 ”some”
 ”day”
 ”the”
 ”true”
 ”story”
 ”may”
 ”be”
 ”told”
julia> close(canon)
```

To store the words and the word counts, create a dictionary:

```
julia> wordcounts = Dict{ASCIIString,Int64}()

Dict{ASCIIString,Int64} with 0 entries
```

To build the dictionary, loop through the list of words, and use `get()` to look up the current tally, if any. If the word has already been seen, the count can be increased. If the word hasn't been seen before, the third argument of `get()` ensures that the absence doesn't cause an error, and 1 is stored instead.

```
for word in wordlist
    wordcounts[word]=get(wordcounts, word, 0) + 1
end
```

Now you can look up words in the `wordcounts` dictionary and find out how many times they appear:

```
julia> wordcounts["watson"]
1040

julia> wordcounts["holmes"]
3057

julia> wordcounts["sherlock"]
415

julia> wordcounts["lestrade"]
244
```

Dictionaries aren't sorted, but you can use the `collect()` and `keys()` functions on the dictionary to collect the keys and then sort them. In a loop you can work through the dictionary in alphabetical order:

```
julia> for i in sort(collect(keys(wordcounts)))
  println("$i, $(wordcounts[i])")
end

000, 5
1, 8
10, 7
100, 4
1000, 9
104, 1
109, 1
10s, 2
10th, 1
11, 9
1100, 1
117, 2
117th, 2
11th, 1
12, 2
120, 2
126b, 3
 ⋮
zamba, 2
zeal, 5
zealand, 3
zealous, 3
zenith, 1
zeppelin, 1
zero, 2
zest, 3
zig, 1
zigzag, 3
zigzagged, 1
zinc, 3
zion, 2
zoo, 1
zoology, 2
zu, 1
zum, 2
```

But how do you find out the most common words? One way is to use `collect()` to convert the dictionary to an array of tuples, and then to sort the array by looking at the last value of each tuple:

```
julia> sort(collect(wordcounts), by = tuple -> last(tuple), rev=true)

19171-element Array{(ASCIIString,Int64),1}:
 ("the",36244)
 ("and",17593)
 ("i",17357)
 ("of",16779)
 ("to",16041)
 ("a",15848)
 ("that",11506)
 ⋮
 ("enrage",1)
 ("smuggled",1)
 ("lounges",1)
 ("devotes",1)
 ("reverberated",1)
 ("munitions",1)
 ("graybeard",1)
```

To see only the top 20 words:

```
julia> sort(collect(wordcounts), by = tuple -> last(tuple), rev=true)[1:20]

20-element Array{(ASCIIString,Int64),1}:
 ("the",36244)
 ("and",17593)
 ("i",17357)
 ("of",16779)
 ("to",16041)
 ("a",15848)
 ("that",11506)
 ("it",11101)
 ("in",10766)
 ("he",10366)
 ("was",9844)
 ("you",9688)
 ("his",7836)
 ("is",6650)
 ("had",6057)
 ("have",5532)
 ("my",5293)
 ("with",5256)
 ("as",4755)
 ("for",4713)
```

In a similar way, you can use the `filter()` function to find, for example, all words that start with "k" and occur less than four times:

```
julia> filter(tuple -> beginswith(first(tuple), "k") && last(tuple) < 4,
 collect(wordcounts))
73-element Array{(ASCIIString,Int64),1}:
 ("keg",1)
 ("klux",2)
 ("knifing",1)
 ("keening",1)
 ("kansas",3)
 ⋮
 ("kaiser",1)
 ("kidnap",2)
 ("keswick",1)
 ("kings",2)
 ("kratides",3)
 ("ken",2)
 ("kindliness",2)
 ("klan",2)
 ("keepsake",1)
```

("kindled",2)
("kit",2)
("kicking",1)
("kramm",2)
("knob",1)

7.1.6 Sets

A set is a collection of elements, just like an array or dictionary.

The two important differences between a set and other types of collection is that in a set you can have only one of each element, and, in a set, the order of elements isn't important. An array, of course, can have multiple copies of elements, and their order is remembered:

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

You can create an empty set using the `Set` constructor function:

```
julia> colors = Set()
Set{Any}({})
```

As elsewhere in Julia, you can use curly braces and omit the type specifications, so that each element can be any type, or specify the type for every element:

```
julia> primes = Set{Int64}()
Set{Int64}({})
```

You can create and fill sets in one go:

```
julia> colors = Set({"red","green","blue","yellow"})
Set{Any}({"yellow","blue","green","red"})
```

or you can use square brackets if you want Julia to play "guess the type":

```
julia> colors = Set(["red","green","blue","yellow"])
Set{ASCIIString}({"yellow","blue","green","red"})
```

Quite a few of the functions that work with arrays also work with sets. Adding elements to sets, for example, is a bit like adding elements to arrays. You can use `push!()` :

`julia> push!(colors, "black") Set{ASCIIString}({"yellow","blue","green","black","red"})`

But you can't use `unshift!()` , because that works only for ordered things like arrays. What happens if you try to add something to the set that's already there? Absolutely nothing. You don't get a copy added, because it's a set, not an array, and sets don't store repeated elements. To see if something is in the set, you can use `in()` :

```
julia> in("green", colors)
true
```

There are some standard operations you can do with sets, namely find their **union** , **intersection** , and **difference** , with the functions, `union()` , `intersect()` , and `setdiff()` :

```
julia> rainbow = Set({"red","orange","yellow","green","blue","indigo","violet"})
Set{Any}({"indigo","yellow","orange","blue","violet","green","red"})
```

The union of two sets is the set of everything that is in one or the other sets. The result is another set — so you can't have two "yellow"s here, even though we've got a "yellow" in each set:

```
julia> union(colors, rainbow)
Set{Any}({"indigo","yellow","orange","blue","violet","green","black","red"})
```

The intersection of two sets is the set that contains every element that belongs to both sets:

```
julia> intersect(colors, rainbow)
Set{Any}({"yellow","blue","green","red"})
```

The difference between two sets is the set of elements that are in the first set, but not in the second. This time, the order in which you supply the sets matters. The `setdiff()` function finds the elements that are in the first set, `colors` , but not in the second set, `rainbow` :

```
julia> setdiff(colors, rainbow)
Set{Any}({"black"})
```

8 Strings and Characters

8.1 Strings and characters

8.1.1 Strings

A string is a sequence of one or more characters, usually seen when enclosed in double quotes:

```
"this is a string"
```

If you want to include a double quote character in the string, it has to be preceded with a backslash, otherwise the rest of the string would be interpreted as Julia code, with potentially interesting results. And if you want to include a dollar sign ($) in a string, that should also be prefaced by a backslash, because it's used for string interpolation[1].

```
julia> demand = "You owe me \$50!"
"You owe me \$50!"

julia> println(demand)
You owe me $50!
```

Strings can also be enclosed in triple double quotes. This is useful because you can use ordinary double quotes inside the string *without* having to put backslashes before them:

```
julia> """this is "a" string"""
"this is \"a\" string"
```

You'll encounter a few specialized types of string too:

- `r" "` indicates a regular expression
- `v" "` indicates a version string
- `b" "` indicates a byte literal

Strings are immutable objects, so you can't change them once they're created. But it's easy to make new strings from parts of existing ones.

8.1.2 Substrings

To extract a smaller string from a string, use the same techniques that you use to extract a group of elements from arrays:

```
julia> s = "a load of characters"
"a load of characters"
julia> s[1:end]
```

1 Chapter 8.1.4 on page 106

```
"a load of characters"
julia> s[3:6]
"load"
julia> s[3:end-6]
"load of char"
julia> s[1:1]
"a"
```

But watch out:

```
julia> s[1]
'a'
```

The result is a character (single quotes), not a string (double quotes).

8.1.3 Splitting and joining strings

You can stick strings together (a process often called *concatenation*) using the multiply (`*`) operator:

```
julia> "s" * "t"
"st"
```

If you've come from other languages, you might expect to use the addition (`+`) operator:

```
julia> "s" + "t"
ERROR: `+` has no method matching +(::ASCIIString, ::ASCIIString)
```

If you can 'multiply' strings, you can also raise them to a power:

```
julia> "s" ^ 18
"ssssssssssssssssss"
```

You can also use `string()` :

```
julia> string("s", "t")
"st"
```

but if you want to do a lot of concatenation, inside a loop, perhaps, it might be better to use the string buffer approach (see below).

To split a string, use `split(string, chars)` . Given this simple string:

```
julia> s = "You know my methods, Watson."
"You know my methods, Watson."
```

a simple call to the `split()` function divides the string at the spaces, returning a five-piece array:

```
julia> split(s)
5-element Array{SubString{ASCIIString},1}:
 "You"
 "know"
 "my"
 "methods,"
 "Watson."
```

Or you can specify the character or characters to split at:

```
julia> split(s, "e")
2-element Array{SubString{ASCIIString},1}:
 "You know my m"
 "thods, Watson."


julia> split(s, " m")
3-element Array{SubString{ASCIIString},1}:
 "You know"
 "y"
 "ethods, Watson."
```

The characters you use to do the splitting don't appear in the final result:

```
julia> split(s, "hod")
2-element Array{SubString{ASCIIString},1}:
 "You know my met"
 "s, Watson."
```

If you want to split a string into separate single-character strings, use the empty string ("") which splits the string *between* the characters:

```
julia> split(s,"")
28-element Array{SubString{ASCIIString},1}:
 "Y"
 "o"
 "u"
 " "
 "k"
 "n"
 "o"
 "w"
 " "
 "m"
 "y"
 " "
 "m"
 "e"
 "t"
 "h"
 "o"
 "d"
 "s"
 ","
 " "
 "W"
 "a"
 "t"
 "s"
 "o"
 "n"
 "."
```

You can also split strings using a regular expression to define the splitting points. Use the special regex string construction `r" "` :

```
julia> split(s, r"a|e|i|o|u")
8-element Array{SubString{ASCIIString},1}:
 "Y"
 ""
 " kn"
 "w my m"
 "th"
 "ds, W"
 "ts"
 "n."
```

Here, the `r"a|e|i|o|u"` is a regular expression string, and — as you'll know if you love regular expressions as much as I do :) — that this matches any of the vowels. So the resulting array consists of the string split at every vowel. Notice the empty string -— if you don't want those, add a *false* flag at the end:

```
julia> split(s, r"a|e|i|o|u", false)
7-element Array{SubString{ASCIIString},1}:
 "Y"
 " kn"
 "w my m"
 "th"
 "ds, W"
 "ts"
 "n."
```

If you wanted to keep the vowels, rather than use them for splitting work, you have to delve deeper into the world of regex literal strings. That's to come.

You can join the elements of a split string in array form using `join()` :

```
julia> join(split(s, r"a|e|i|o|u", false), "aiou")
"Yaiou knaiouw my maiouthaiouds, Waioutsaioun."
```

8.1.4 String interpolation

You often want to use the results of Julia expressions inside strings. For example, suppose you want to say:

"The value of x is n."

where `n` is the current value of `x` .

Any Julia expression can be inserted into a string using the `$()` construction:

```
julia> x = 42
42
julia> "The value of x is $(x)."
```

displays:

```
"The value of x is 42."
```

You don't have to use the parentheses if you're just using the name of a variable:

```
julia> "The value of x is $x."
"The value of x is 42."
```

You can include a Julia expression in a string, enclose it in parentheses first, then precede it with a dollar sign:

```
julia> "The value of 2 + 2 is $(2 + 2)."
"The value of 2 + 2 is 4."
```

8.1.5 Character objects

Above we extracted smaller strings from larger strings:

```
julia> s[1:1]
"a"
```

But when we extract a single element from a string:

```
julia> s[1]
'a'
```

- notice the single quotes. In Julia, these are used to mark **character objects,** so `'a'` is a character object, but `"a"` is a string with length 1. These are not equivalent.

You can convert character objects to strings easily enough:

```
julia> string('s') * string('d')
"sd"
```

or

```
julia> string('s', 'd')
"sd"
```

It's easy to input Unicode characters:

```
julia> '\u2640'
'♀'
```

8.1.6 Converting to and from strings

The `dec()` function turns a (decimal) number into a string. The `int()` function turns a character into an integer, and the `char()` function turns an integer into a character. And `string()` converts objects to strings.

If you're deeply attached to C-style `printf()` functionality, you'll be able to use a Julia macro (which are called by prefacing them with the `@` sign):

```
julia> @printf("pi = %0.20f", float(pi))
pi = 3.14159265358979311600
```

or you can create another string using the `sprintf()` macro:

```
julia> @sprintf("pi = %0.20f", float(pi))
"pi = 3.14159265358979311600"
```

8.1.7 Convert a string to an array

To read from a string into an array, you can use the `IOBuffer()` function. This is available with a number of Julia functions (including `printf()`). Here's a string of data (it could have been read from a file):

```
julia> data="1 2 3 4
       5 6 7 8
       9 0 1 2"

"1 2 3 4\n5 6 7 8\n9 0 1 2"
```

Now you can ”read” this string using functions such as `readdlm()` , the ”read with delimiters” function:

```
julia> readdlm(IOBuffer(data))

3x4 Array{Float64,2}:
 1.0  2.0  3.0  4.0
 5.0  6.0  7.0  8.0
 9.0  0.0  1.0  2.0
```

You can add an optional type specification:

```
julia> readdlm(IOBuffer(data), Int)

3x4 Array{Int32,2}:
 1  2  3  4
 5  6  7  8
 9  0  1  2
```

Sometimes you want to do things to strings that you can do better with arrays. Here’s an example.

```
julia> i = "/Users/me/Music/iTunes/iTunes Media/Mobile Applications";
```

Explode the pathname string into character objects:

```
julia> collect(i)
57-element Array{Char,1}:
 '/'
 'U'
 's'
 'e'
 'r'
 's'
 '/'
 ...
```

Count the occurrences of a particular character object, using an anonymous function:

```
julia> count(c -> c =='/', collect(i))
6
```

Similarly, you can use `split()` to split the string and count the results:

```
julia> count(c -> c == "/", split(i, ""))
6
```

8.1.8 Finding and replacing things inside strings

If you want to know whether a string contains a specific character, use the general-purpose `in()` function.

```
julia> s = "Elementary, my dear Watson";
julia> in('m', s)
true
```

The `contains()` function, which accepts two strings, is more generally useful, because you can use substrings with one or more characters. Notice that you place the container first, then the string you're looking for:

```
julia> contains(s, "Wat")
true
julia> contains(s, "m")
true
julia> contains(s, "mi")
false
julia> contains(s, "me")
true
```

You can get the location of the first occurrence of a substring using `search()` . The second argument can be a single character, a vector or a set of characters, a string, or a regular expression:

```
julia> s ="You know my methods, Watson.";
julia> search(s, "h")
16:16

julia> search(s, ['a', 'e', 'i', 'o', 'u'])
2
```

This search is for the first occurrence of any of the set of characters, and 'o' was in the second position.

```
julia> search(s, "meth")
13:16

julia> search(s, r"m.*")
13:14
```

In each case, the result contains the indices of the characters, if present. If not:

```
julia> search(s, "mo")
0:-1

julia> s[0:-1]
""
```

You can also use a regular expression string (`r" "`) with `search()` :

```
julia> search(s, r"m(y|e)")
10:11
```

looks for "my" or "me"

```
julia> s[search(s, r"m(y|e)")]
"my"
```

The `replace()` function returns a new string with a substring of characters replaced with something else:

```
julia> replace("Sherlock Holmes", "e", "ee")
"Sheerlock Holmees"
```

Usually the third argument is another string, as here. But you can also supply a function that processes the result:

```
julia> replace("Sherlock Holmes", "e", uppercase)
"ShErlock HolmEs"
```

where the function (here, the built-in `uppercase()` function) is applied to the matching substring.

8.1.9 Regular expressions

You can use regular expressions to find matches for substrings. Some functions that accept a regular expression are:

- `ismatch()` returns true or false if there's a match for a regular expression
- `match()` returns the first match or nothing
- `matchall()` returns an array of matches
- `eachmatch()` returns an iterator that lets you go through all the matches
- `search()` searches a string for a match
- `split()` splits a string at every match

Here I've loaded the text of ”The Adventures of Sherlock Holmes” into the string called `text` :

```
julia> f = "adventures-of-sherlock-holmes.txt"
julia> text = readall(f);
```

To use the possibility of a match as a Boolean condition, suitable for use in an `if` statement, for example, use `ismatch()` .

```
julia> ismatch(r"Opium", text)
false

julia> ismatch(r"(?i)Opium", text)
true
```

The word ”opium” does appear in the text, but only in lower-case, hence the `false` result — regular expressions are case-sensitive. A case-insensitive search for ”Opium” returns true.

There's an alternative syntax for adding regex modifiers, such as case-insensitive matches. Notice the ”i” following the string:

```
julia> ismatch(r"m"i, s)
true
```

With the `eachmatch()` function, you apply the regex to the string to produce an iterator. For example, to look for substrings in our text matching the letters ”L”, followed by some other characters, ending with ”ed”:

```
julia> lmatch = eachmatch(r"L.*?ed", text);
```

The result in `lmatch` is an iterable object containing all the matches, as RegexMatch objects. Now we can work through the iterator and look at each match in turn. You can access a number of fields of the RegexMatch, to extract information about the match. For example, the `.match` field contains the matched substring:

```
julia>for i in lmatch
   println(i.match)
end

London - quite so! Your Majesty, as I understand, became entangled
Lodge. As it pulled
Lord, Mr. Wilson, that I was a red
League of the Red
League was founded
London when he was young, and he wanted
LSON" in white letters, upon a corner house, announced
League, and the copying of the 'Encyclopaed
Leadenhall Street Post Office, to be left till called
Let the whole incident be a sealed
Lestrade, being rather puzzled
Lestrade would have noted
...
Lestrade," drawled
Lestrade looked
Lord St. Simon has not already arrived
Lord St. Simon sank into a chair and passed
Lord St. Simon had by no means relaxed
Lordship. "I may be forced
London. What could have happened
London, and I had placed
```

Other fields include `.captures` , the captured substrings as an array of strings, `.offset` , the offset into the string at which the whole match begins, and `.offsets` , the offsets of the captured substrings.

If you don't want an iterable object, use the `matchall()` function instead:

```
julia> lmatches = matchall(r"L.*?ed", text);
```

Now the `lmatches` array contains the matching substrings, which you can inspect any way you want:

```
julia> lmatches[4:6]
3-element Array{SubString{UTF8String},1}:
 "League of the Red"
 "League was founded"
 "London when he was young, and he wanted"
```

The basic `match()` function looks for the first match for your regex. Use the `.match` field to extract the information from the RegexMatch object:

```
julia> match(r"She.*",text).match
"Sherlock Holmes she is always THE woman. I have seldom heard\r"
```

8.1.10 Testing and changing strings

There are lots of functions for testing and changing strings:

- `length()`
- `sizeof()`
- `beginswith()`

- `endswith()`
- `contains()`
- `isalnum()`
- `isalpha()`
- `isascii()`
- `isblank()`
- `iscntrl()`
- `isdigit()`
- `islower()`
- `ispunct()`
- `isspace()`
- `isupper()`
- `isxdigit()`
- `uppercase()`
- `lowercase()`
- `ucfirst()`
- `lcfirst()`
- `chop()`
- `chomp()`

8.1.11 Streams

To write to a string, you can use a Julia stream. The `sprint()` (String Print) function lets you use a function as the first argument, and uses the function and the rest of the arguments to send information to a stream.

For example, consider the following function, `f` . The body of the function maps an anonymous 'print' function over the arguments, enclosing them with angle brackets. When used by `sprint` , the function `f` processes the remaining arguments and sends them to the stream, which, with `sprint()` , is a string.

```
julia> function f(io::IO, args...)
    map( (a) -> print(io,"<",a, ">"), args)
end

f (generic function with 1 method)
julia> sprint(f, "fred", "jim", "bill", "fred blogs")

"<fred><jim><bill><fred blogs>"
```

Functions like `println()` can take an IOBuffer or stream as their first argument. This lets you print to streams instead of printing to the standard output device:

```
julia> iobuffer = IOBuffer()

IOBuffer(data=Uint8[...], readable=true, writable=true, seekable=true,
 append=false, size=0, maxsize=Inf, ptr=1, mark=-1)
julia> for i in 1:100
           println(iobuffer, string(i))
       end
```

After this, the in-memory stream called `iobuffer` is full of numbers and newlines, even though nothing was printed on the terminal. To copy the contents of `iobuffer` from the stream to a string or array, you can use `takebuf_string()` (or `takebuf_array()`):

```
julia> takebuf_string(iobuffer)
"1\n2\n3\n4\n5\n6\n7\n8\n9\n10\n11\n12\n13\n14\n15\n16\n17\n18\n19\n20\n21\n22\n
23\n24\n25\n26\n27\n28\n29\n30\n31\n32\n33\n34\n35\n36\n37\n38\n39\n40\n41\n42\n4
3\n44\n45\n46\n47\n48\n49\n50\n51\n52\n53\n54\n55\n56\n57\n58\n59\n60\n61\n62\n63
\n64\n65\n66\n67\n68\n69\n70\n71\n72\n73\n74\n75\n76\n77\n78\n79\n80\n81
\n82\n83\n84\n85\n86\n87\n88\n89\n90\n91\n92\n93\n94\n95\n96\n97\n98\n99\n100\n"
julia>
```

9 Working with Text Files

The standard approach for getting information from a text file is using the `open()` , `read()` , and `close()` functions.

9.1 Reading from files

9.1.1 Open

To read text from a file, you obtain a file handle:

```
f = open("sherlock-holmes.txt")
```

`f` is now Julia's connection to the file on disk.

9.1.2 Slurp — reading a file all at once

You can read the entire contents of the file at once with readall():

```
  s = readall(f)
```

This returns a string, complete with newlines.

Or use readlines() to read in the whole file as an array, with each line an element:

```
julia> f = open("sherlock-holmes.txt");
julia> lines = readlines(f)
12640-element Array{Union(UTF8String,ASCIIString),1}:
 "THE ADVENTURES OF SHERLOCK HOLMES by SIR ARTHUR CONAN DOYLE\r\n"
 "\r\n"
 "   I. A Scandal in Bohemia\r\n"
 "  II. The Red-headed League\r\n"
 ...
 "Holmes, rather to my disappointment, manifested no further\r\n"
 "interest in her when once she had ceased to be the centre of one\r\n"
 "of his problems, and she is now the head of a private school at\r\n"
 "Walsall, where I believe that she has met with considerable success.\r\n"
```

Now you can step through the lines:

```
counter = 1
for l in lines
   println("$counter $l")
   counter += 1
end


1 THE ADVENTURES OF SHERLOCK HOLMES by SIR ARTHUR CONAN DOYLE
2
3    I. A Scandal in Bohemia
```

```
4   II. The Red-headed League
5  III. A Case of Identity
6   IV. The Boscombe Valley Mystery
...
12638 interest in her when once she had ceased to be the centre of one
12639 of his problems, and she is now the head of a private school at
12640 Walsall, where I believe that she has met with considerable success.
```

There's a better way to do this — see `enumerate()` , below.

9.1.3 Line by line

The `eachline()` function turns a source into an iterator. This allows you to process a file a line at a time:

```
f = open("sherlock-holmes.txt");
for ln in eachline(f)
       print("$(length(ln)), $ln")
end
```

```
1, THE ADVENTURES OF SHERLOCK HOLMES by SIR ARTHUR CONAN DOYLE
2,
28,     I. A Scandal in Bohemia
29,    II. The Red-headed League
26,   III. A Case of Identity
35,    IV. The Boscombe Valley Mystery
…
62, the island of Mauritius. As to Miss Violet Hunter, my friend
60, Holmes, rather to my disappointment, manifested no further
66, interest in her when once she had ceased to be the centre of one
65, of his problems, and she is now the head of a private school at
70, Walsall, where I believe that she has met with considerable success.
```

Another approach is to read until you reach the end of the file. Typically you also want to keep track of which line you're on:

```
 f = open("sherlock-holmes.txt")
 line = 1
 while !eof(f)
    x = readline(f)
    println("$line $x")
    line += 1
 end
```

An easier approach is to use `enumerate()` on an iterable object — you'll get the line numbering for free:

```
for i in enumerate(eachline(open(" holmes.txt")))
    println(i)
end
```

Using a `do` block works like this:

```
open("sherlock-holmes.txt") do filehandle
  for line in eachline(filehandle)
      println(length(line), line)
  end
 end
```

or

```
open("sherlock-holmes.txt") do f
    uppercase(readall(f))
end
```

See Controlling the flow[1] for more about `do` blocks.

With all these file functions, remember to:

```
close(f)
```

i.e. close the file when you're done with it.

If you have a specific function that you want to call on a file, you can use this alternative syntax instead of the open/read/close sequence:

```
function func(f::IOStream)
    return uppercase(readall(f))
end
open(func, filename)
```

This opens the file, runs the function on it, then closes it again, returning the processed contents.

You can use `readcsv()` and `readdlm()` functions to read lines from CSV files or files delimited with certain characters, such as data files, arrays stored as text files, and tables. And if you use the DataFrames package, there's also a `readtable()` specifically designed to read data into a table.

9.2 Working with paths and filenames

These functions will be useful for working with filenames:

- `cd(path)` changes the current directory
- `readdir(path)` returns a lists of the contents of the named directory, or the current directory,
- `abspath(path)` returns the full path of the string
- `joinpath(str, str, ...)` assembles a pathname from a pieces
- `isdir(path)` tells you whether the path is a directory

Putting these together, here's a function for walking a directory tree recursively:

```
function walkdir(dir)
    f = readdir(abspath(dir))
    for i in f
        spaces = length(matchall(r"/", joinpath(dir,i)))
        println(" "^ (spaces * 3),  joinpath(dir,i))
        if isdir(joinpath(dir,i))
            walkdir(joinpath(dir,i))
        end
    end
end
```

```
julia> walkdir("/Users/me/.julia/v0.3/")
            /Users/me/.julia/v0.3/.DS_Store
            /Users/me/.julia/v0.3/.cache
               /Users/me/.julia/v0.3/.cache/ASCIIPlots
                  /Users/me/.julia/v0.3/.cache/ASCIIPlots/HEAD
```

1 Chapter 5.1.7 on page 76

```
                     /Users/me/.julia/v0.3/.cache/ASCIIPlots/config
                     /Users/me/.julia/v0.3/.cache/ASCIIPlots/objects
                        /Users/me/.julia/v0.3/.cache/ASCIIPlots/objects/info
                        /Users/me/.julia/v0.3/.cache/ASCIIPlots/objects/pack
                           /Users/me/.julia/v0.3/
.cache/ASCIIPlots/objects/pack/pack-f79a0577cec6e3c8d1c154c2a5fe92a793cefe26.idx
                           /Users/me/.julia/v0.3/.
cache/ASCIIPlots/objects/pack/pack-f79a0577cec6e3c8d1c154c2a5fe92a793cefe26.pack
                     /Users/me/.julia/v0.3/.cache/ASCIIPlots/packed-refs
                     /Users/me/.julia/v0.3/.cache/ASCIIPlots/refs
                        /Users/me/.julia/v0.3/.cache/ASCIIPlots/refs/heads
                        /Users/me/.julia/v0.3/.cache/ASCIIPlots/refs/tags
                     /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/branches

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/description
                        /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/applypatch-msg.sample

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/commit-msg.sample

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/post-update.sample

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/pre-applypatch.sample

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/pre-commit.sample

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/pre-push.sample

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/pre-rebase.sample
                           /
Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/prepare-commit-msg.sample

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/hooks/update.sample
                        /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/info

 /Users/me/.julia/v0.3/.cache/ASCIIPlots/templates/info/exclude
                  /Users/me/.julia/v0.3/.cache/ArrayViews
                     /Users/me/.julia/v0.3/.cache/ArrayViews/HEAD
                     /Users/me/.julia/v0.3/.cache/ArrayViews/config
                     /Users/me/.julia/v0.3/.cache/ArrayViews/objects
                        /Users/me/.julia/v0.3/.cache/ArrayViews/objects/info
                        /Users/me/.julia/v0.3/.cache/ArrayViews/objects/pack
                           /Users/me/.julia/v0.3/
.cache/ArrayViews/objects/pack/pack-581bd75df9315438e4d4569fb264d22a4f027944.idx
                           /Users/me/.julia/v0.3/.
cache/ArrayViews/objects/pack/pack-581bd75df9315438e4d4569fb264d22a4f027944.pack
                     /Users/me/.julia/v0.3/.cache/ArrayViews/packed-refs
                     /Users/me/.julia/v0.3/.cache/ArrayViews/refs
                        /Users/me/.julia/v0.3/.cache/ArrayViews/refs/heads
                        /Users/me/.julia/v0.3/.cache/ArrayViews/refs/tags
                     /Users/me/.julia/v0.3/.cache/ArrayViews/templates

 /Users/me/.julia/v0.3/.cache/ArrayViews/templates/branches

 /Users/me/.julia/v0.3/.cache/ArrayViews/templates/description
                        /Users/me/.julia/v0.3/.cache/ArrayViews/templates/hooks
...
```

Here, the rough-and-ready `println()` expression makes a guess at how far to indent each line, based on how many ”/” strings the full pathname contains.

There are plenty more useful file-handling functions:

- `splitdir(path)` - split a path into a tuple of the directory name and file name.
- `splitdrive(path)` - on Windows, split a path into the drive letter part and the path part. On Unix systems, the first component is always the empty string.
- `splitext(path)` - if the last component of a path contains a dot, split the path into everything before the dot and everything including and after the dot. Otherwise, return a tuple of the argument unmodified and the empty string.
- `expanduser(path)` - replace a tilde character at the start of a path with the current user's home directory.
- `normpath(path)` - normalize a path, removing "." and ".." entries.
- `realpath(path)` - canonicalize a path by expanding symbolic links and removing "." and ".." entries.
- `homedir()` - current user's home directory.
- `dirname(path)` - get the directory part of a path.
- `basename(path)` - get the file name part of a path.

To work on a restricted selection of files in a directory, use `filter()` and an anonymous function to filter the file names and just keep the ones you want. (`filter()` is more of a fishing net or sieve, and less of a coffee filter, in that it catches what you want to keep.)

```
for f in filter(x -> endswith(x, "jl"), readdir())
    println(f)
end

Astro.jl
calendar.jl
constants.jl
coordinates.jl
...
pseudoscience.jl
riseset.jl
sidereal.jl
sun.jl
utils.jl
vsop87d.jl
```

If you want to match a group of files using a regular expression, then use `ismatch()` . Let's look for both JPG and PNG files (remembering to escape the "."):

```
for f in filter(x -> ismatch(r"\.jpg|\.png", x), readdir())
    println(f)
end

034571172750.jpg
034571172750.png
51ZN2sCNfVL._SS400_.jpg
51bU7lucOJL._SL500_AA300_.jpg
Voronoy.jpg
kblue.png
korange.png
penrose.jpg
r-home-id-r4.png
wave.jpg
```

9.2.1 File information

If you want information about a specific file, use `stat("pathname")` , and then use one of the fields to find out the information. Here's how to get all the information and the field names listed for a file "i":

```
julia> for n in names(stat(i))
    println(n, ": ", getfield(stat(i),n))
end

device: 16777219
inode: 2955324
mode: 16877
nlink: 943
uid: 502
gid: 20
rdev: 0
size: 32062
blksize: 4096
blocks: 0
mtime:1.409769933e9
ctime:1.409769933e9
julia>
```

9.2.2 Interacting with the file system

The `cp()` , `mv()` , `rm()` , and `touch()` functions have the same names and functions as their Unix shell counterparts.

To convert filenames to pathnames, use `abspath()` . You can map this over a list of files in a directory:

```
julia> map(abspath,readdir())
67-element Array{ASCIIString,1}:
 "/Users/me/.CFUserTextEncoding"
 "/Users/me/.DS_Store"
 "/Users/me/.Trash"
 "/Users/me/.Xauthority"
 "/Users/me/.ahbbighrc"
 "/Users/me/.apdisk"
 "/Users/me/.atom"
...
```

To restrict the list to filenames that contain a particular substring, use an anonymous function inside `filter()` — something like this:

```
julia> filter(x -> contains(x, "re"),map(abspath,readdir()))
4-element Array{ASCIIString,1}:
 "/Users/me/.DS_Store"
 "/Users/me/.gitignore"
 "/Users/me/.hgignore_global"
 "/Users/me/Pictures"
 ...
```

9.3 Writing to files

To write to a file, open it using the "w" flag and make sure that you have permission to create the file in the current directory:

```
outfile = open("/tmp/temp.txt", "w")
```

Then you can write a line to a file using `write()` :

```
write(outfile,"A, B, C, D\n")
```

Here's how to write 20 lines of 4 random numbers, separated by commas:

```
function fourrandom()
    return rand(Int,4)
end

for i = 1:20
     n1, n2, n3, n4  = fourrandom()
     write(outfile, join((n1, n2, n3, n4), ","), "\n")
end

close(outfile)
```

Don't forget to close the file when you've finished.

9.3.1 Writing and reading array to and from a file

If you have an array that you want to save in a text file, or if you want to read data from a file into an array, you can use the convenient `writedlm()` and `readdlm()` functions.

`writedlm()` writes the contents of an array (or any iterable object) to a text file, and `readdlm()` reads the data from a file into an array:

```
julia> numbers = rand(5,5)
5x5 Array{Float64,2}:
 0.913583  0.312291  0.0855798  0.0592331  0.371789
 0.13747   0.422435  0.295057   0.736044   0.763928
 0.360894  0.434373  0.870768   0.469624   0.268495
 0.620462  0.456771  0.258094   0.646355   0.275826
 0.497492  0.854383  0.171938   0.870345   0.783558

julia> writedlm("/tmp/test.txt", numbers)
```

You can see the file using the shell (type a semicolon ";" to switch):

```
<shell>  cat "/tmp/test.txt"
.913583332
8830523
.3122905420350348    .08557977218948465    .0592330821115965    .3717889559226475
.13747015238
054083
.42243494637594203    .29505701073304524    .7360443978397753
.7639280496847236
.3608943267
2073607
.43437288984307787    .870767989032692    .4696243851552686    .26849468736154325
.6204624598
015906
.4567706404666232    .25809436255988105    .6463554854347682
.27582613759302377
.49749166
```

```
25466639
.8543829989347014    .17193814498701587    .8703447748713236    .783557793485824
```

The elements are separated by tabs unless you specify another delimiter. Here, a colon is used to delimit the numbers:

```
julia> writedlm("/tmp/test.txt", rand(1:6, 10, 10), ":")
shell>  cat "/tmp/test.txt"
3:3:3:2:3:2:6:2:3:5
3:1:2:1:5:6:6:1:3:6
5:2:3:1:4:4:4:3:4:1
3:2:1:3:3:1:1:1:5:6
4:2:4:4:4:2:3:5:1:6
6:6:4:1:6:6:3:4:5:4
2:1:3:1:4:1:5:4:6:6
4:4:6:4:6:6:1:4:2:3
1:4:4:1:1:1:5:6:5:6
2:4:4:3:6:6:1:1:5:5
```

To read in data from a text file, you can use `readdlm()` .

```
julia> numbers = rand(5,5)
5x5 Array{Float64,2}:
 0.862955  0.00827944  0.811526  0.854526  0.747977
 0.661742  0.535057    0.186404  0.592903  0.758013
 0.800939  0.949748    0.86552   0.113001  0.0849006
 0.691113  0.0184901   0.170052  0.421047  0.374274
 0.536154  0.48647     0.926233  0.683502  0.116988

julia> writedlm("/tmp/test.txt", numbers)

julia> numbers = readdlm("/tmp/test.txt")
5x5 Array{Float64,2}:
 0.862955  0.00827944  0.811526  0.854526  0.747977
 0.661742  0.535057    0.186404  0.592903  0.758013
 0.800939  0.949748    0.86552   0.113001  0.0849006
 0.691113  0.0184901   0.170052  0.421047  0.374274
 0.536154  0.48647     0.926233  0.683502  0.116988
```

Since it's so common to use files where the elements are separated with commas rather than tabs (CSV files), Julia provides "-csv" versions of these "-dlm" functions, `writecsv()` and `readcsv()` . As ever, refer to the official documentation for options and keywords.

10 Working with Dates and Times

10.1 Working with dates and times

To work with dates, you must first import the Dates package (if you're still using version 0.3 — version 0.4 includes this package):

```
julia> using Dates
```

The documentation is extensive but, to save you a few hours, here are the basic essentials.

10.1.1 Date and DateTimes

A Date object represents just a date: no time zones, no daylight saving issues, etc.. It's accurate to, well, a day.

A DateTime object is a combination of a date and a time of day, and so it specifies an exact moment in time. It's accurate to a millisecond or so.

Use these two constructors to make the type of object you want:

```
julia> birthday = Date(1997,3,15)
1997-03-15
julia> armistice = DateTime(1918,11,11,11,11,11)
1918-11-11T11:11:11
```

The `today()` and `now()` functions return a Date and DateTime object for the current instant in time:

```
julia> datetoday = today()
2014-09-02


julia> datetimenow = now()
2014-09-02T08:20:07
```

To create an object from a formatted string, use the `DateTime()` function in Dates, and supply a suitable format string that matches the formattting:

```
julia> Dates.DateTime("20140529 120000","yyyymmdd HHMMSS")
2014-05-29T12:00:00
julia> Dates.DateTime("18/05/2009 16:12","dd/mm/yyyy HH:MM")
2009-05-18T16:12:00
julia> today = Dates.DateTime("2014-09-02T08:20:07") # defaults to ISO8601
 format
2014-09-02T08:20:07
```

10.1.2 Date and time queries

Once you have a date/time or date object, you can extract particular information from it with the following functions. For both date and datetime objects:

```
julia> year(birthday)
1997
julia> year(today)
2014
julia> month(birthday)
3
julia> month(today)
9
julia> day(birthday)
15
julia> day(today)
2
```

and, for date/time objects:

```
julia> minute(now())
37
julia> hour(now())
16
julia> second(now())
8
```

There's also a bunch of other useful ones:

```
julia> dayofweek(birthday)
6
julia> dayname(birthday)
"Saturday"
julia> yearmonth(now())
(2014,9)
julia> yearmonthday(birthday)
(1997,3,15)
julia> isleapyear(birthday)
false
julia> daysofweekinmonth(today)
5
julia> monthname(birthday)
"March"
julia> monthday(now())
(9,2)
julia> dayofweekofmonth(birthday)
3
```

The `daysofweekinmonth()` tells you how many days there are in the month with the same day name as the specified day — there are five Tuesdays in the current month (at the time of writing). The last function, `dayofweekofmonth(birthday)` , tells us that the 15th of March, 1997, was the third Saturday of the month.

10.1.3 Date arithmetic

You can do arithmetic on dates and date/time objects. Subtracting two dates to find the difference is the most obvious one:

```
julia> datetoday - birthday
6380 days
```

```
julia> datetimenow - armistice
3023472252000 milliseconds
```

which you can convert to a number:

```
julia> int(datetoday - birthday)
6380
julia> int(datetimenow - armistice) / (1000 * 60 * 60 * 24)
34993.891805555555
```

which is the number of days.

To add and subtract periods of time to date and date/time objects, use the `Dates.*` constructors to specify the period. For example, `Dates.Year(20)` defines a period of 20 years, and `Dates.Month(6)` defines a period of 6 months. So, to add 20 years and 6 months to the birthday date:

```
julia> birthday + Dates.Year(20) + Dates.Month(6)
2017-09-15
```

Here's 6 months ago from now:

```
julia> now() - Dates.Month(6)
2014-03-02T16:43:08
```

and similarly for months, weeks:

```
julia> now() - Dates.Year(2) - Dates.Month(6)
2012-03-02T16:44:03
```

10.1.4 Range of dates:

You can make iterable range objects that define a range of dates:

```
julia> d = [Date(1980,1,1):Month(3):Date(2015,1,1)]
141-element Array{Date,1}:
 1980-01-01
 1980-04-01
```

and, to find out which are weekdays, provide an anonymous function to `filter` that tests the day name against the given day names:

```
weekdays = filter(dy -> dayname(dy) != "Saturday" && dayname(dy) != "Sunday" ,
 d)
```

Similarly, here's a range of times 3 hours apart from now, for a year hence:

```
julia> d = [DateTime(now()):Hour(3):DateTime(now() + Dates.Year(1))]
2921-element Array{DateTime,1}:
 2014-09-02T17:45:19
 2014-09-02T20:45:19
 2014-09-02T23:45:19
 2014-09-03T02:45:19
 2014-09-03T05:45:19
 2014-09-03T08:45:19
 2014-09-03T11:45:19
 2014-09-03T14:45:19
 2014-09-03T17:45:19
 2014-09-03T20:45:19
 2014-09-03T23:45:19
 2014-09-04T02:45:19
 ⋮
 2015-08-31T23:45:19
 2015-09-01T02:45:19
 2015-09-01T05:45:19
 2015-09-01T08:45:19
 2015-09-01T11:45:19
 2015-09-01T14:45:19
 2015-09-01T17:45:19
 2015-09-01T20:45:19
 2015-09-01T23:45:19
 2015-09-02T02:45:19
 2015-09-02T05:45:19
 2015-09-02T08:45:19
 2015-09-02T11:45:19
 2015-09-02T14:45:19
 2015-09-02T17:45:19
```

10.1.5 Date formatting

To output a date in another format, use the `Dates.format()` function.

```
julia> Dates.format(DateTime(2014), DateFormat("e, dd u yyyy HH:MM:SS"))
"Wed, 01 Jan 2014 00:00:00"
```

The letters you supply in the format string determine which elements are output.

10.1.6 Date adjustments

Sometimes you want to find a date nearest to another - for example, the first day of that week, or the last day of the month that contains that date. You can do this with the functions like `Dates.firstdayofweek()` and `Dates.lastdayofmonth()` . So, if we're currently in the middle of the week:

```
julia> dayname(now())
"Wednesday"
```

the first day of the week is returned by this:

```
julia> Dates.firstdayofweek(now())
2014-09-01T00:00:00
```

A more general solution is provided by the `tofirst()` , `tolast()` , `tonext()` , and `toprev()` methods.

With `tonext()` and `toprev()` , you can provide a (possibly anonymous) function that returns true when a date has been correctly adjusted. For example, the function:

```
d->Dates.dayofweek(d) == Dates.Tuesday
```

returns true if the day `d` is a Tuesday. Use this with the `tonext()` method:

```
julia> Dates.tonext(d->Dates.dayofweek(d) == Dates.Tuesday, birthday)
1997-03-18
```

Or you can find the next Sunday following the birthday date:

```
julia> Dates.tonext(d->Dates.dayname(d) == "Sunday", birthday)
1997-03-16
```

— it was a Sunday the day after the 15th.

With `tofirst()` and `tolast()` , you supply the date, and the first day of that month is returned. Supply the keyword argument `of=Year` to get the first day of that year.

```
julia> Dates.tofirst(birthday, 1)
1997-03-03
julia> Dates.tofirst(birthday, 1, of=Year)
1997-01-06
```

10.1.7 Recurring dates

It's useful to be able to find all dates in a range of dates that satisfy some particular criteria. For example, you can work out the second Sunday in a month by using the `Dates.dayofweekofmonth()` and `Dates.dayname()` functions. If you supply these and a range of dates, the `recur()` function can find recurring dates.

For example, let's create a range of dates from the first of September 2014 until Christmas Day, 2014:

```
julia> dr = Dates.Date(2014,9,1):Dates.Date(2014,12,25)
2014-09-01:1 day:2014-12-25
```

Now an anonymous function similar to the ones we used in `tonext()` earlier finds a selection of those dates that satisfy the function:

```
julia> recur(d->Dates.dayname(d) == "Sunday",dr)
15-element Array{Date,1}:
 2014-09-14
 2014-09-21
 2014-09-28
 2014-10-05
 2014-10-12
 2014-10-19
 2014-10-26
 2014-11-02
 2014-11-09
 2014-11-16
 2014-11-23
 2014-11-30
 2014-12-07
 2014-12-14
 2014-12-21
```

These are the dates of every Sunday between now and Christmas.

By combining criteria in the anonymous function, you can build up more complicated recurring events. Here's a list of all the Tuesdays in the period which are on days that are prime numbers:

```
julia> recur(d->Dates.dayname(d) == "Tuesday" && isprime(Dates.day(d)), dr)
6-element Array{Date,1}:
 2014-09-02
 2014-09-23
 2014-10-07
 2014-11-11
 2014-12-02
 2014-12-23
```

10.1.8 Unix time

You might have to deal with another type of timekeeping: Unix time. Unix time is a count of the number of seconds that have elapsed since the beginning of the year 1970 (the Unix epoch). In Julia the count is stored in a 64 bit integer, and we'll never see the end of Unix time. (The universe will have ended long before 64 bit Unix time reaches the maximum possible value, in approximately 292 billion years from now, at 15:30:08 on Sunday, 4 December 292,277,026,596.)

In Julia, the `time()` function, used without arguments, returns the Unix time value of the current second:

```
julia> time()
1.414141581230945e9
```

The `strftime()` ("string format time") function converts a number of seconds in Unix time to a more readable form:

```
julia> strftime(86400 * 365.25 * 4)
"Tue  1 Jan 00:00:00 1974"
```

You can choose a different format by supplying a format string, with the different components of the date and time defined by '%' letter codes:

```
julia> strftime("%A, %B %e at %T, %Y", 86400 * 365.25 * 4)
"Tuesday, January  1 at 00:00:00, 1974"
```

The `strptime()` function takes a format string and a date string, and returns a TmStruct expression. This can then be converted to a Unix time value by passing it to `time()` :

```
julia> strptime("%A, %B %e at %T, %Y", "Tuesday, January  1 at 00:00:00, 1974")
TmStruct(0,0,0,1,0,74,2,0,0,0,0,0,0,0)
julia> time(ans)
1.262304e8
julia> time(strptime("%Y-%m-%d","2014-10-1"))
1.4121216e9
```

Julia also offers a `unix2datetime()` function, which converts a Unix time value to a date/time object:

```
julia> unix2datetime(time())
2014-10-24T09:26:29.305
```

10.2 Timing and monitoring

If you want to tell how long functions take to run, you can use `tic()` and `toc()` , which provide the start and finish buttons of a virtual stopwatch:

```
function test(n)
    tic()
    for i in 1:n
       x = sin(rand())
    end
    toc()
end

julia> test(100000000)
elapsed time: 1.313614113 seconds
1.313614113
```

`toc()` prints the time since the most recent `tic()` and returns the value as well.

The `@elapsed` macro returns the number of seconds an expression took to evaluate:

```
function test(n)
     for i in 1:n
        x = sin(rand())
     end
end

julia> @elapsed test(100000000)
1.309819509
```

The `@time` macro tells you how long an expression took to evaluate, and how memory was allocated.

```
julia> @time test(100000000)
elapsed time: 1.308975149 seconds (75644 bytes allocated)
```

11 Plotting

11.1 Plotting

There are a number of different packages for plotting in Julia. In most cases, adding a plotting package to Julia is going to involve adding and installing additional graphics libraries and components. Sometimes this goes well, sometimes it doesn't...

11.1.1 ASCIIPlots

If you want a quick plot and can't wait while the bigger plotting packages (and their associated graphic paraphernalia) load, use ASCIIPlots. Your graphs will be constructed out of pure ASCII art, as if you'd traveled back in time to the days of monochrome terminals.

You can plot using ASCII characters! You have a choice of three functions: `scatterplot()` , `lineplot()` , and `imagesc()` .

First, add the package to your Julia installation. You have to do this just once:

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

Now load the module and import the functions:

```
julia> using ASCIIPlots
```

Here's `lineplot()` . You supply a series of x-values, followed by a series of y-values:

```
julia> x = [-pi:0.2:pi];
julia> lineplot(x, sin(x))

    ------------------------------------------------------------
    |                                             \              | 1.00
    |                                         / -   \            |
    |                                       /         \          |
    |                                      /            \        |
    |                                    /                \      |
    |                                                            |
    |                                  /                    \    |
    |                                                         \  |
    |                                /                           |
    |                              /                            \|
    |\                                                           |
    |                            /                               |
    | \                                                          |
    |                          /                                 |
    |   \                                                        |
    |     \                  /                                   |
    |                      /                                     |
    |       \                                                    |
    |         \          /                                       |
    |           \ - - / /                                        | -1.00
    ------------------------------------------------------------
    -3.14                                                       3.06
```

11.1.2 PyPlot

If you're already familiar with MatLab, or Python's plotting environment, you should be happy with PyPlot[1].

First, add the package to your Julia installation. You have to do this just once:

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

A simple plot can be produced with:

```
julia> using PyPlot
julia> x = [-pi:0.2:pi];
julia> plot(x, sin(x), ".")
1-element Array{Any,1}:
 PyObject <matplotlib.lines.Line2D object at 0x114734940>
```

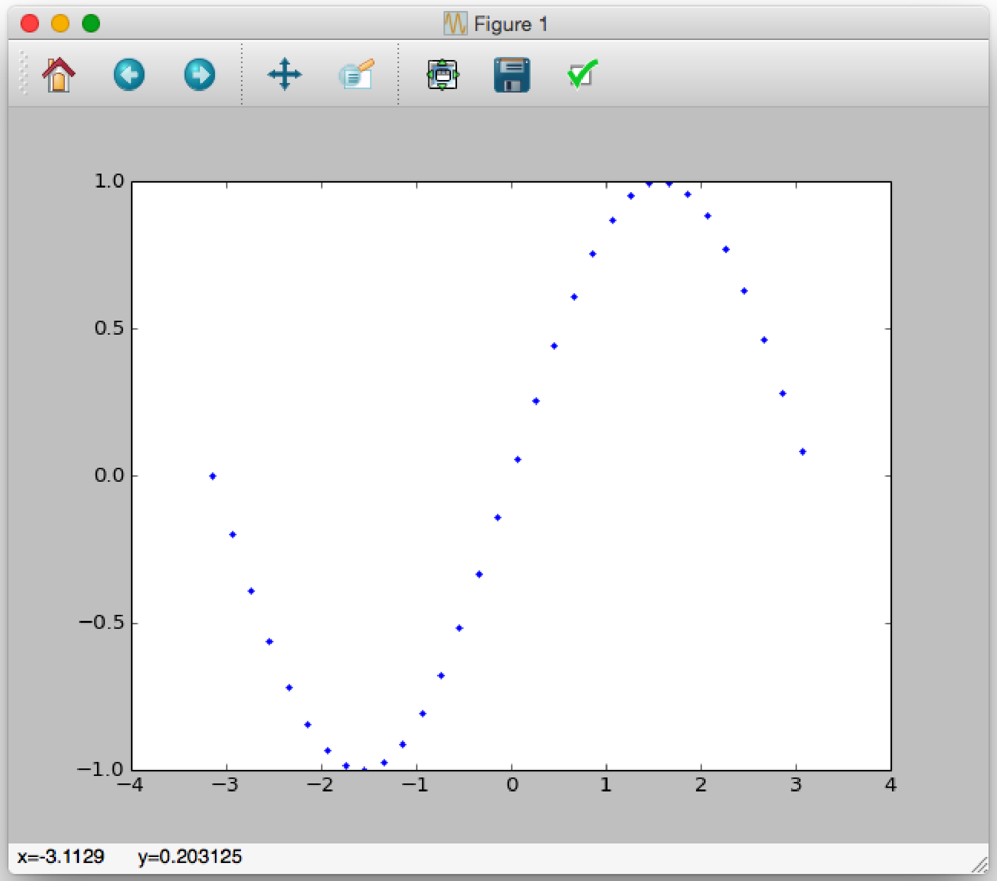

Figure 2 plot produced by PyPlot

1 http://pyplotjl.readthedocs.org/en/latest/

11.1.3 Winston

Winston (perhaps named for Julia's lover in George Orwell's 1984?) offers "an easy to use plot command to create figures without any fuss":

```
julia> Pkg.add("Winston")
julia> using Winston
```

For a simple plot, supply Winston's `plot()` function with an array:

```
julia> plot([sin(a) for a in 0.0:0.1:2 * pi])
FramedPlot(...)
julia> savefig("figure.png")
```

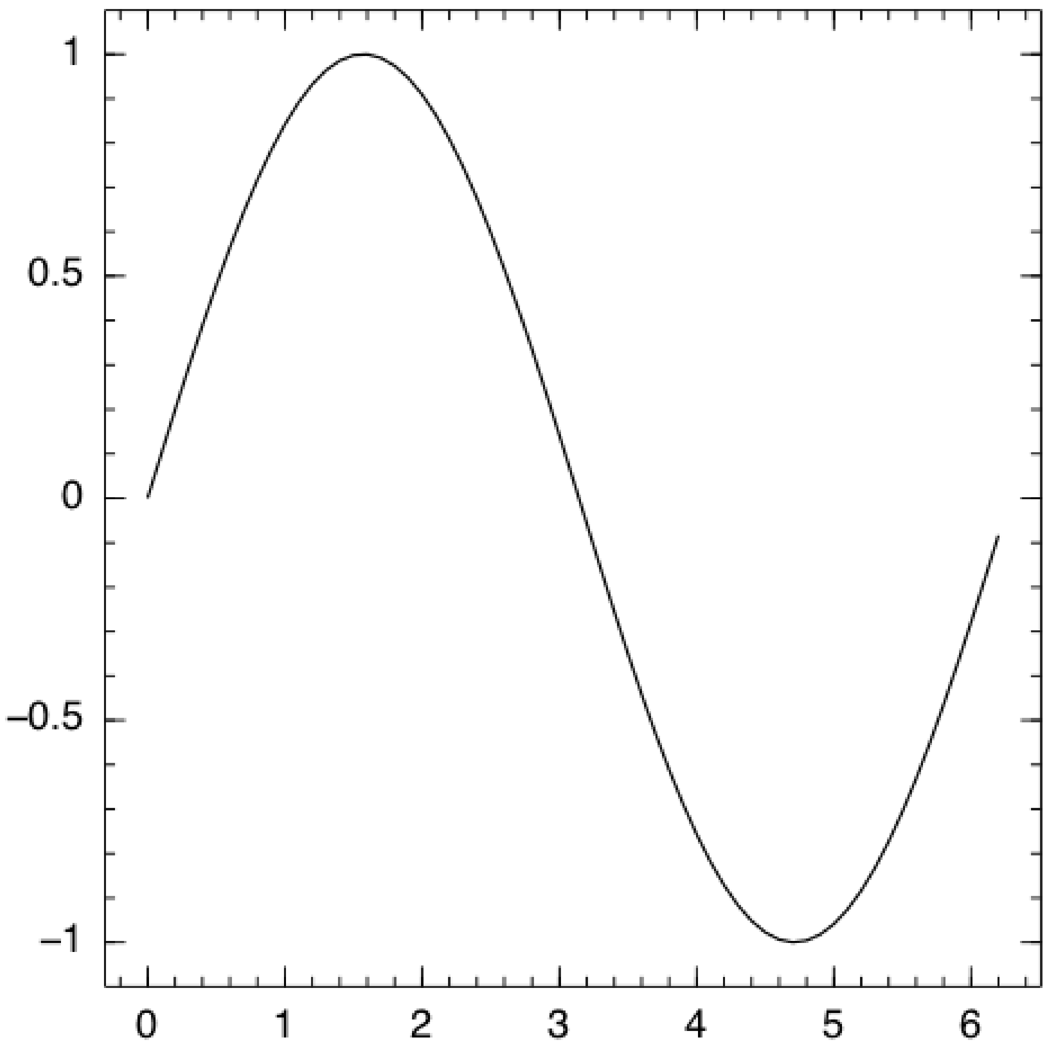

Figure 3 simple plot made by Winston (Julia plotting package)

11.1.4 Gadfly

And then there's Gadfly, which is possibly the most interesting of the plot libraries. First, add the package to your Julia installation. You have to do this just once:

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

Now's a good time to go for a coffee...

OK. You first want to load the Gadfly package:

```
julia> using Gadfly
```

and this can take a few seconds.

Most of your plotting work is going to involve using the `plot()` function. This has many different forms — it's ”overloaded” so that it can plot the data in:

- functions
- arrays
- dataframes (provided by the DataFrames package)

The basic form of a plot instruction is:

```
plot(data, mappings, element1, element2, …)
```

where 'data' is the function, array, or other data source, the 'mappings' are the symbol and assignments that link to the data, and the 'elements' are the various options such as themes, scales, plot markers, labels, titles, and guides that you'll want to tweak and adjust to make the plot look amazing.

```
julia> p = plot(x=randn(2000), Geom.histogram(bincount=100))
```

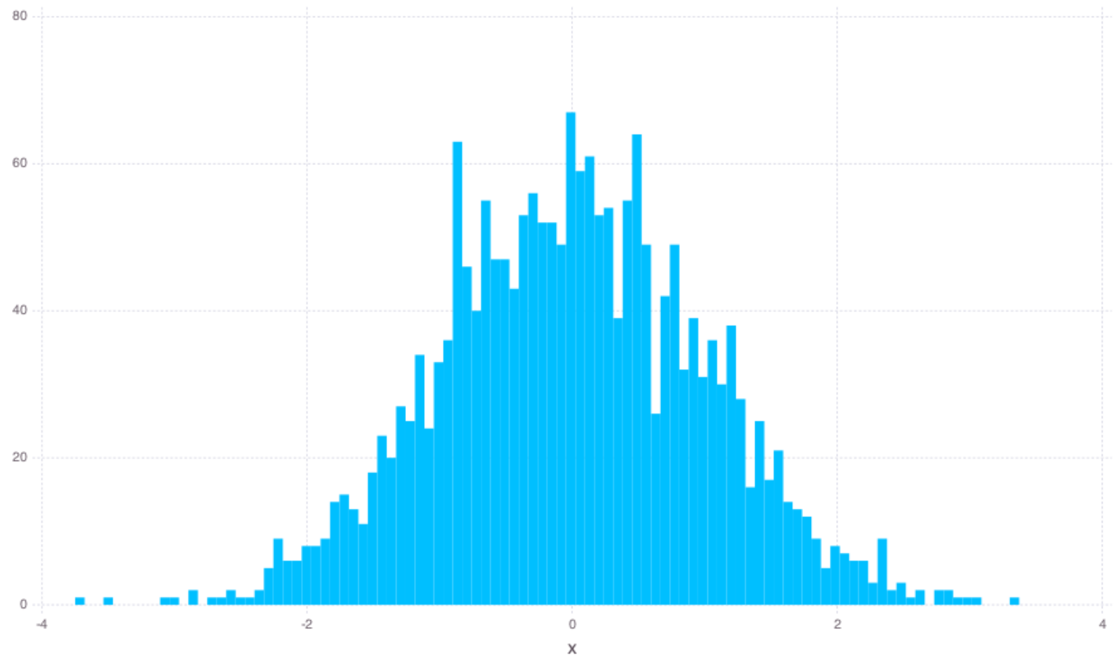

Figure 4 histogram plot created in Julia using Gadfly

11.2 Example of plotting a simple graph in Gadfly

Here's an example of plotting a simple graph using Gadfly. In this example, the aim is to plot the varying value of the equation of time[2] during a year. The data to be plotted is an array of 365 real numbers ranging between -20 and 20 minutes. The values can be calculated with the following code (which uses the Astro.jl[3] library).

```
using Astro, Dates
days = [DateTime(2015,1,1, 0,0,0):DateTime(2015,12,31,0,0,0)]; # an array of
365 datetimes
eq_values = Array(Float64, 0);
for day in days
   push!(eq_values, equation_time(Dates.datetime2julian(day)))
end
```

The array `eq_values` now holds the 365 numbers to be plotted:

```
length(eq_values)
365
```

To produce a very basic plot, assign the varying values in the `eq_values` array to the y 'aesthetic', and use a simple range object (`1:365`) for the x 'aesthetic'. The default geometry is a point:

```
plot1 = plot(
   x = 1:length(days), # the x-axis from 1 to 365
   y = eq_values       # the y-axis for the values of the equation of time
   )
```

2 https://en.wikipedia.org/wiki/Equation_of_time
3 https://github.com/cormullion/Astro.jl

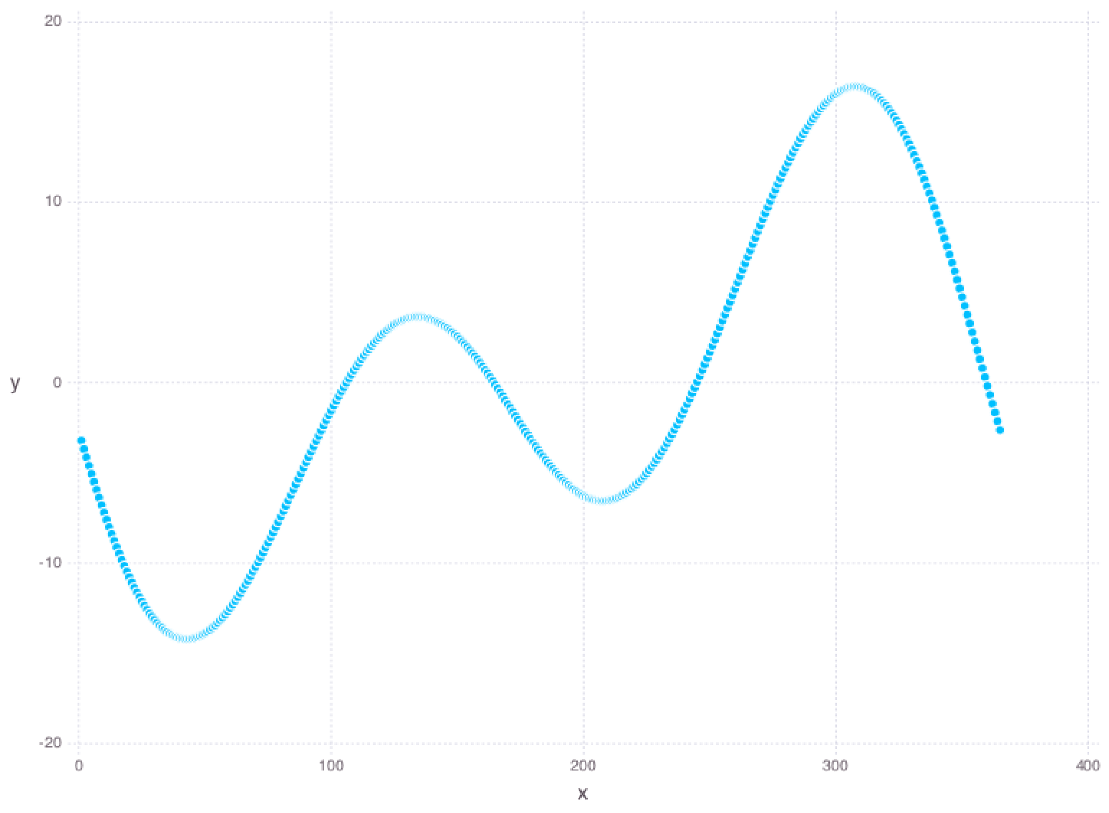

Figure 5 graph of equation of time created in Gadfly (Julia)

You don’t have to store the plot in a variable, but it can be useful later on.

The first obvious addition is to add explanatory text labels and a title to the graph. In Gadfly-speak, these are called ”guide” objects:

```
plot2 = plot(
   x=1:length(days),
   y=eq_values,
   Guide.ylabel("Minutes faster or slower than GMT"), # label for y-axis
   Guide.xlabel("day in year"),  # label for x-axis
   Guide.title("The Equation of Time")  # a title
)
```

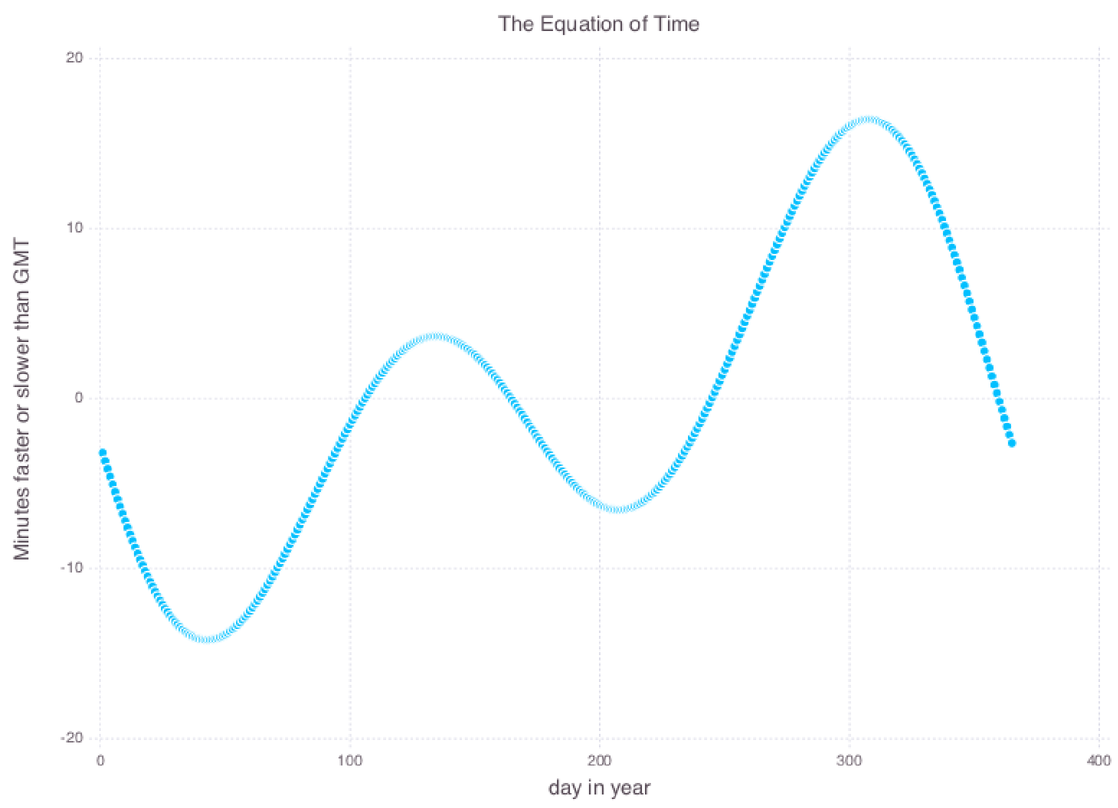

Figure 6 graph of equation of time created in Gadfly (Julia)

One problem with this plot is that the ticks and labels running along x-axis aren't very informative, so we'll convert the list of 365 DateTime objects to strings:

```
datestrings = Dates.format(days, DateFormat("u dd"))
365-element Array{Any,1}:
"Jan 01"
"Jan 02"
"Jan 03"
"Jan 04"
"Jan 05"
⋮
"Dec 28"
"Dec 29"
"Dec 30"
"Dec 31"
```

Then we can use these for the x aesthetic, replacing the simple range from 1 to 365. We've switched from `Geom.point` to `Geom.line` . And we're adding two new property settings: ticks, and a theme.

The Guide.xticks and Guide.yticks settings control the tick marks and labels for the axes. The ticks can be specified with an array.

There are far too many strings to draw each one, so a step size of 14 is useful for `ticks` — every 14 days there's a tick and a label.

The Theme settings determine the visual styling of the plot.

```
plot3 = plot(
   x=datestrings,                                    # use dates for x
   y=eq_values,                                      # equation values
   Guide.xticks(
      ticks=[1:14:365],                              # show 1 in 14 ticks
      orientation=:vertical                          # rotate to vertical
      ),
   Guide.yticks(
      ticks=[-20:5:20]                               # show labels and ticks
for every 5 seconds
      ),
   Guide.ylabel("Minutes faster or slower than GMT"), # label for y-axis
   Guide.xlabel("day in year"),                      # label for x-axis
   Guide.title("The Equation of Time"),              # title
   Geom.line,                                        # line rather than point
   Theme(
     default_color=color("#FF0022"),                 # color of plot line
     default_point_size=3pt,                         # text size
     panel_fill=color("#00FF00"),                    # background color of
plot
     panel_stroke=color("Blue"),                     # edge of plot
     line_width=2px                                  # width of line
     )
   )
```

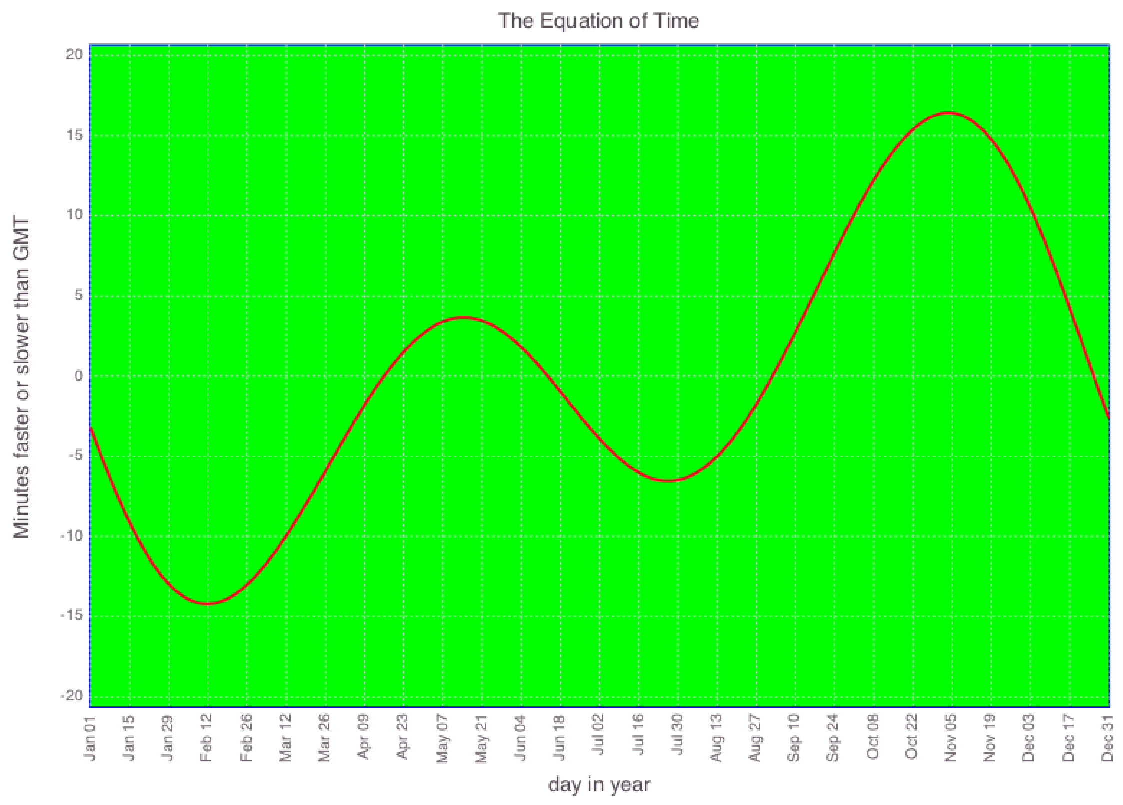

Figure 7 graph of equation of time created in Gadfly (Julia)

Another way of avoiding the mess that you get if you try to plot too many items is to use stepped ranges to 'sample' the x and y data aesthetics.

```
plot4 = plot(
 x=datestrings[1:7:end],                                       # step through
every seventh date
 y=eq_values[1:7:end],                                         # step through
every seventh value
 Guide.ylabel("Minutes faster or slower than GMT"),
 Guide.xlabel("day in year"),
 Guide.xticks(orientation=:vertical),
 Geom.hline(size=0.25mm, color="darkgray"), yintercept=[5,8],  # thin grey
horizontal lines at y = 5 and y = 8
 Geom.point,
 Theme(
     default_point_size=1mm,
     minor_label_font_size=6pt),
)
```

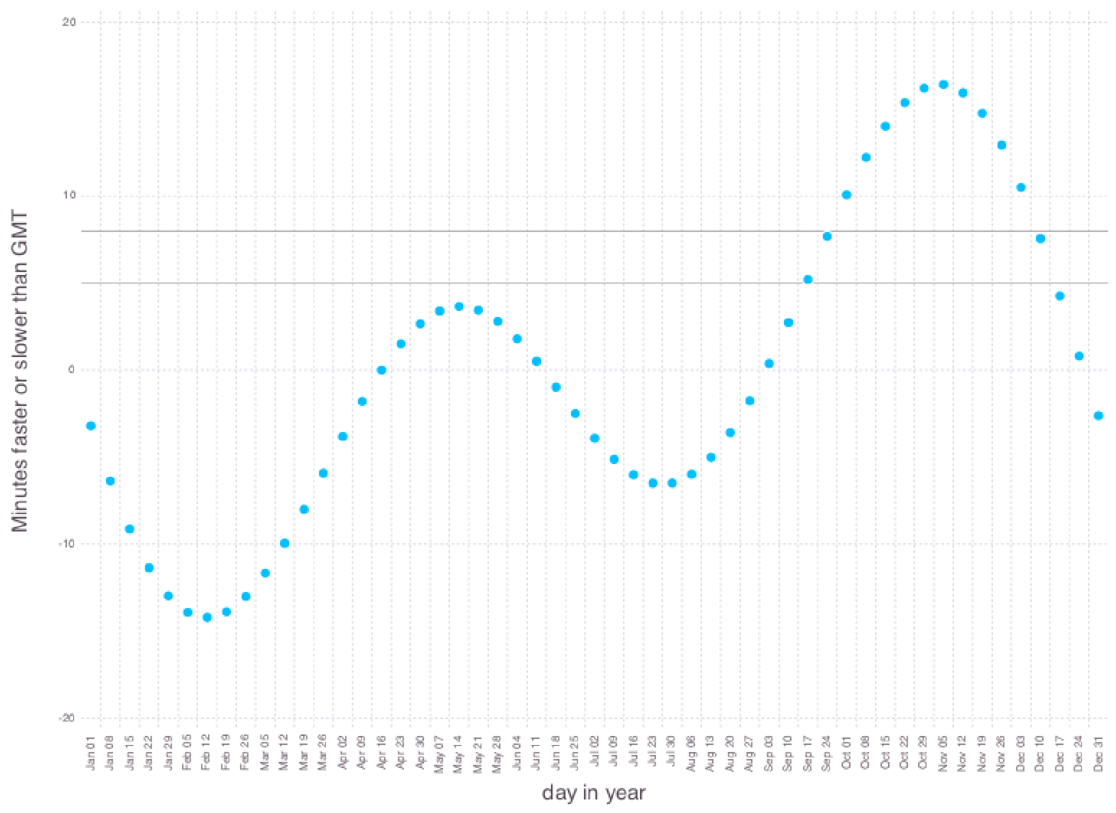

Figure 8 graph of equation of time created in Gadfly (Julia)

As an alternative to placing pre-made date strings along the x-axis, we can make use of the `Scale` element. This is how to transform the basic data to make labels or re-scaled values. For `x_discrete` , you can supply a function which takes a numerical value and generates a string that defines the label. To avoid plotting every label, this anonymous function provides the date string label only every 14 times - i.e. every fortnight, producing an empty string otherwise:

```
plot5 = plot(
   x=1:length(eq_values),
   y=eq_values,
   Theme(default_point_size=2pt,
         minor_label_font_size=6pt,
         grid_line_width=0.1pt),
   Guide.ylabel("Minutes faster or slower than GMT"),
   Guide.xlabel("day in year"),
   Geom.point,
   Guide.xticks(orientation=:vertical),
   Scale.x_discrete(
         labels=n -> n%14 == 0 ? Dates.format(days[n], DateFormat("e, dd u")) :
"")
)
```

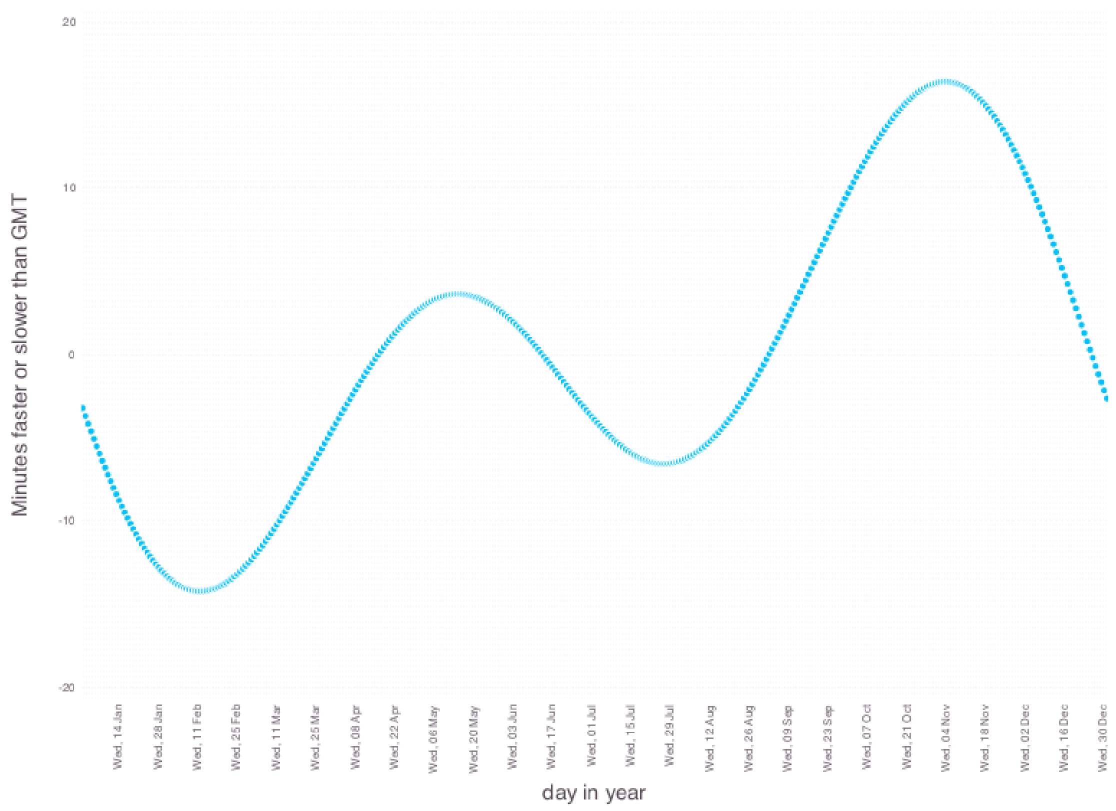

Figure 9 graph of equation of time created in Gadfly (Julia)

11.2.1 Combining plots

A simplified model of the equation of time is given by the following formula:

```
equation(d) = -7.65 * sind(d) + 9.87 * sind((2 * d) + 206)
```

which is plotted easily from 1 to 365 using another Gadfly plot method:

```
plot6 = plot(equation, 1, 365)
```

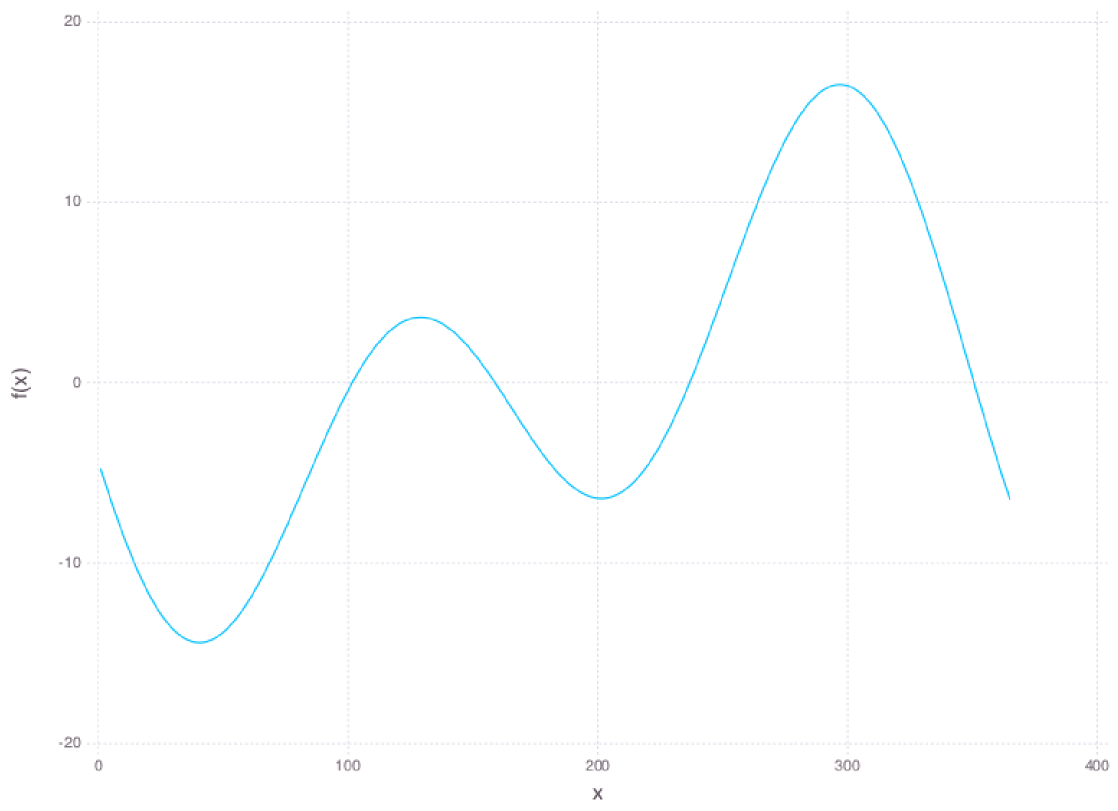

Figure 10 graph of equation of time created in Gadfly (Julia)

You can stack these plots one above the other, using `vstack()` :

```
vstack(plot5,plot6)
```

(There's an `hstack()` as well.)

To superimpose the two graphs, we can make use of `layers` . Here, the first layer holds the point/line plot, the second layer draws the equation.

```
plot7 = plot(
 layer(
   # first layer
   x=1:365,
   y=eq_values,
   Theme(
     default_point_size=2pt,
     minor_label_font_size=6pt,
     default_color=color("orange")),
   Geom.point),
 layer(
  # second layer
   equation, 1, 365),
 Theme(
 # theme for all layers
   default_point_size=2pt,
   major_label_font_size=12pt,
   minor_label_font_size=16pt,
   default_color=color("purple")),
 Guide.ylabel("Minutes faster or slower than GMT"),
 Guide.xlabel("day in year"),
 Guide.xticks(orientation=:vertical)
)
```

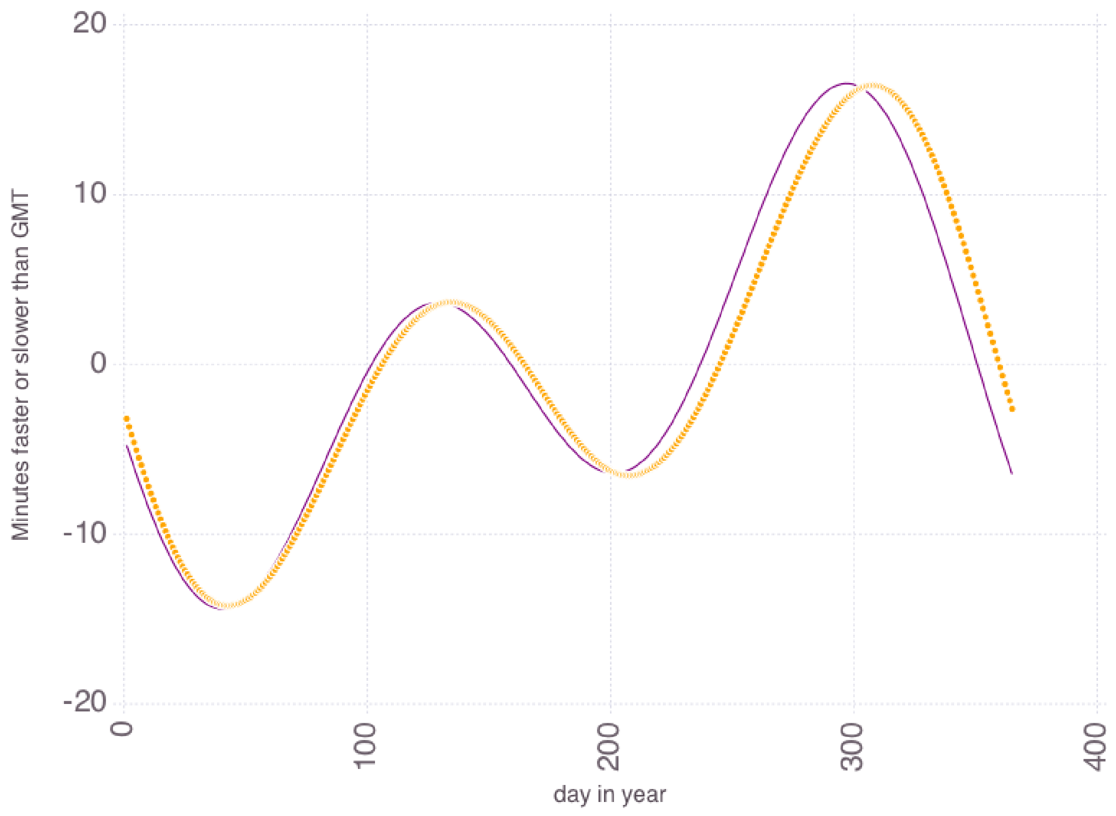

Figure 11 graph of equation of time created in Gadfly (Julia)

Each layer can be styled differently, using the Theme setting.

```
plot8 = plot(
 # first layer
 layer(
   x=datestrings[1:7:end],
   y=eq_values[1:7:end],
   Geom.line,
   Theme(
      line_width=1pt,
      default_color=color("orange")
   )
 ),
 # second layer
 layer(
   x=datestrings[1:7:end],
   y=eq_values[1:7:end],
   Geom.point,
   Theme(
      default_point_size=1mm,
      default_color=color("green"))
 ),
  # background
  Theme(
     panel_fill=color("#FFFF00"),
     panel_opacity=0.1,
     panel_stroke=color("Blue"),
   )
)
```

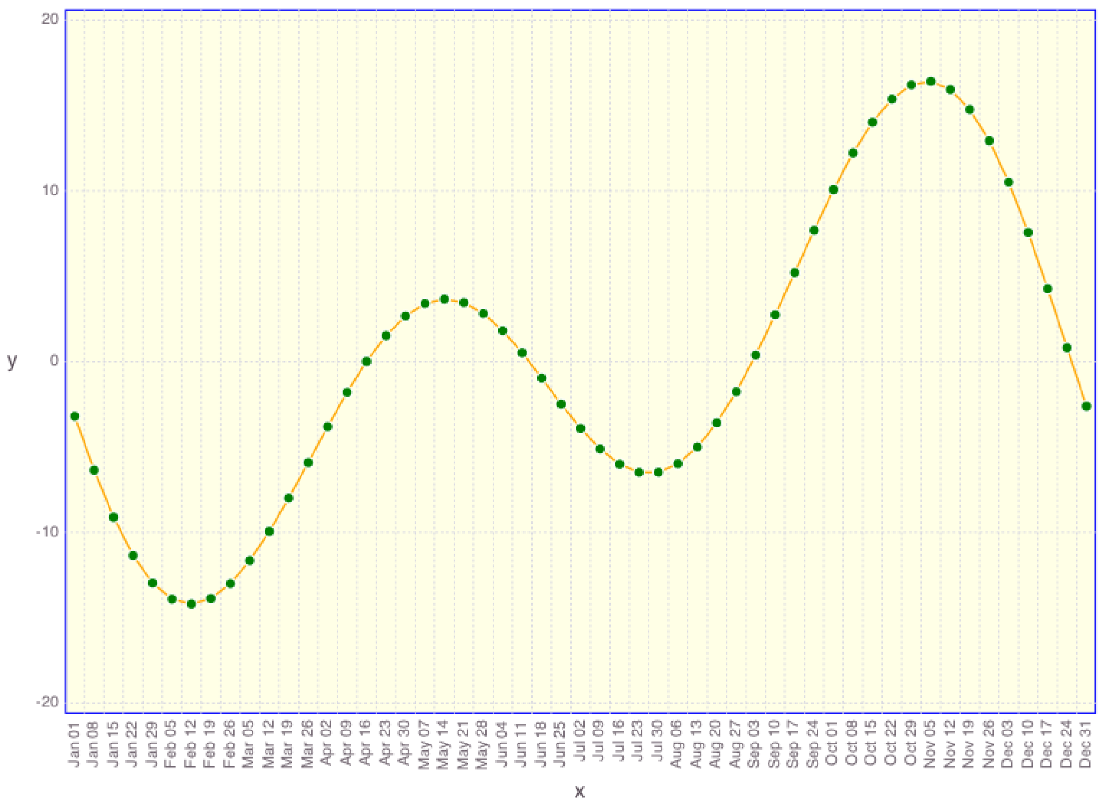

Figure 12 graph of equation of time created in Gadfly (Julia)

You can use some theme settings inside layers, and others for styling the whole plot.

11.2.2 Saving plots

To save a plot to a file, use Gadfly's `draw()` function. Because we named each plot, saving them all to automatically-named files is easy.

```
for i in 1:8
   println("Saving plot $i")
   draw(PDF(join(["equation-graph-", string(i), ".pdf"]), 16cm, 12cm),
eval(symbol("plot" * string(i))))
end
```

12 Metaprogramming

12.1 Metaprogramming

Meta-programming is when you write Julia code to process and modify Julia code. With the meta-programming tools, you can write Julia code that modifies other parts of your source files, and even control if and when the modified code runs.

In Julia, the execution of code takes place in two stages. Stage one is when your raw Julia code is parsed — converted into a form that is suitable for evaluation. (You'll be familiar with this phase, because this is when all your syntax mistakes are noticed...) Stage 2 is when that parsed code is executed. Usually, when you type code into the REPL and press Return, or when you run a Julia file from the command line, you don't notice the two stages, because they happen so quickly. However, with Julia's metaprogramming facilities, you can access the code **after** it's been parsed but **before** it's evaluated. This lets you do things that you can't normally do. For example, you can convert simple expressions to more complicated expressions, or examine code before it runs and change it so that it runs faster. Any code that you intercept and modify using these meta-programming tools will eventually be evaluated in the usual way, running as fast as ordinary Julia code.

Two existing examples of meta-programming in Julia are:

- the `@time` macro:

```
julia> @time [sin(cos(i)) for i in 1:100000];
elapsed time: 0.00721026 seconds (800048 bytes allocated)
```

The `@time` macro inserts a "start the stopwatch" command at the beginning of the code, before passing the expression on to be evaluated. When the code has finished running, a "finish the stopwatch" command is added, followed by the calculations to report the elapsed time and memory usage.

- the `@which` macro

```
julia> @which 2 + 2
+(x::Int64,y::Int64) at int.jl:33
```

This macro doesn't allow the expression `2 + 2` to be evaluated at all. Instead, it reports which method would be used for these particular arguments. And it also tells you the source file that contains the method's definition, and the line number.

Other uses for meta-programming include the automation of tedious coding jobs by writing short pieces of code that produce larger chunks of code, and the ability to improve the performance of 'standard' code by producing the sort of faster code that perhaps you wouldn't want to write by hand.

12.1.1 Quoted expressions

For meta-programming to work, there has to be a way to stop Julia evaluating expressions as soon as the parsing phase has finished. This is the ':' (colon) prefix operator:

```
julia> x = 3
3
julia> :x
:x
```

To Julia, the `:x` is an unevaluated or quoted symbol.

(If you're unfamiliar with the use of quoted symbols in computer programming, think of how quotes are sometimes used in writing to distinguish between ordinary use and special use. For example, in the sentence:

`'Copper' contains six letters.`

the quotes indicate that the word 'Copper' is not a reference to the metal, but to the word itself. In the same way, in `:x` , the colon before the symbol is to make you and Julia think of 'x' as an unevaluated symbol rather than as the value 3.)

To quote whole expressions rather than individual symbols, start with a colon and then enclose the Julia expression in parentheses:

```
julia> :(2 + 2)

:(2 + 2)
```

There's an alternative form of the `:( )` construction that uses the `quote` ... `end` keywords to enclose and quote an expression:

```
julia>expression =
quote
    for i = 1:10 # line 2:
        println(i)
    end
end
```

This object `expression` is of type Expr:

```
julia> typeof(expression)
Expr
```

It's parsed, primed, and ready to go.

12.1.2 Evaluating expressions

There's also a function for evaluating an unevaluated expression. It's called `eval()` :

```
julia> eval(:x)
3
julia> eval(:(2 + 2))
4
julia> eval(expression)
1
2
```

```
3
4
5
6
7
8
9
10
```

With these tools, it's possible to create any expression and store it without having it evaluate:

```
julia> e =
:(
    for i in 1:10
        println(i)
    end
)

:(for i = 1:10 # line 2:
    println(i)
end)
```

and then to recall and evaluate it later:

```
julia> eval(e)
1
2
3
4
5
6
7
8
9
10
```

It's also possible to modify the contents of the expression before it's evaluated.

12.1.3 Inside Expressions

Once you have Julia code in an unevaluated expression, rather than as a piece of text in a string, you can do things with it.

Here's another expression:

```
julia> e =
quote
     a = 2
     b = 3
     c = 4
     d = 5
     e = sum([a,b,c,d])
end

quote  # none, line 2:
    a = 2 # line 3:
    b = 3 # line 4:
    c = 4 # line 5:
    d = 5 # line 6:
    e = sum([a,b,c,d])
```

```
end
```

Notice the helpful line numbers that have been added to each line of the quoted expression.

We can use the `names()` function to see what's inside this expression:

```
julia> names(e)
3-element Array{Symbol,1}:
 :head
 :args
 :typ
```

The `head` field is `Block` . The args field is another array, containing expressions (including comments). We can examine these using ordinary Julia code:

```
julia> e.args[2]
:(a = 2)

julia> for (n,expr) in enumerate(e.args)
             println(n, ": ", expr)
          end
1:  # none, line 3:
2: a = 2
3:  # line 4:
4: b = 3
5:  # line 5:
6: c = 4
7:  # line 6:
8: d = 5
9:  # line 7:
10: e = sum([a,b,c,d])
```

As you can see, the expression `e` contains a number of sub-expressions. You can modify this expression quite easily, for example by changing the last line of the expression to use `prod()` rather than `sum()` :

```
julia> e.args[end] = quote prod([a,b,c,d]) end
quote  # none, line 1:
    prod([a,b,c,d])
end

julia> eval(e)
120
```

12.1.4 Expression interpolation

In a way, strings and expressions are similar — any Julia code they happen to contain is usually unevaluated. We've met the string interpolation operator, the dollar sign ($). If used inside a string, and possibly with parentheses to enclose the expression, this evaluates the Julia code and inserts the resulting value into the string at that point:

```
julia> "the sine of 1 is $(sin(1))"
"the sine of 1 is 0.8414709848078965"
```

In just the same way, you can use the dollar sign to include the results of executing Julia code into an expression (which is otherwise unevaluated):

```
julia> quote s = $(sin(1) + cos(1))
       end
```

```
quote  # none, line 1:
    s = 1.3817732906760363
end
```

Even though this is a quoted expression, the value of `sin(1) + cos(1)` was calculated and inserted into the expression, replacing the original code. This operation is called ”splicing”.

As with string interpolation, the parentheses are needed only if you want to include the value of an expression — a single symbol can be interpolated using just a single dollar sign.

12.1.5 Macros

Once you know how to create and handle unevaluated Julia expressions, you'll want to know how you can modify them. A `macro` is a way of generating a new output expression, given an unevaluated input expression. When your Julia program runs, it first parses and evaluates the macro, and the processed code produced by the macro is eventually evaluated like an ordinary expression.

Here's the definition of a simple macro that prints out the contents of the thing you pass to it, and then evaluates it. The syntax is very similar to the way you define functions:

```
macro p(n)
    if typeof(n) == Expr
       println(n.args)
    end
    eval(n)
end
```

You run macros by preceding the name with the `@` prefix. This macro is expecting a single argument. You're providing unevaluated Julia code, you don't have to enclose it with parentheses, like you do for function arguments.

First, let's call this with a single numeric argument:

```
julia> @p 3
3
```

Numbers aren't expressions, so the `if` condition inside the macro didn't apply. All the macro did was evaluate `n` and return the result. But if you pass an expression, the code in the macro has the opportunity to inspect and/or process the expression's content before it is evaluated, using the `.args` field:

```
julia> @p 3 + 4 - 5 * 6 / 7 % 8
{:-,:(3 + 4),:(((5 * 6) / 7) % 8)}
2.7142857142857144
```

In this case, the `if` condition was triggered, and the arguments of the incoming expression were printed in unevaluated form. So you can see the arguments as an array of expressions after being parsed by Julia but before being evaluated. You can also see how the different precedence of arithmetic operators has been taken into account in the parsing operation. Notice that the operator symbols are quoted with a colon (`:`), as are the subexpressions.

In this example, the macro `p` evaluated the code. It doesn't have to — it could return a quoted expression instead. For example, the built-in `@time` macro returns a quoted and —

as yet — unevaluated expression, rather than using `eval()` to evaluate the expression inside the macro. The quoted expression returned by `@time` is evaluated in the calling context.

Here's the definition:

```
macro time(ex)
           quote
               local t0 = time()
               local val = $(esc(ex))
               local t1 = time()
               println("elapsed time: ", t1-t0, " seconds")
               val
           end
       end
```

Notice the `$(esc(ex))` expression. This is the way that you 'escape' the code you want to time, which is in `ex` , so that it isn't evaluated in the macro, but left intact until the entire quoted expression is returned to the calling context and executed there. If this just said `$ex` , then the expression would be interpolated and evaluated immediately.

If you want to pass a multi-line expression to a macro, use the `begin` ... `end` form:

```
julia> @p begin
       2 + 2 - 3
       end
{:( # none, line 2:),:((2 + 2) - 3)}
1
```

(You can also call macros with parentheses similar to the way you do when calling functions, using the parentheses to enclose the arguments:

```
julia> @p(2 + 3 + 4 - 5)
{:-,:(2 + 3 + 4),5}
4
```

This would allow you to define macros that accepted more than one expression as arguments.)

Scope and context

When you use macros, you have to keep an eye out for scoping issues. In the previous example, the `$(esc(ex))` syntax was used to prevent the expression from being evaluated in the wrong context. Here's another contrived example to illustrate this point.

```
macro f(x)
       quote
           s = 4
           (s, $(esc(s)))
       end
end
```

This macro declares a variable `s` , and returns a quoted expression containing `s` and an escaped version of `s` .

Now, outside the macro, declare a symbol `s` :

```
julia> s = 0
```

Run the macro:

```
julia> @f 2
(4,0)
```

You can see that the macro returned different values for the symbol `s` : the first was the value inside the macro's context, 4, the second was an escaped version of `s` , that was evaluated in the calling context, where `s` has the value 0. In a sense, `esc()` has protected the value of `s` as it passes unharmed through the macro. For the more realistic @time example, it's important that the expression you want to time isn't modified in any way by the macro.

12.1.6 Expanding macros

To see what the macro expands to just before it's finally executed, use the `macroexpand()` function. It expects a quoted expression containing one or more macro calls, which are then expanded into proper Julia code for you so that you can see what the macro would do when called.

```
 macroexpand(quote @p 3 + 4 - 5 * 6 / 7 % 8 end)

{:-,:(3 + 4),:(((5 * 6) / 7) % 8)}
begin  # none, line 1:
    2.7142857142857144
end
```

(The `#none, line 1:` is a filename and line number reference that's more useful when used inside a source file than when you're using the REPL.)

Here's another example. This macro adds a `dotimes` construction to the language.

```
macro dotimes(n, body)
          quote
              for i = 1:$(esc(n))
                  $(esc(body))
              end
          end
      end
```

This is used as follows:

```
@dotimes 3 println("hi there")
hi there
hi there
hi there
```

Or, less likely, like this:

```
 @dotimes 3 begin
           for i in 4:6
               println("i is $i")
           end
       end

i is 4
i is 5
i is 6
i is 4
```

```
i is 5
i is 6
i is 4
i is 5
i is 6
```

If you use `macroexpand()` on this, you can see what happens to the symbol names:

```
macroexpand( quote  @dotimes 3 begin
           for i in 4:6
               println("i is $i")
           end
       end
end )
```

with the following output:

```
quote  # none, line 1:
    begin  # none, line 3:
        for #378#i = 1:3 # line 4:
            begin  # none, line 2:
                for i = 4:6 # line 3:
                    println("i is $i")
                end
            end
        end
    end
end
```

The `i` local to the macro itself has been renamed to `#378#i` , so as not to clash with the original `i` in the code I've passed to it.

13 Contributors

Edits	User
305	Cormullion[1]
1	Dirk Hünniger[2]

1 https://en.wikibooks.org/wiki/User:Cormullion
2 https://en.wikibooks.org/wiki/User:Dirk_H%25C3%25BCnniger

List of Figures

- EPL: Eclipse Public License. `http://www.eclipse.org/org/documents/epl-v10.php`

Copies of the GPL, the LGPL as well as a GFDL are included in chapter Licenses[3]. Please note that images in the public domain do not require attribution. You may click on the image numbers in the following table to open the webpage of the images in your webbrower.

3 Chapter 14 on page 161

1	Cormullion[4]	
2	Cormullion[5]	
3	Cormullion[6]	
4	Cormullion[7]	
5	Cormullion[8]	
6	Cormullion[9]	
7	Cormullion[10]	
8	Cormullion[11]	
9	Cormullion[12]	
10	Cormullion[13]	
11	Cormullion[14]	
12	Cormullion[15]	

4 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
5 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
6 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
7 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
8 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
9 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
10 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
11 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
12 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
13 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
14 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1
15 http://commons.wikimedia.org/w/index.php?title=User:Cormullion&action=edit&redlink=1

14 Licenses

14.1 GNU GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright © 2007 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program–to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers’ and authors’ protection, the GPL clearly explains that there is no warranty for this free software. For both users’ and authors’ sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users’ freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow. TERMS AND CONDITIONS 0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.

A “covered work” means either the unmodified Program or a work based on the Program.

To “propagate” a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To “convey” a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays “Appropriate Legal Notices” to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion. 1. Source Code.

The “source code” for a work means the preferred form of the work for making modifications to it. “Object code” means any non-source form of a work.

A “Standard Interface” means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The “System Libraries” of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A “Major Component”, in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The “Corresponding Source” for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work’s System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work. 2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary. 3. Protecting Users’ Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work’s users, your or third parties’ legal rights to forbid circumvention of technological measures. 4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program’s source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee. 5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

* a) The work must carry prominent notices stating that you modified it, and giving a relevant date. * b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”. * c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it. * d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an “aggregate” if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate. 6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

* a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange. * b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge. * c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b. * d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements. * e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A “User Product” is either (1) a “consumer product”, which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, “normally used” refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

“Installation Information” for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying. 7. Additional Terms.

“Additional permissions” are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

* a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or * b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or * c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or * d) Limiting the use for publicity purposes of names of licensors or authors of the material; or * e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or * f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered “further restrictions” within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way. 8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10. 9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so. 10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An “entity transaction” is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it. 11. Patents.

A “contributor” is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s “contributor version”.

A contributor’s “essential patent claims” are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, “control” includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor’s essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a “patent license” is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To “grant” such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. “Knowingly relying” means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient’s use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is “discriminatory” if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law. 12. No Surrender of Others’ Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program. 13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such. 14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version. 15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION. 16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. 17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author> This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <http://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <http://www.gnu.org/philosophy/why-not-lgpl.html>.

14.2 GNU Free Documentation License

Version 1.3, 3 November 2008

Copyright © 2000, 2001, 2002, 2007, 2008 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies. 3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission. * B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement. * C. State on the Title page the name of the publisher of the Modified Version, as the publisher. * D. Preserve all the copyright notices of the Document. * E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices. * F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below. * G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice. * H. Include an unaltered copy of this License. * I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence. * J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission. * K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein. * L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles. * M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version. * N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section. * O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements". 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document. 11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with ... Texts." line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

14.3 GNU Lesser General Public License