Julia 1.0 Programming

Second Edition

Dynamic and high-performance programming to build fast scientific applications

Packt

www.packt.com

Ivo Balbaert

Julia 1.0 Programming
Second Edition

Dynamic and high-performance programming to build fast scientific applications

Ivo Balbaert

Packt

BIRMINGHAM - MUMBAI

Julia 1.0 Programming Second Edition

Commissioning Editor: Richa Tripathi
Acquisition Editor: Denim Pinto
Content Development Editor: Pooja Parvatkar
Technical Editor: Romy Dias
Copy Editor: Safis Editing
Project Coordinator: Ulhas Kambali
Proofreader: Safis Editing
Indexer: Tejal Daruwale Soni
Graphics: Tom Scaria
Production Coordinator: Shraddha Falebhai

First published: February 2015
Second edition: September 2018

Production reference: 1200918

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78899-909-0

www.packtpub.com

Contributors

About the author

Ivo Balbaert has been a lecturer in web programming and databases at CVO Antwerpen (www.cvoantwerpen.be), a community college in Belgium. He received a Ph.D. in Applied Physics from the University of Antwerp in 1986. He worked for 20 years in the software industry as a developer and consultant in several companies, and for 10 years as project manager at the University Hospital of Antwerp. From 2000 onwards, he switched to partly teaching and partly developing software (at KHM Mechelen, CVO Antwerpen). He also wrote an introductory book in Dutch about developing in Ruby and Rails, *Programmeren met Ruby en Rails*, by Van Duuren Media. In 2012, he authored a book on the Go programming language, *The Way To Go*, by IUniverse. He wrote a number of introductory books for new programming languages, notably Dart, Julia, Rust, and Red, all published by Packt.

I would like to thank the technical reviewer, Malcolm Sherrington, for the many useful remarks that improved the text.

About the reviewer

Malcolm Sherrington has been working in computing for over 35 years. He holds degrees in mathematics, chemistry, and engineering. He is running his own company, focusing on the aerospace, healthcare, and finance sectors, with specific interests in High-Performance Computing and applications of GPUs and parallelism. Always hands-on, Malcolm started programming scientific problems in Fortran and C, progressing through Ada and Common Lisp, and recently became involved with data processing and analytics in Perl, Python, and R. Malcolm is the organizer of the London Julia User Group

Table of Contents

Preface

Julia is a now a well-established programming language. It was developed at MIT in the Applied Computing Group under the supervision of Prof. Alan Edelman. Its development started in 2009, and it was first presented publicly in February 2012. It has now reached its first production version: Julia v1.0 (published Aug 8, 2018), which means that stability is guaranteed for the complete duration of version 1. It is based on clear and solid principles, and its popularity is steadily increasing in the technical, data scientist, and high-performance computing arena.

Who this book is for

This book is intended for data scientists and all those who work on technical and scientific computation projects. It will get you up and running quickly with Julia to start simplifying your projects' applications. The book assumes you already have some basic working knowledge of a high-level dynamic language such as MATLAB, R, Python, or Ruby.

What this book covers

`Chapter 1`, *Installing the Julia Platform*, explains how to install all the necessary components for a Julia environment. It teaches you how to work with Julia's console (the REPL) and discusses some of the more elaborate development editors you can use.

`Chapter 2`, *Variables, Types, and Operations*, discusses the elementary built-in types in Julia and the operations that can be performed on them so that you are prepared to start writing code with them.

`Chapter 3`, *Functions*, teaches you why functions are the basic building blocks of Julia, and how to effectively use them.

`Chapter 4`, *Control Flow*, shows Julia's elegant control constructs, how to perform error handling, and how to use coroutines (called Tasks in Julia) to structure the execution of your code.

`Chapter 5`, *Collection Types*, explores the different types that group individual values, such as arrays and matrices, tuples, dictionaries, and sets.

`Chapter 6`, *More on Types, Methods, and Modules*, digs deeper into the type concept and how it is used in multiple dispatch to get C-like performance. Modules, a higher code organizing concept, are discussed as well.

`Chapter 7`, *Metaprogramming in Julia*, touches on deeper layers of Julia, such as expressions and reflection capabilities, and demonstrates the power of macros.

`Chapter 8`, *I/O, Networking, and Parallel Computing*, shows how to work with data in files and databases by using DataFrames. It also looks at networking capabilities, and how to set up a parallel computing environment with Julia.

`Chapter 9`, *Running External Programs*, looks at how Julia interacts with the command-line and with other languages, and also discusses performance tips.

`Chapter 10`, *The Standard Library and Packages*, digs deeper into the standard library, and demonstrates important packages for the visualization of data.

To get the most out of this book

To run the code examples in the book, you will need the Julia platform for your computer, which can be downloaded from `http://julialang.org/downloads/`. To work more comfortably with Julia scripts, a development environment such as IJulia, Sublime Text, or Visual Studio Code is advisable. The first chapter contains detailed instructions on how to set up your Julia environment.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Julia-1.0-Programming-Second-Edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Add `/Applications/Julia-n.m.app/Contents/Resources/julia/bin/Julia` to make Julia available everywhere on your computer."

A block of code is set as follows:

```
for arg in ARGS
    println(arg)
end
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
mutable struct Point
    x::Float64
    y::Float64
    z::Float64
end
```

Any command-line input or output is written as follows:

```
julia> include("hello.jl")
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Start it up, go to Settings, and then Install Panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Installing the Julia Platform

This chapter guides you through the download and installation process of all the necessary components of Julia. The topics covered in this chapter are as follows:

- Installing Julia
- Working with Julia's REPL
- Startup options and Julia scripts
- Packages
- Installing and working with IJulia
- Installing Juno
- Installing julia-vscode
- Installing Sublime-IJulia
- Other editors and IDEs
- How Julia works

By the end of this chapter, you will have a running Julia platform. Moreover, you will be able to work with Julia's shell as well as with editors or integrated development environments with a lot of built-in features to make development more comfortable.

Installing Julia

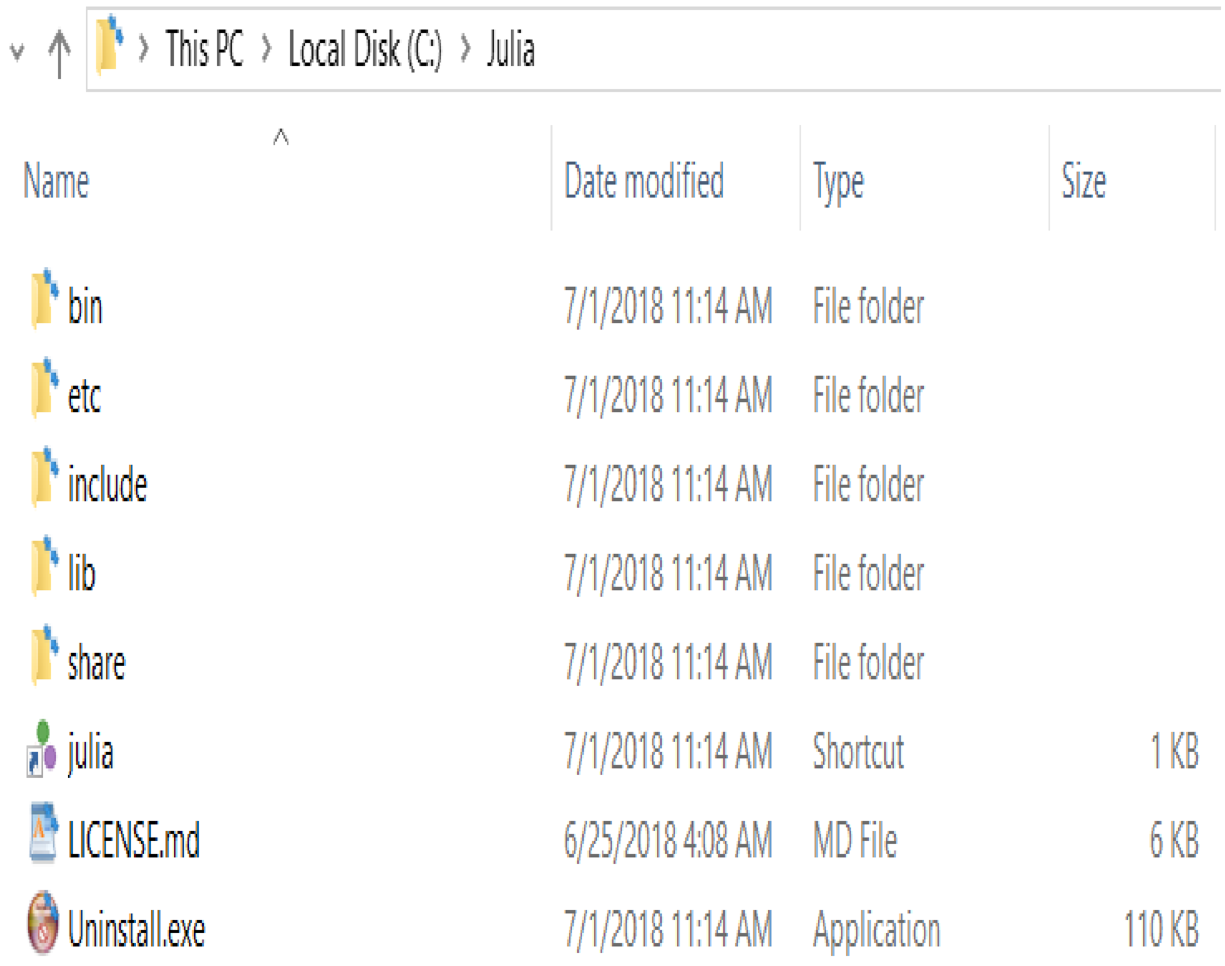
The Julia platform, in binary (that is, executable) form, can be downloaded from `http://julialang.org/downloads/`. It exists for three major platforms (Windows, Linux, and OS X) in 32- and 64-bit format, and it is delivered as a package or in an archive version. FreeBSD 64-bit is also supported.

You should use the current official stable release when doing serious professional work with Julia. At the time of writing, Julia has reached its version 1.0 production release. The previous link contains detailed and platform-specific instructions for the installation. We will not repeat these instructions here completely, but we will summarize some important points.

Windows OS

Keep in mind that your Windows OS must be version 7 or higher. Now, follow the steps shown here:

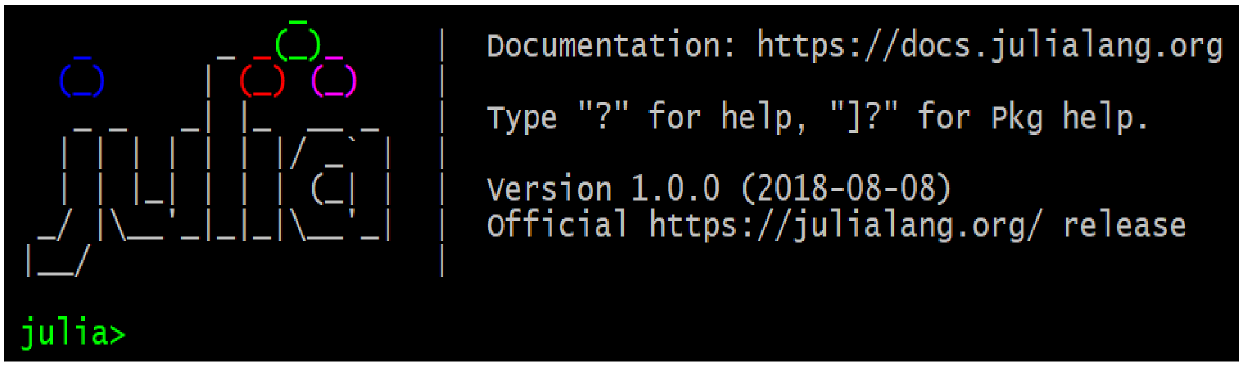
1. Download the `julia-n.m.p-win64.exe` file into a temporary folder (`n.m.p` is the version number, such as `0.7.0` or `0.1.0`; `win32`/`win64` are the 32- and 64-bit versions, respectively; a release candidate file looks like `julia-1.0.0-rc1-nnnnnnn-win64` (where `nnnnnnn` is a checksum number such as `0480f1b`)).
2. Double-click on the file (or right-click and select Run as Administrator if you want Julia installed for all users on the machine). Click OK on the security dialog message. Then, choose the installation directory (for example, for `C:\julia`, the default installation folder is: `C:\Users\UserName\AppData\Local\Julia-n.m.p` (where `n.m.p` is the version number)) and the setup program will extract the archive into the chosen folder, producing the following directory structure, and taking some 800 MB of disk space:



Name	Date modified	Type	Size
bin	7/1/2018 11:14 AM	File folder	
etc	7/1/2018 11:14 AM	File folder	
include	7/1/2018 11:14 AM	File folder	
lib	7/1/2018 11:14 AM	File folder	
share	7/1/2018 11:14 AM	File folder	
julia	7/1/2018 11:14 AM	Shortcut	1 KB
LICENSE.md	6/25/2018 4:08 AM	MD File	6 KB
Uninstall.exe	7/1/2018 11:14 AM	Application	110 KB

The Julia folder structure in Windows

3. A menu shortcut will be created which, when clicked, starts the Julia command-line version or **Read Evaluate Print Loop** (**REPL**), as shown in the following screenshot:



```
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.0.0 (2018-08-08)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

The Julia REPL

4. On Windows, if you have chosen `C:\Julia` as your installation directory, this is the `C:\Julia\bin\julia.exe` file. Add `C:\Julia\bin` to your `PATH` variable if you want the REPL to be available on any command window.
5. More information on Julia's installation for the Windows OS can be found at `https://github.com/JuliaLang/julia/blob/master/README.windows.md`.

OS X

Installation for OS X is straightforward, and can be done using the standard software installation tools for the platform. Add `/Applications/Julia-n.m.app/Contents/Resources/julia/bin/Julia` to make Julia available everywhere on your computer.

Linux OS

Generic Linux binaries for x86 can be downloaded. This will get you a compressed `tar.gz` archive that will have a name similar to `julia-1.0-linux-x86_64.tar.gz`, for example, in your `~/Downloads` directory in Ubuntu. Open up a Terminal window and navigate to the `Downloads` directory using `cd Downloads`. Move the `tar.gz` file to a directory of your choice, and then extract the `tar.gz` file using the `tar -zxvf julia-1.0-linux-x86_64.tar.gz` command. A directory with the extracted contents will be generated in the same parent directory as the compressed archive with a name similar to `julia-n.m.p`, where `n.m.p` is Julia's version number.

This is the directory from which Julia will be run; no further installation is needed. To run it, simply navigate to the `julia-n.m.p\bin` directory in your Terminal and type: `./julia`.

If you want to be at the bleeding edge of development, you can download the nightly builds instead of the stable releases from `https://julialang.org/downloads/nightlies.html`. The nightly builds are generally less stable, but will contain the most recent features. They are available for Windows, Linux, and OS X.

The path to the Julia executable is contained in the environment variable, `JULIA_BINDIR` (for example, in our installation procedure, this was `C:\Julia\bin` on Windows).

If you want code to be run whenever you start a Julia session, put it in `/home/.juliarc.jl` on Ubuntu, `~/.juliarc.jl` on OS X, or `C:\Users\username\.juliarc.jl` on Windows.

Building from source

Download the source code, rather than the binaries, if you intend to contribute to the development of Julia itself, or if no Julia binaries are provided for your operating system or particular computer architecture. The Julia source code can be found on GitHub at `https://github.com/JuliaLang/julia.git`. Compiling the source code will get you the latest Julia version, not the stable version (if you want the latter, download the binaries, and refer to the previous section).

Because of the diversity of platforms and the possible issues involved, we refer you to `https://github.com/JuliaLang/julia`, and in that, the *Source Download and Compilation* section.

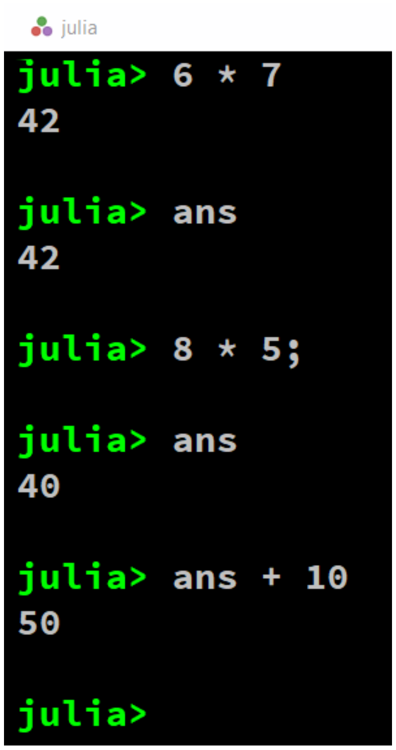
JuliaPro

Another alternative is JuliaPro, which is available from `https://juliacomputing.com/products/juliapro.html`. This is an Anaconda-style Julia repository, which, at present, is only up to version 0.6.4. It does come with about 200+ verified ready-to-go packages, and is a very good way for beginners to start. JuliaPro version 1.0 will probably become available after some time.

There are two ways of using Julia. As described in the previous section, we can use the Julia shell for interactive work. Alternatively, we can write programs in a text file, save them with a `.jl` extension, and let Julia execute the program by starting it by running `julia program.jl`.

Working with Julia's REPL

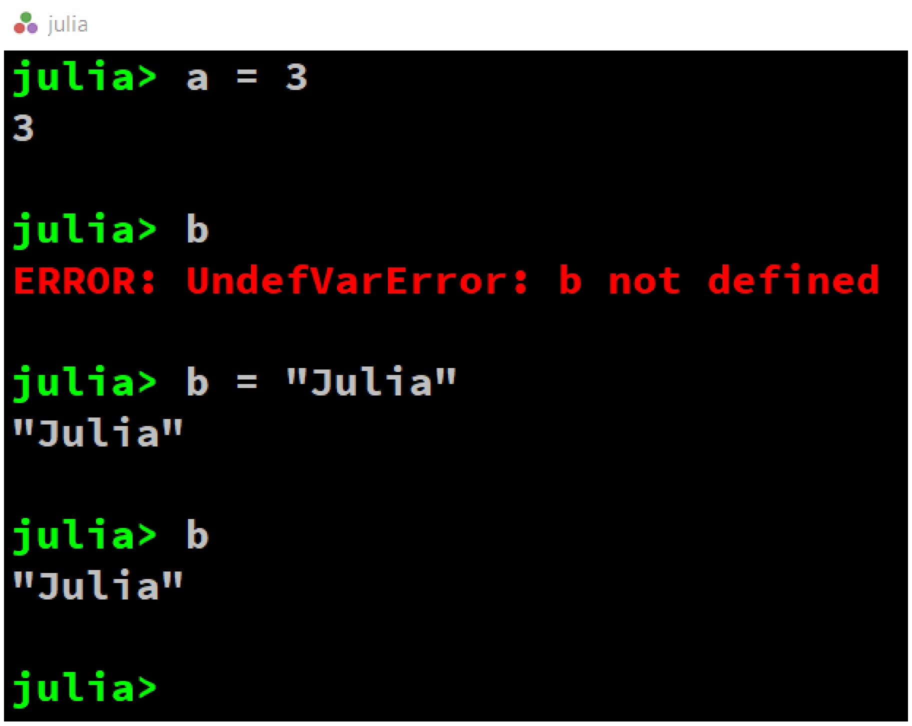
We started with Julia's REPL in the previous section to verify the correctness of the installation by issuing the `julia` command in a Terminal session. The REPL is Julia's working environment, where you can interact with the **just in time** (**JIT**) compiler to test out pieces of code. When satisfied, you can copy and paste this code into a file with a `.jl` extension, such as `program.jl`. Alternatively, you can continue to work on this code from within a text editor or an IDE, such as the ones we will point out later in this chapter. After the banner with Julia's logo has appeared, you will get a `julia>` prompt for the input. To end this session and get to the OS Command Prompt, type *Ctrl* + *D*, and hit *Enter*. To evaluate an expression, type it and press *Enter* to show the result, as shown in the following screenshot:



```
julia> 6 * 7
42

julia> ans
42

julia> 8 * 5;

julia> ans
40

julia> ans + 10
50

julia>
```

Working with the REPL (1)

If, for some reason, you don't need to see the result, end the expression with a `;` (semicolon) such as `8 * 5;` In both the cases, the resulting value is stored, for convenience, in a variable named `ans` that can be used in expressions, but only inside the REPL. You can bind a value to a variable by entering an assignment as `a = 3`. Julia is dynamic, and we don't need to enter a type for `a`, but we do need to enter a value for the variable so that Julia can infer its type. Using a variable `b` that is not bound to the `a` value results in the `ERROR: UndefVarError: b not defined` message. Strings are delineated by double quotes (`" "`), as in `b = "Julia"`. The following screenshot illustrates this in the REPL:



```
julia> a = 3
3

julia> b
ERROR: UndefVarError: b not defined

julia> b = "Julia"
"Julia"

julia> b
"Julia"

julia>
```

Working with the REPL (2)

Previous expressions can be retrieved in the same session by working with the up and down arrow keys. The following key bindings are also handy:

- To clear or interrupt a current command, press *Ctrl* + *C*
- To clear the screen, press *Ctrl* + *L* (variables are kept in memory)

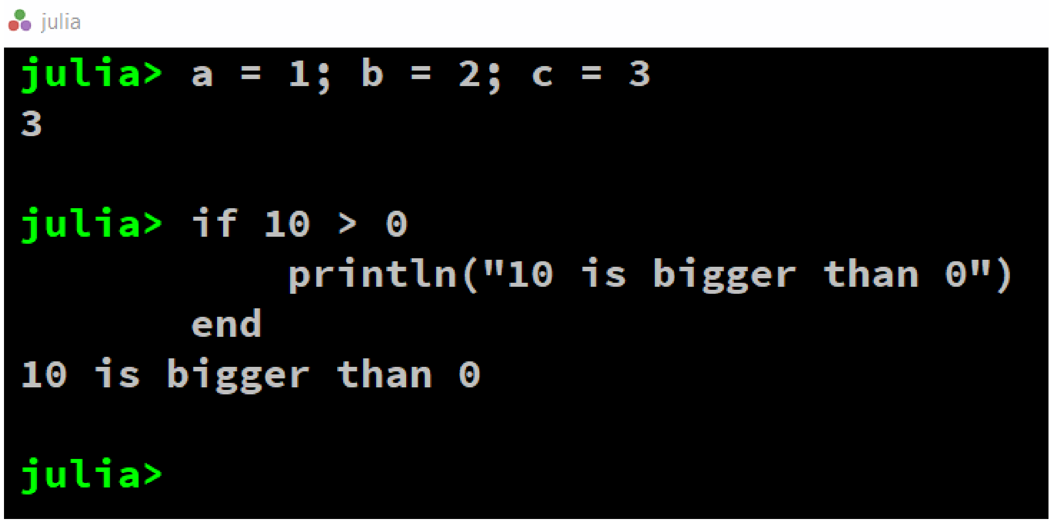
Commands from the previous sessions can still be retrieved, because they are stored (with a timestamp) in a `repl_history.jl` file (in `/home/$USER/.julia/logs` on Ubuntu, `C:\Users\username\.julia\logs` on Windows, or `~/.julia/logs/repl_history` on OS X). *Ctrl* + *R* (produces a `reverse-i-search` prompt) searches through these commands.

Typing `?` starts up the help mode (`help?>`) to give you quick access to Julia's documentation. Information on function names, types, macros, and so on, is given when typing in their names. Alternatively, to get more information on a variable, for example, `a`, type `?a`, and to get more information on a function such as `sort`, type `?sort`. To find all the places where a function such as `println` is defined or used, type `apropos("println")`, which gives the following output:

```
Base.Pair
Base.any
Base.@isdefined
Base.eachindex
Base.all
Base.Generator
Base.Timer
…
Printf.@sprintf
REPL.TerminalMenus.request
```

Thus, we can see that it is defined in the `Base` module, and that it is used in several other functions.

Different complete expressions on the same line have to be separated by a `;` (semicolon), and only the last result is shown. You can enter multiline expressions, as shown in the following screenshot. If the shell detects that the statement is syntactically incomplete, it will not attempt to evaluate it. Rather, it will wait for the user to enter additional lines until the multiline statement can be evaluated:



```
julia> a = 1; b = 2; c = 3
3

julia> if 10 > 0
           println("10 is bigger than 0")
       end
10 is bigger than 0

julia>
```

Working with the REPL (3)

A handy autocomplete feature also exists. Type one or more letters, press the *Tab* key twice, and then a list of functions starting with these letters appears. For example, type `so`, press the *Tab* key twice, and then you will get the list as `sort sort! sortcols sortperm sortperm! sortrows`.

If you start a line with `;`, the rest of the line is interpreted as a system shell command (try, for example, `ls`, `cd`, `mkdir`, `whoami` on Linux). The *Backspace* key returns to the Julia prompt.

A Julia script can be executed in the REPL by calling it with `include`. For example, for `hello.jl`, which contains the `println("Hello, Julia World!")` statement, the command is as follows:

```
julia> include("hello.jl")
```

The preceding command prints the output as follows:

```
Hello, Julia World!
```

Experiment a bit with different expressions to get a feeling for this environment.

Startup options and Julia scripts

Without any options, the `julia` command starts up the REPL environment. A useful option to check your environment is `julia -v`. This shows Julia's version, for example, `julia version 1.0.0`. (The `versioninfo()function` in REPL is more detailed, and the `VERSION` constant only gives you the version number: `v"1.0.0"`). An option that lets you evaluate expressions on the command line itself is `-e`, for example:

```
julia -e 'a = 6 * 7;
println(a)'
```

The preceding commands print out `42` (this also works in a PowerShell window on Windows, but in an ordinary Windows Command Prompt, use `"` instead of the `'` character).

Some other options that are useful for parallel processing will be discussed in Chapter 9, *Running External Programs*. Type `julia -h` for a list of all options.

A `script.jl` file with Julia source code can be started from the command line with the following command:

```
julia script.jl arg1 arg2 arg3
```

Here, `arg1`, `arg2`, and `arg3` are optional arguments to be used in the script's code. They are available from the global constant `ARGS`. Take a look at the `args.jl` file, which contains the following:

```
for arg in ARGS
    println(arg)
end
```

The `julia args.jl 1 Red C` command prints out `1`, `Red`, and `C` on consecutive lines.

A script file can also execute other source files by including them in the REPL; for example, `main.jl` contains `include("hello.jl")`, which will execute the code from `hello.jl` when called with `julia main.jl`.

Sometimes, it can be useful to know when code is executed interactively in the REPL, or when started up with the Julia VM with the `julia` command. This can be tested with the `isinteractive()` function. The `isinteractive.jl` script contains the following code:

```
println("Is this interactive? $(isinteractive())")
```

If you start this up in the REPL with `include("isinteractive.jl")`, the output will be `Is this interactive? true`.

When started up in a Terminal window as `julia isinteractive.jl`, the output is `Is this interactive? false`.

You can download the example code files from your account at `http://www.packtpub.com` *for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit* `https://github.com/PacktPublishing/Julia-1.0-Programming-Second-Edition`*.*

Packages

Most of the standard library in Julia (which can be found in `/share/julia/base` and `/share/julia/stdlib`, relative to where Julia was installed) is written in Julia itself. The rest of Julia's code ecosystem is contained in packages that are simply GitHub repositories. They are most often authored by external contributors, and already provide functionality for such diverse disciplines such as bioinformatics, chemistry, cosmology, finance, linguistics, machine learning, mathematics, statistics, and high-performance computing. A package listing can be found at `http://pkg.julialang.org`.

Julia's installation contains a built-in package manager, `Pkg`, for installing additional packages that are written in Julia. Version and dependency management is handled automatically by `Pkg`.

`Pkg` has a REPL mode, which can be started from within the Julia REPL by entering the `]` key, which is often called the REPL's package mode. The `Pkg` mode is shown as a blue prompt, like this: `(v1.0) pkg>`.

From this mode, we can start all functions of `Pkg`. To return to the normal REPL mode, press *Backspace* or *Ctrl* + *C*.

To initialize your environment, enter the `init` command, which creates an empty `Project.toml` file in your Julia installation folder.

Adding a new package

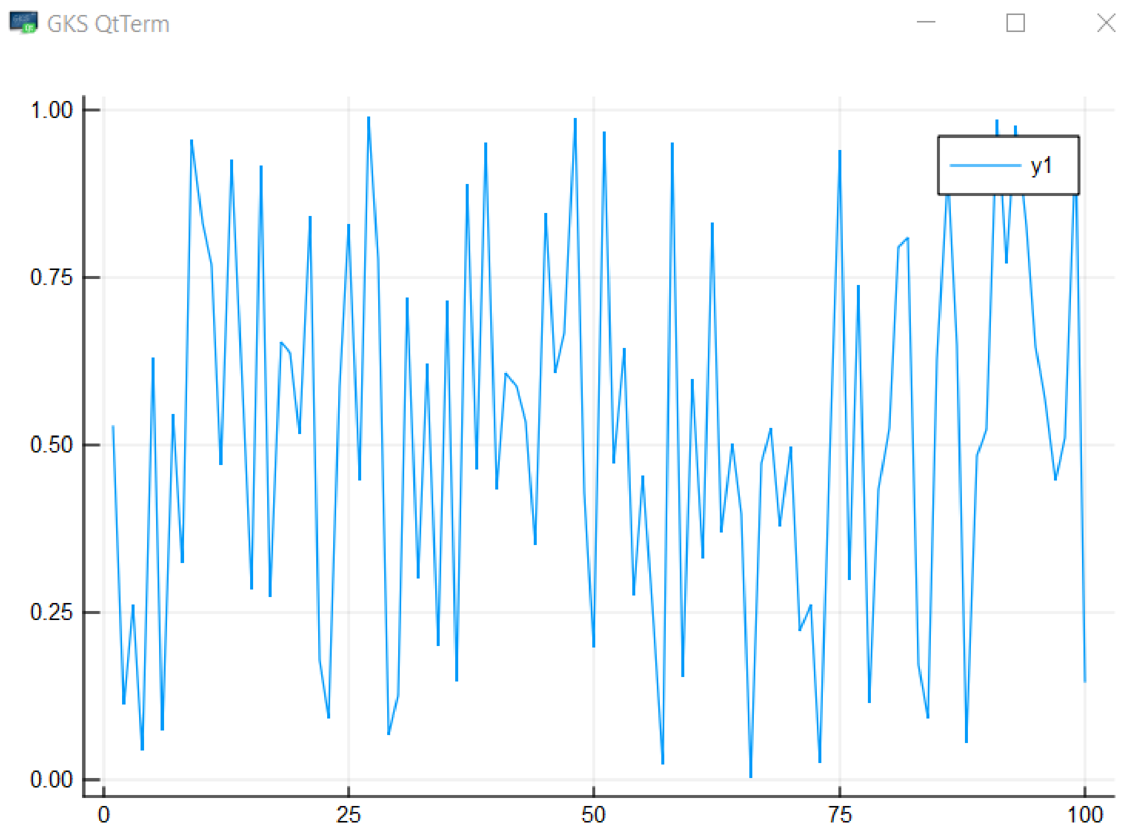
Before adding a new package, it is always a good idea to update your package database for the already installed packages with the `up` command. Then, add a new package by issuing the `add PackageName` command, and execute it by using `PackageName` in the code or in the REPL.

For example, to add 2D plotting capabilities, install the `Plots` package with `add Plots` in the `Package` mode by first typing `]`. This installs the `Plots` package and all of its dependencies, building them when needed.

To make a graph of 100 random numbers between 0 and 1, execute the following commands:

```
using Plots
plot(rand(100))
```

The `rand(100)` function is an array with 100 random numbers. This produces the following output:



A plot of white noise with Plots

After installing a new Julia version, update all the installed packages by running `up` in the `Pkg` REPL-mode.

Installing and working with IJulia

IJulia (`https://github.com/JuliaLang/IJulia.jl`) is a combination of the Jupyter Notebook interactive environment (`http://jupyter.org/`) with a Julia language backend. It allows you to work with a powerful graphical notebook (which combines code, formatted text, math, and multimedia in a single document) with a regular REPL. Detailed instructions for installation can be found at the GitHub page for IJulia (`https://github.com/JuliaLang/IJulia.jl`) and in the Julia at MIT notes (`https://github.com/stevengj/julia-mit/blob/master/README.md`). Add the IJulia package in the REPL package mode with `add IJulia`.

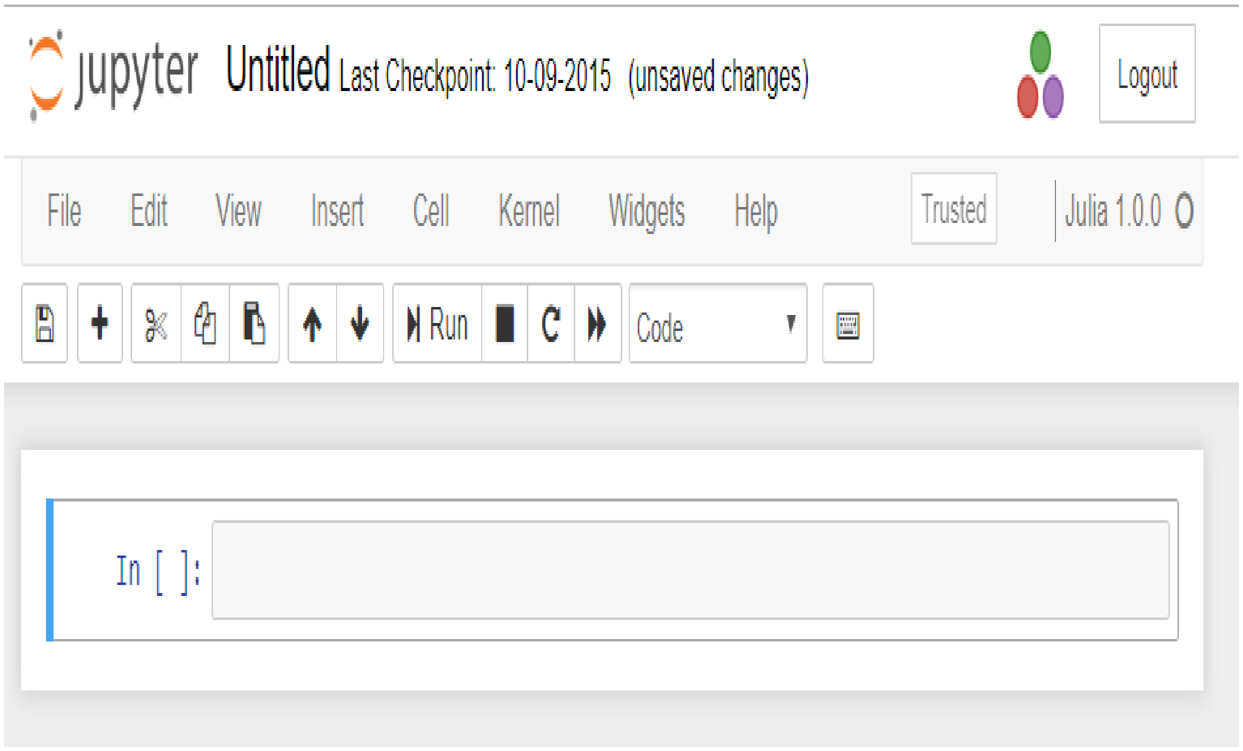
Then, whenever you want to use it, start up a Julia REPL and type the following commands:

```
using IJulia
notebook()
```

If you want to run it from the command line, type:

```
jupyter notebook
```

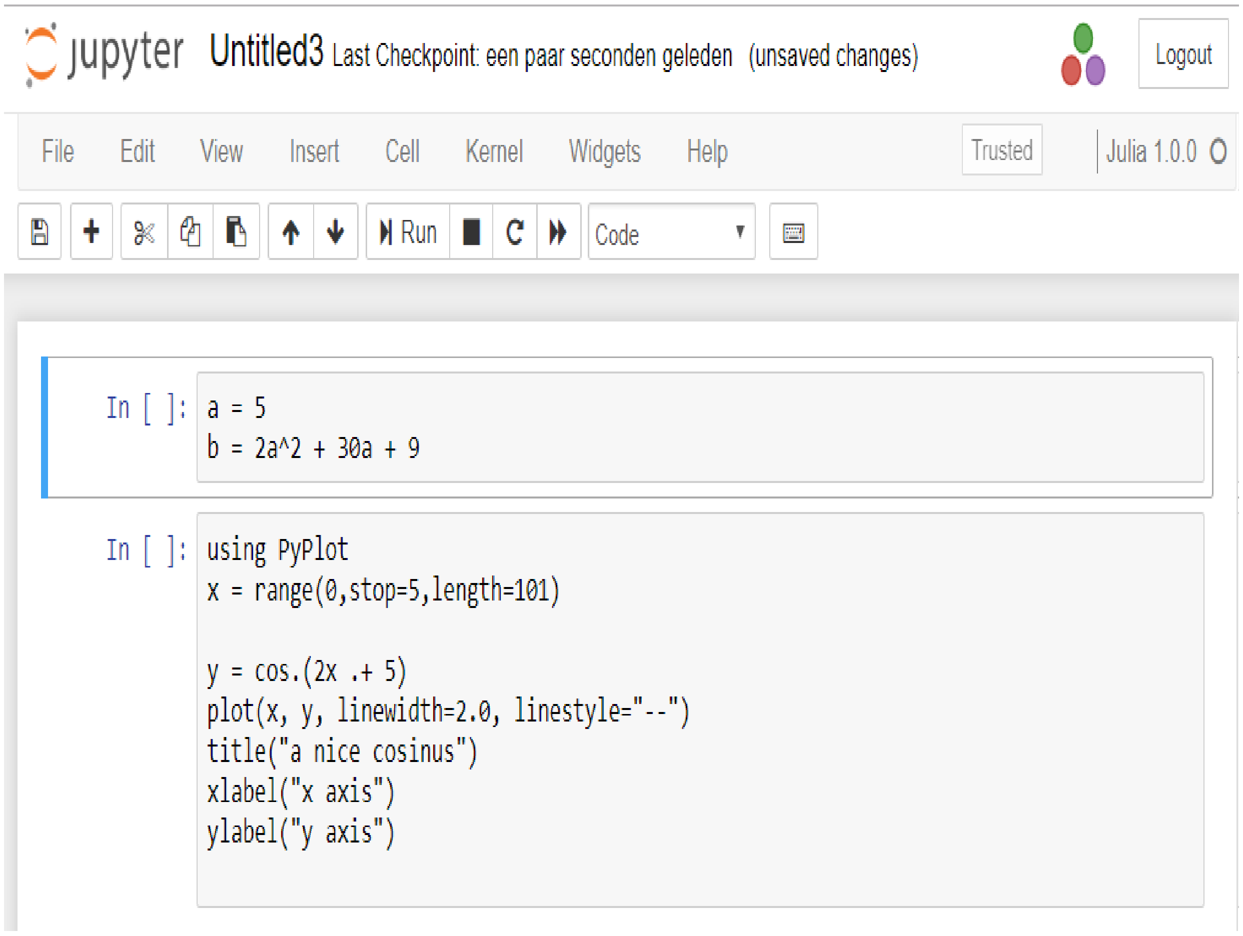
The IJulia dashboard should look as follows:



The IJulia dashboard

You should see the Jupyter logo in the upper-left corner of the browser window. Julia code is entered in the input cells (the input can be multiline) and then executed with *Shift* + *Enter*.

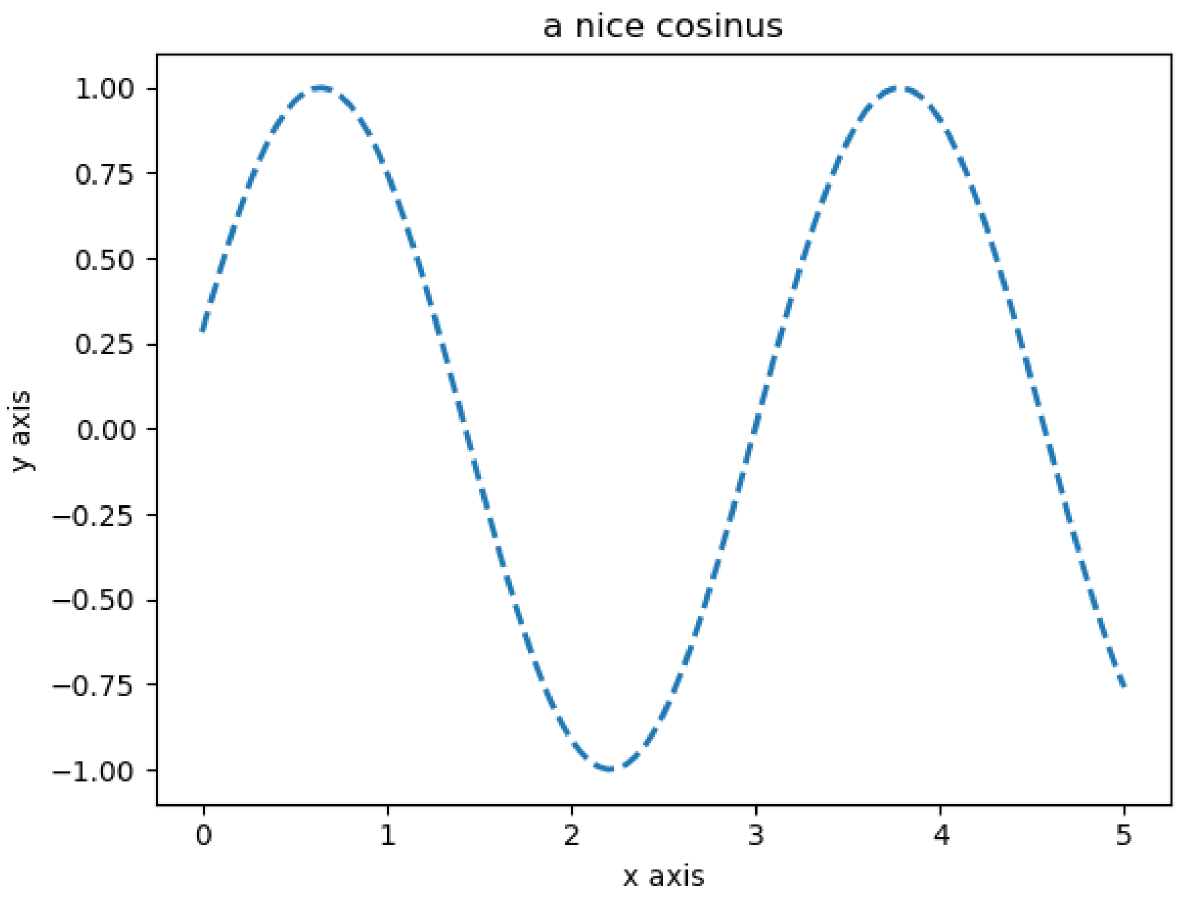
Here is a small example (`ijulia-example.jl`):


jupyter Untitled3 Last Checkpoint: een paar seconden geleden (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Julia 1.0.0
Run
Code

```
In [ ]: a = 5
        b = 2a^2 + 30a + 9
```

```
In [ ]: using PyPlot
        x = range(0,stop=5,length=101)

        y = cos.(2x .+ 5)
        plot(x, y, linewidth=2.0, linestyle="--")
        title("a nice cosinus")
        xlabel("x axis")
        ylabel("y axis")
```

The output should be something as follows:



An IJulia session example

In the first input cell, the value of `b` is calculated from `a`:

```
a = 5
b = 2a^2 + 30a + 9
```

In the second input cell, we use PyPlot. Install this package with `add PyPlot` in the REPL package mode, and by issuing `using PyPlot` in the REPL.

The `range(0,stop=5,length=101)` command defines an array of 100 equally spaced values between 0 and 5; `y` is defined as a function of `x` and is then shown graphically with the `plot` command, as follows:

```
using PyPlot
x = range(0,stop=5,length=101)
y = cos.(2x .+ 5)
plot(x, y, linewidth=2.0, linestyle="--")
title("a nice cosinus")
xlabel("x axis")
ylabel("y axis")
```

Save a notebook in file format (with the `.ipynb` extension) by downloading it from the menu.

Installing Juno

Juno (`http://junolab.org/`) is a full-fledged IDE for Julia by Mike Innes, which is based on the Atom environment. The setup page at `https://github.com/JunoLab/uber-juno/blob/master/setup.md` provides detailed instructions for installing and configuring Juno. Here is a summary of the steps:

1. Download and install Atom (`https://atom.io/`)
2. Start it up, go to Settings, and then click Install Panel
3. Enter `uber-juno` into the search box

Atom works extensively with a command palette that you can open by typing *Ctrl* + spacebar, entering a command, and then selecting it. Juno provides an integrated console, and you can evaluate single expressions in the code editor directly by typing *Ctrl* + *Enter* at the end of the line. A complete script is evaluated by typing *Ctrl* + *Shift* + *Enter*. More on basic usage can be found here: `http://docs.junolab.org/latest/man/basic_usage.html`.

Installing julia-vscode

Another IDE called `julia-vscode` is based on the Visual Studio Code editor (https://code.visualstudio.com/). Install it by following the instructions given here: https://github.com/JuliaEditorSupport/julia-vscode. This IDE provides syntax highlighting, code snippets, Julia-specific commands (execute code by pressing *F5*), an integrated Julia REPL, code completion, hover help, a linter, code navigation, and tasks for running tests, builds, benchmarks, and build documentation.

Installing Sublime-IJulia

The Sublime Text editor (`http://www.sublimetext.com/3`) has a plugin called `Julia-sublime`: `https://github.com/JuliaEditorSupport/Julia-sublime`. It gives you syntax highlighting, autocompletion, auto-indentation, and code snippets. To install it, select `Julia` from the `Package Control: Install Package` drop-down list in the Command Palette.

Other editors and IDEs

For Terminal users, the available editors are as follows:

- Vim, together with `julia-vim`, works great (`https://github.com/JuliaLang/julia-vim`)
- Emacs, with `julia-mode.el`, from the `https://github.com/JuliaLang/julia/tree/master/contrib` directory

On Linux, gedit is very good. The Julia plugin works well and provides autocompletion. Notepad++ also has Julia support from the `contrib` directory mentioned earlier.

The CoCalc project (`https://cocalc.com/`) runs Julia in the cloud within a Terminal and lets you work with Jupyter notebooks. You can also work and teach with Julia in the cloud by using the JuliaBox platform (`https://juliabox.org/`).

How Julia works

(You can safely skip this section on a first reading.) Julia works with an LLVM JIT compiler framework that is used for JIT generation of machine code. The first time you run a Julia function, it is parsed, and the types are inferred. Then, LLVM code is generated by the JIT compiler, which is then optimized and compiled down to native code. The second time you run a Julia function, the native code that's already generated is called. This is the reason why, the second time you call a function with arguments of a specific type, it takes much less time to run than the first time (keep this in mind when doing benchmarks of Julia code).

This generated code can be inspected. Suppose, for example, that we have defined a `f(x) = 2x + 5` function in a REPL session. Julia responds with the message `f` (generic function with one method); the code is dynamic because we didn't have to specify the type of `x` or `f`. Functions are, by default, generic in Julia because they are ready to work with different data types for their variables.

The `code_llvm` function can be used to see the JIT bytecode. This is the bytecode generated by LLVM, and it will be different for each target platform. For example, for the Intel x64 platform, if the `x` argument is of type `Int64`, it will be as follows:

```
julia> code_llvm(f, (Int64,))

; Function f
; Location: REPL[7]:1
; Function Attrs: uwtable
define i64 @julia_f_33833(i64) #0 {
top:
; Function *; {
; Location: int.jl:54
  %1 = shl i64 %0, 1
;}
; Function +; {
; Location: int.jl:53
  %2 = add i64 %1, 5
;}
  ret i64 %2
}
```

The `code_native` function can be used to see the assembly code that was generated for the same type of `x`:

```
julia> code_native(f, (Int64,))

        .text
; Function f {
; Location: REPL[7]:1
        pushq   %rbp
        movq    %rsp, %rbp
; Function +; {
; Location: int.jl:53
        leaq    5(%rcx,%rcx), %rax
;}
        popq    %rbp
        retq
        nopl    (%rax,%rax)
;}
```

Compare this with the code generated when `x` is of type `Float64`:

```
julia> code_native(f, (Float64,))

        .text
; Function f {
; Location: REPL[7]:1
        pushq   %rbp
        movq    %rsp, %rbp
; Function *; {
; Location: promotion.jl:314
; Function *; {
; Location: float.jl:399
        vaddsd  %xmm0, %xmm0, %xmm0
        movabsq $424735072, %rax        # imm = 0x1950F160
;}}
; Function +; {
; Location: promotion.jl:313
; Function +; {
; Location: float.jl:395
        vaddsd  (%rax), %xmm0, %xmm0
;}}
        popq    %rbp
        retq
        nopl    (%rax,%rax)
;}
```

Julia code is fast because it generates specialized versions of functions for each data type. Julia also implements automatic memory management. The user doesn't have to worry about allocating and keeping track of the memory for specific objects. Automatic deletion of objects that are not needed anymore (and hence, reclamation of the memory associated with those objects) is done using a **garbage collector** (**GC**).

The GC runs at the same time as your program. Exactly when a specific object is garbage collected is unpredictable. The GC implements an incremental mark-and-sweep algorithm. You can start garbage collection yourself by calling `GC.gc()`, or if you don't need it, you can disable it by calling `GC.enable(false)`.

The standard library is implemented in Julia itself. The I/O functions rely on the `libuv` library for an efficient, platform-independent I/O. The standard library is contained in a package called `Base`, which is automatically imported when starting Julia.

Summary

By now, you should have been able to install Julia in the working environment you prefer using. You should also have some experience with working in the REPL. We will put this to good use starting in the next chapter, where we will meet the basic data types in Julia by testing out everything in the REPL.

Variables, Types, and Operations

Julia is an optionally typed language, which means that the user can choose to specify the type of arguments passed to a function and the type of variables used inside a function. Julia's type system is the key for its performance; understanding it well is important, and it can pay off to use type annotations, not only for documentation or tooling, but also for execution speed. This chapter discusses the realm of elementary built-in types in Julia, the operations that can be performed on them, as well as the important concepts of types and scope.

The following topics are covered in this chapter:

- Variables, naming conventions, and comments
- Types
- Integers
- Floating point numbers
- Elementary mathematical functions and operations
- Rational and complex numbers
- Characters
- Strings
- Regular expressions
- Ranges and arrays
- Dates and times
- Scope and constants

You will need to follow along by typing in the examples in the REPL, or executing the code snippets in the code files of this chapter.

Variables, naming conventions, and comments

Data is stored in values such as `1`, `3.14`, and `"Julia"`, and every other value has a type, for example, the type of `3.14` is `Float64`. Some other examples of elementary values and their data types are `42` of the `Int64` type, `true` and `false` of the `Bool` type, and `'X'` of the `Char` type.

Julia, unlike many modern programming languages, differentiates between single characters and strings. Strings can contain any number of characters, and are specified using double quotes—single quotes are only used for a character literal. Variables are the names that are bound to values by assignments, such as `x = 42`. They have the type of the value they contain (or reference); this type is given by the `typeof` function. For example, `typeof(x)` returns `Int64`.

The type of a variable can change, because putting `x = "I am Julia"` now results in `typeof(x)` returning `String`. In Julia, we don't have to declare a variable (that indicates its type) such as in C or Java, for instance, but a variable must be initialized (that is, bound to a value) so that Julia can deduce its type:

```
julia> y = 7
7
typeof(y)   # Int64
julia> y + z
ERROR: UndefVarError: z not defined
```

In the preceding example, `z` was not assigned a value before we used it, so we got an error. By combining variables through operators and functions such as the `+` operator (as in the preceding example), we get expressions. An expression always results in a new value after computation. Contrary to many other languages, everything in Julia is an expression, so it returns a value. That's why working in a REPL is so great: because you can see the values at each step.

The type of variables determines what you can do with them, that is, the operators with which they can be combined with. In this sense, Julia is a strongly typed language. In the following example, `x` is still a `String` value, so it can't be summed with `y`, which is of type `Int64`, but if we give `x` a float value, the sum can be calculated, as shown in the following example:

```
julia> x + y
ERROR: MethodError: no method matching +(::String, ::Int64)
julia> x = 3.5; x + y
10.5
```

Here, the semicolon (`;`) ends the first expression and suppresses its output. Names of the variables are case-sensitive. By convention, lowercase is used with multiple words separated by an underscore. They start with a letter and, after that, you can use letters, digits, underscores, and exclamation points. You can also use Unicode characters. Use clear, short, and to-the-point names. Here are some valid variable names: `mass`, `moon_velocity`, `current_time`, `pos3`, and `ω1`. However, the last two are not very descriptive, and they should better be replaced with, for example, `particle_position` and `particle_ang_velocity`.

A line of code preceded by a hash sign (`#`) is a comment, as we can see in the following example:

```
# Calculate the gravitational acceleration grav_acc:
gc = 6.67e-11 # gravitational constant in m3/kg s2
mass_earth = 5.98e24  # in kg
radius_earth = 6378100 # in m
grav_acc = gc * mass_earth / radius_earth^2 # 9.8049 m/s2
```

Multiline comments are helpful for writing comments that span across multiple lines or commenting out code. In Julia, all lines between `#=` and `=#` are treated as a comment. For printing out values, use the `print` or `println` functions, as follows:

```
julia> print(x)
3.5
```

If you want your printed output to be in color, use `printstyled("I love Julia!", color=:red)`, which returns the argument string in the color indicated by the second argument.

The term object (or instance) is frequently used when dealing with variables of more complex types. However, we will see that, when doing actions on objects, Julia uses functional semantics. We write `action(object)` instead of `object.action()`, as we do in more object-oriented languages such as Java or C#.

In a REPL, the value of the last expression is automatically displayed each time a statement is evaluated (unless it ends with a `;` sign). In a standalone script, Julia will not display anything unless the script specifically instructs it to. This is achieved with a `print` or `println` statement. To display any object in the way the REPL does in code, use `display(object)` or `show(object)` (`show` is a basic function that prints a text representation of an object, which is often more specific than `print`).

Types

Julia's type system is unique. Julia behaves as a dynamically typed language (such as Python, for instance) most of the time. This means that a variable bound to an integer at one point might later be bound to a string. For example, consider the following:

```
julia> x = 10
10
julia> x = "hello"
"hello"
```

However, one can, optionally, add type information to a variable. This causes the variable to only accept values that match that specific type. This is done through a type of annotation. For instance, declaring `x::String` implies that only strings can be bound to `x`; in general, it looks like `var::TypeName`. These are used the most often to qualify the arguments a function can take. The extra type information is useful for documenting the code, and often allows the JIT compiler to generate better-optimized native code. It also allows the development environments to give more support, and code tools such as a linter that can check your code for possible wrong type use.

Here is an example: a function with the `calc_position` name defined as the function `calc_position(time::Float64)` indicates that this function takes one argument named `time` of type `Float64`.

Julia uses the same syntax for type assertions, which are used to check whether a variable or an expression has a specific type. Writing `(expr)::TypeName` raises an error if `expr` is not of the required type. For instance, consider the following:

```
julia> (2+3)::String
ERROR: TypeError: in typeassert, expected String, got Int64
```

Notice that the type comes after the variable name, unlike in most other languages. In general, the type of a variable can change in Julia, but this is detrimental to performance. For utmost performance, you need to write *type-stable code*. Code is type-stable if the type of every variable does not vary over time. Carefully thinking in terms of the types of variables is useful in avoiding performance bottlenecks. Adding type annotations to variables that are updated in the inner loop of a critical region of code can lead to drastic improvements in the performance by helping the JIT compiler remove some type checking. To see an excellent example where this is important, read the article available at `http://www.johnmyleswhite.com/notebook/2013/12/06/writing-type-stable-code-in-julia/`.

A lot of types exist; in fact, a whole type hierarchy is built-in in Julia. If you don't specify the type of a function argument, it has the `Any` type, which is effectively the root or parent of all types. Every object is at least of the universal type `Any`. At the other end of the spectrum, there is type `Nothing`, which has no values. No object can have this type, but it is a subtype of every other type. While running the code, Julia will infer the type of the parameters passed in a function, and with this information, it will generate optimal machine code.

You can define your own custom types as well, for instance, a `Person` type. We'll come back to this in `Chapter 6`, *More on Types, Methods, and Modules*. By convention, the names of types begin with a capital letter, and if necessary, the word separation is shown with `CamelCase`, such as `BigFloat` or `AbstractArray`.

If `x` is a variable, then `typeof(x)` gives its type, and `isa(x, T)` tests whether `x` is of type `T`. For example, `isa("ABC", String)` returns `true`, and `isa(1, Bool)` returns `false`.

Everything in Julia has a type, including types themselves, which are of type `DataType: typeof(Int64)` returns `DataType`. Conversion of a variable `var` to a type `Type1` can be done using the type name as a function, such as `Type1(var)`. For example, `Int64(3.0)` returns `3`.

However, this raises an error if type conversion is impossible, as follows:

```
julia> Int64("hello")
ERROR: MethodError: Cannot `convert` an object of type String to an object of type
Int64
```

Integers

Julia offers support for integer numbers ranging from types `Int8` to `Int128`, with 8 to 128 representing the number of bits used, and with unsigned variants with a `U` prefix, such as `UInt8`. The default type (which can also be used as `Int`) is `Int32` or `Int64`, depending on the target machine architecture. The bit width is given by the `Sys.WORD_SIZE` variable. The number of bits used by the integer affects the maximum and minimum value this integer can have. The minimum and maximum values are given by the `typemin()` and `typemax()` functions, respectively; for example, `typemax(Int16)` returns `32767`.

If you try to store a number larger than that allowed by `typemax`, overflow occurs. For example, note the following:

```
julia> typemax(Int)
9223372036854775807 # might be different on 32 bit platform
julia> ans + 1
-9223372036854775808
```

Overflow checking is not automatic, so an explicit check (for example, the result has the wrong sign) is needed when this can occur. Integers can also be written in binary (`0b`), octal (`0o`), and hexadecimal (`0x`) format.

For computations needing arbitrary-precision integers, Julia has a `BigInt` type. These values can be constructed as `BigInt(number)` or `big(number)`, and support the same operators as normal integers. Conversions between numeric types are automatic, but not between the primitive types and the big types. The normal operations of addition (`+`), subtraction (`-`), and multiplication (`*`) apply for integers. A division (`/`) always gives a floating point number. If you only want integer divisor and remainder, use `div` and `rem`. The symbol `^` is used to obtain the power of a number.

The logical values, `true` and `false`, of type `Bool` are also integers with 8 bits. `0` amounts to `false`, and `1` to `true`. Negation can be done with the `!` operator; for example, `!true` is `false`. Comparing numbers with `==` (equal), `!=` or `<` and `>`

returns a `Bool` value, and comparisons can be chained after one another (as in `0 < x < 3`).

Floating point numbers

Floating point numbers follow the IEEE 754 standard and represent numbers with a decimal point, such as `3.14`, or an exponent notation, such as `4e-14`, and come in the types `Float16` up to `Float64`, the last one being used for double precision.

Single precision is achieved through the use of the `Float32` type. Single precision float literals must be written in scientific notation, such as `3.14f0`, but with `f`, where one normally uses `e`. That is, `2.5f2` indicates `2.5*10^2` with single precision, while `2.5e2` indicates `2.5*10^2` in double precision. Julia also has a `BigFloat` type for arbitrary-precision floating numbers computations.

A built-in type promotion system takes care of all the numeric types that can work together seamlessly, so that there is no explicit conversion needed. Special values exist: `Inf` and `-Inf` are used for infinity, and `NaN` is used for "not a number" values such as the result of `0/0` or `Inf - Inf`.

Floating point arithmetic in all programming languages is often a source of subtle bugs and counter-intuitive behavior. For instance, note the following:

```
julia> 0.1 + 0.2
0.30000000000000000004
```

This happens because of the way the floating point numbers are stored internally. Most numbers cannot be stored internally with a finite number of bits, such as 1/3 having no finite representation in base 10. The computer will choose the closest number it can represent, introducing a small **round-off error**. These errors might accumulate over the course of long computations, creating subtle problems.

Maybe the most important consequence of this is the need to avoid using equality when comparing floating point numbers:

```
julia> 0.1 + 0.2 == 0.3
false
```

A better solution is to use `>=` or `<=` comparisons in logical tests that involve floating point numbers, wherever possible.

Elementary mathematical functions and operations

You can view the binary representation of any number (integer or float) with the `bitstring` function, for example, `bitstring(3)` returns `"0000000000000000000000000000000000000000000000000000000000000011"`.

To round a number, use the `round()` function which returns a floating point number. All standard mathematical functions are provided, such as `sqrt()`, `cbrt()`, `exp()`, `log()`, `sin()`, `cos()`, `tan()`, `erf()` (the error function), and many more (refer to the URL mentioned at the end of this section). To generate a random number, use `rand()`.

Use parentheses `( )` around expressions to enforce precedence. Chained assignments, such as `a = b = c = d = 1`, are allowed. The assignments are evaluated right-to-left. Assignments for different variables can be combined, as shown in the following example:

```
a = 1; b = 2; c = 3; d = 4
a, b = c, d
```

Now, `a` has a value of `3` and `b` has a value of `4`. In particular, this makes an easy swap possible:

```
a, b = b, a   # now a is 4 and b is 3
```

Like in many other languages, the Boolean operators working on the `true` and `false` values for *and*, *or*, and *not* have `&&`, `||`, and `!` as symbols, respectively. Julia applies a short-circuit optimization here. That means the following:

- In `a && b`, `b` is not evaluated when `a` is false (since `&&` is already false)
- In `a || b`, `b` is not evaluated when `a` is true (since `||` is already true)

The operators `&` and `|` are also used for non-short-circuit Boolean evaluations.

Julia also supports bitwise operations on integers. Note that `n++` or `n--` with `n` as an integer does not exist in Julia, as it does in C++ or Java. Use `n += 1` or `n -= 1` instead.

For more detailed information on operations, such as the bitwise operators, special precedence, and so on, refer to `http://docs.julialang.org/en/latest/manual/mathematical-operations/`.

Rational and complex numbers

Julia supports these types out of the box. The global constant `im` represents the square root of `-1`, so that `3.2 + 7.1im` is a complex number with floating point coefficients, so it is of the type `Complex{Float64}`.

This is the first example of a **parametric type** in Julia. For this example, we can write this as `Complex{T}`, where type `T` can take a number of different type values, such as `Int32`, `Int64`, or `Float64`.

All operations and elementary functions, such as `exp()`, `sqrt()`, `sinh()`, `real()`, `imag()`, `abs()`, and so on, are also defined on complex numbers; for example, `abs(3.2 + 7.1im) = 7.787810988975015`.

If `a` and `b` are two variables that contain a number, use `complex(a,b)` to form a complex number with them. Rational numbers are useful when you want to work with exact ratios of integers, for example, `3//4`, which is of type `Rational{Int64}`.

Again, comparisons and standard operations are defined: `float()` converts to a floating point number, and `num()` and `den()` gives the numerator and denominator. Both types work together seamlessly with all the other numeric types.

Characters

Like C or Java, but unlike Python, Julia implements a type for a single character, the `Char` type. A character literal is written as `'A'`, where `typeof('A')` returns `Char`. A `Char` value is a Unicode code point, and it ranges from `'\0'` to `'\Uffffffff'`. Convert this to its code point with `Int()`: `Int('A')` returns `65`, and `Int('α')` returns `945`, so this takes two bytes.

The reverse also works: `Char(65)` returns `'A'`, `Char(945)` returns `'\u3b1'`, which is the code point for `α` (`3b1` is hexadecimal for `945`).

Unicode characters can be entered by a `\u` in single quotes, followed by four hexadecimal digits (ranging from 0-9 or A-F), or `\U` followed by eight hexadecimal digits. The `isvalid(Char, value)` function can test whether a number returns an existing Unicode character: `isvalid(Char,0x3b1)` returns `true`. The normal escape characters, such as `\t` (tab), `\n` (newline), `\'`, and so on, also exist in Julia.

Strings

Literal strings are always of type `String`:

```
julia> typeof("hello")
String
```

This is also true if they contain UTF-8 characters that cannot be represented in ASCII, as in this example:

```
julia> typeof("Güdrun")
String
```

UTF-16 and UTF-32 are also supported. Strings are contained in double quotes (`" "`) or triple quotes (`""" """`). They are immutable, which means that they cannot be altered once they have been defined:

```
julia> s = "Hello, Julia"
julia> s[2] = 'z'
ERROR: MethodError: no method matching setindex!(::String, ::Char, ::Int64)
```

`String` is a succession, or an array of characters (see the *Ranges and arrays* section) that can be extracted from the string by indexing it, starting from `1`: with `str = "Julia"`, `str[1]` returns the character `'J'`, and `str[end]` returns the character `'a'`, the last character in the string. The index of the last byte is also given by `endof(str)`, and `length()` returns the number of characters. These two are different if the string contains multi-byte Unicode characters, for example, `endof("Güdrun")` gives `7`, while `length("Güdrun")` gives `6`.

Using an index less than one or greater than the index of the last byte gives a `BoundsError`. In general, strings can contain Unicode characters, which can take up to four bytes, so not every index is a valid character index. For example, for `str2 = "I am the α: the beginning"`, we have `str2[10]`, which returns `'\u3b1'` (the two-byte character representing `α`), `str2[11]` returns `ERROR: StringIndexError` (because this is the second byte of the `α` character), and `str2[12]` returns colon (`:`).

We can see 25 characters. `length(str2)` returns `25`, but the last index given by `lastindex(str2)` returns `26`. For this reason, looping over a string's characters can best be done as an iteration and not by using the index, as follows:

```
for c in str2
    println(c)
end
```

A substring can be obtained by taking a range of `indices:str[3:5]` or using `str[3:end]`, which returns `"lia"`. A string that contains a single character is different from that `Char` value: `'A' == "A"` returns `false`.

Julia has an elegant string interpolation mechanism for constructing strings: `$var` inside a string is replaced by the value of `var`, and `$(expr)`, where `expr` is an expression, is replaced by its computed value. When `a` is `2` and `b` is `3`, the following expression `"$a * $b = $(a * b)"` returns `"2 * 3 = 6"`. If you need to write the `$` sign in a string, escape it as `\$`.

You can also concatenate strings with the `*` operator or with the `string()` function: `"ABC" * "DEF"` returns `"ABCDEF"`, and `string("abc", "def", "ghi")` returns `"abcdefghi"`.

Strings prefixed with `:` are of type `Symbol`, such as `:green`; we already used it in the `printstyled` function. They are more efficient than strings and are used for IDs or keys. Symbols cannot be concatenated. They should only be used if they are expected to remain constant over the course of the execution of the program.

The `String` type is very rich, and it has 96 functions defined on it, given by `methodswith(String)`. Some useful methods include the following:

- `replace(string, str1, str2)`: This changes substrings `str1` to `str2` in string, for example, `replace("Julia","u" => "o")` returns `"Jolia"`.
- `split(string, char or [chars])`: This splits a string on the specified character or characters, for example, `split("34,Tom Jones,Pickwick Street 10,Aberdeen", ',')` returns the four strings in an array: `["34","Tom Jones","Pickwick Street 10","Aberdeen"]`. If `char` is not specified, the split is done on space characters (spaces, tabs, newlines, and so on).

Formatting numbers and strings

The `@printf` macro from the `Printf` package (we'll look deeper into macros in Chapter 7, *Metaprogramming in Julia*) takes a format string and one or more variables to substitute into this string while being formatted. It works in a manner similar to `printf` in C. You can write a format string that includes placeholders for variables, for example, as follows:

```
julia> name = "Pascal"
julia> using Printf
julia> @printf("Hello, %s \n", name) # returns Hello, Pascal
```

Because `@printf` now lives in another package, you have to do this using `Printf` first (prior to 1.0, it belonged to `Base`).

If you need a string as the return value, use the macro `@sprintf`.

The following `formatting.jl` script shows the most common formats:

```
using Printf
# d for integers:
@printf("%d\n", 1e5) #> 100000
x = 7.35679
# f = float format, rounded if needed:
@printf("x = %0.3f\n", x) #> 7.357
aa = 1.5231071779744345
bb = 33.976886930000695
@printf("%.2f %.2f\n", aa, bb) #> 1.52 33.98
# or to create another string:
str = @sprintf("%0.3f", x)
show(str) #> "7.357"
println()
# e = scientific format with e:
@printf("%0.6e\n", x) #> 7.356790e+00
# c = for characters:
@printf("output: %c\n", 'α') #> output: α
# s for strings:
@printf("%s\n", "I like Julia")
# right justify:
@printf("%50s\n", "text right justified!")
# p for pointers:
@printf("a pointer: %p\n", 1e10) #> a pointer: 0x00000002540be400
```

The following output is obtained upon running the preceding script:

```
100000
x = 7.357
1.52 33.98
"7.357"
7.356790e+00
output: α
I like Julia
                   text right justified!
a pointer: 0x00000002540be400
```

A special kind of string is `VersionNumber`, which the form `v"0.3.0"` (note the preceding `v`), with optional additional details. They can be compared, and are used for Julia's versions, but also in the package versions and dependency mechanism of `Pkg` (refer to the *Packages* section of `Chapter 1`, *Installing the Julia Platform*). If you have the code that works differently for different versions, use something as follows:

```
if v"0.5" <= VERSION < v"0.6-"
# do something specific to 0.5 release series
end
```

Regular expressions

To search for and match patterns in text and other data, regular expressions are an indispensable tool for the data scientist. Julia adheres to the Perl syntax of regular expressions. For a complete reference, refer to http://www.regular-expressions.info/reference.html. Regular expressions are represented in Julia as a double (or triple) quoted string preceded by `r`, such as `r"..."` (optionally, followed by one or more of the `i`, `s`, `m`, or `x` flags), and they are of type `Regex`. The `regexp.jl` script shows some examples.

In the first example, we will match the email addresses (`#>` shows the result):

```
email_pattern = r".+@.+"
input = "john.doe@mit.edu"
println(occursin(email_pattern, input)) #> true
```

The regular expression pattern `+` matches any (non-empty) group of characters. Thus, this pattern matches any string that contains `@` somewhere in the middle.

In the second example, we will try to determine whether a credit card number is valid or not:

```
visa = r"^(?:4[0-9]{12}(?:[0-9]{3})?)$"  # the pattern
input = "4457418557635128"
occursin(visa, input)  #> true
if occursin(visa, input)
    println("credit card found")
    m = match(visa, input)
    println(m.match) #> 4457418557635128
    println(m.offset) #> 1
    println(m.offsets) #> []
end
```

The `occursin(regex, string)` function returns `true` or `false`, depending on whether the given `regex` matches the string, so we can use it in an if expression. If you want the detailed information of the pattern matching, use `match` instead of `occursin`. This either returns `nothing` when there is no match, or an object of type `RegexMatch` when the pattern is found (`nothing` is, in fact, a value to indicate that nothing is returned or printed, and it has a type of `Nothing`).

The `RegexMatch` object has the following properties:

- `match` contains the entire substring that matches (in this example, it contains the complete number)
- `offset` states at what position the matching begins (here, it is `1`)
- `offsets` gives the same information as the preceding line, but for each of the captured substrings
- `captures` contains the captured substrings as a tuple (refer to the following example)

Besides checking whether a string matches a particular pattern, regular expressions can also be used to capture parts of the string. We can do this by enclosing parts of the pattern in parentheses `( )`. For instance, to capture the username and hostname in the email address pattern used earlier, we modify the pattern as follows:

```
email_pattern = r"(.+)@(.+)"
```

Notice how the characters before `@` are enclosed in brackets. This tells the regular expression engine that we want to capture this specific set of characters. To see how this works, consider the following example:

```
email_pattern = r"(.+)@(.+)"
input = "john.doe@mit.edu"
m = match(email_pattern, input)
println(m.captures) #> Union{Nothing,
SubString{String}}["john.doe", "mit.edu"]
```

Here is another example:

```
m = match(r"(ju|l)(i)?(a)", "Julia")
println(m.match) #> "lia"
println(m.captures) #> l - i - a
println(m.offset) #> 3
println(m.offsets) #> 3 - 4 - 5
```

The `search` and `replace` functions also take regular expressions as arguments, for example, `replace("Julia", r"u[\w]*l" => "red")` returns `"Jredia"`. If you want to work with all the matches, `matchall` and `eachmatch` come in handy:

```
str = "The sky is blue"
reg = r"[\w]{3,}" # matches words of 3 chars or more
r = collect((m.match for m = eachmatch(reg, str)))
show(r) #> ["The","sky","blue"]
```

```
iter = eachmatch(reg, str)
for i in iter
    println("\"$(i.match)\" ")
end
```

The `collect` function returns an array with `RegexMatch` for each match. `eachmatch` returns an iterator, `iter`, over all the matches, which we can loop through with a simple `for` loop. The screen output is `"The"`, `"sky"`, and `"blue"`, printed on consecutive lines.

Ranges and arrays

Ranges come in handy when you have to work with an interval of numbers, for example, one up to thousand: `1:1000`. The type of this object, `typeof(1:1000)`, is `UnitRange{Int64}`. By default, the step is `1`, but this can also be specified as the second number; `0:5:100` gives all multiples of `5` up to `100`. You can iterate over a range, as follows:

```
# code from file chapter2\arrays.jl
for i in 1:2:9
    println(i)
end
```

This prints out `1`, `3`, `5`, `7`, `9` on consecutive lines.

In the previous section on *Strings*, we already encountered the array type when discussing the `split` function:

```
a = split("A,B,C,D",",")
typeof(a) #> Array{SubString{String},1}
show(a) #> SubString{String}["A","B","C","D"]
```

Julia's arrays are very efficient, powerful, and flexible. The general type format for an array is `Array{Type, n}`, with `n` number of dimensions (we will discuss multidimensional arrays or matrices in `Chapter 6`, *More on Types, Methods, and Modules*). As with the complex type, we can see that the `Array` type is generic, and all the elements have to be of the same type. A one-dimensional array (also called a **vector** in Julia) can be initialized by separating its values by commas and enclosing them in square brackets, for example, `arr = [100, 25, 37]` is a `3-element Array{Int64,1}`; the type is automatically inferred with this notation. If you want the type to be `Any`, then define it as follows: `arra = Any[100, 25, "ABC"]`.

The index starts from `1`:

```
julia> arr[0]
ERROR: BoundsError: attempt to access 3-element Array{Int64,1} at index [0]
julia> arr[1]
100
```

Notice that we don't have to indicate the number of elements. Julia takes care of that and lets an array grow dynamically when needed.

Arrays can also be constructed by passing a type parameter and a number of elements:

```
arr2 = Array{Int64}(undef, 5) # is a 5-element Array{Int64,1}
show(arr2) #> [326438368, 326438432, 326438496, 326438560, 326438624]
```

`undef` makes sure that your array gets populated with random values of the given type.

You can define an array with `0` elements of type `Float64` as follows:

```
arr3 = Float64[] #> 0-element Array{Float64,1}
```

To populate this array, use `push!`; for example, `push!(arr3, 1.0)` returns `1-element Array{Float64,1}`.

Creating an empty array with `arr3 = []` is not very useful because the element type is `Any`. Julia wants to be able to infer the type!

Arrays can also be initialized from a range with the `collect` function:

```
arr4 = collect(1:7) #> 7-element Array{Int64,1}
show(arr4) #> [1, 2, 3, 4, 5, 6, 7]
```

Of course, when dealing with large arrays, it is better to indicate the final number of elements from the start for the performance. Suppose you know beforehand that `arr2` will need `10^5` elements, but not more. If you use `sizehint!(arr2, 10^5)`, you'll be able to `push!` at least `10^5` elements without Julia having to reallocate and copy the data already added, leading to a substantial improvement in performance.

Arrays store a sequence of values of the same type (called elements), indexed by integers `1` through the number of elements (as in mathematics, but unlike most other high-level languages such as Python). Like with strings, we can access the individual elements with the bracket notation; for example, with `arr` being `[100, 25, 37]`, `arr[1]` returns `100`, and `arr[end]` is `37`. Use an invalid index result in an exception, as follows:

```
arr[6] #> ERROR: BoundsError: attempt to access 3-element Array{Int64,1} at index
[6]
```

You can also set a specific element the other way around:

```
arr[2] = 5 #> [100, 5, 37]
```

The main characteristics of an array are given by the following functions:

- The element type is given by `eltype(arr)`; in our example, this is `Int64`
- The number of elements is given by `length(arr)`, and here this is `3`
- The number of dimensions is given by `ndims(arr)`, and here this is `1`
- The number of elements in dimension `n` is given by `size(arr, n)`, and here, `size(arr, 1)` returns `3`

A `for...in` loop over an array is read-only, and you cannot change elements of the array inside it:

```
da = [1,2,3,4,5]
for n in da
    n *= 2
end
da #> 5-element Array{Int64,1}: 1 2 3 4 5
```

Instead, use an index `i`, like this:

```
for i in 1:length(da)
    da[i] *= 2
end
da #> 5-element Array{Int64,1}: 2 4 6 8 10
```

It is easy to join the array elements to a string separated by a comma character and a space, for example, with `arr4 = [1, 2, 3, 4, 5, 6, 7]`:

```
join(arr4, ", ") #> "1, 2, 3, 4, 5, 6, 7"
```

We can also use this range syntax (called a slice as in Python) to obtain subarrays:

```
arr4[1:3] #>#> 3-element array [1, 2, 3]
arr4[4:end] #> 3-element array [4, 5, 6, 7]
```

Slices can be assigned to, with one value or with another array:

```
arr = [1,2,3,4,5]
arr[2:4] = [8,9,10]
println(arr) #> 1 8 9 10 5
```

Other ways to create arrays

For convenience, `zeros(n)` returns an `n` element array with all the elements equal to `0.0`, and `ones(n)` does the same with elements equal to `1.0`.

`range(start, stop=value, length=value)` creates a vector of `n` equally spaced numbers from start to stop, for example, as follows:

```
eqa = range(0, step=10, length=5) #> 0:10:40
show(eqa) #> 0:10:40
```

You can use the following to create an array with undefined values `#undef`, as shown here:

```
println(Array{Any}(undef, 4)) #> Any[#undef,#undef,#undef,#undef]
```

To fill an array `arr` with the same value for all the elements, use `fill!(arr, 42)`, which returns `[42, 42, 42, 42, 42]`.

To create a five-element array with random `Int32` numbers, execute the following:

```
v1 = rand(Int32,5)
5-element Array{Int32,1}:
  905745764
  840462491
 -227765082
 -286641110
  16698998
```

Some common functions for arrays

If `b = [1:7]` and `c = [100,200,300]`, then you can concatenate `b` and `c` with the following command:

```
append!(b, c) #> Now b is [1, 2, 3, 4, 5, 6, 7, 100, 200, 300]
```

The array, `b`, is changed by applying this `append!` method—that's why it ends in an exclamation mark (`!`). This is a general convention.

A function whose name ends in a `!` changes its first argument.

Likewise, `push!` and `pop!` append one element at the end, or take one away and return that, while the array is changed:

```
pop!(b) #> 300, b is now [1, 2, 3, 4, 5, 6, 7, 100, 200]
push!(b, 42) # b is now [1, 2, 3, 4, 5, 6, 7, 100, 200, 42]
```

If you want to do the same operations on the front of the array, use `popfirst!` and `pushfirst!` (formerly `unshift!` and `shift!`, respectively):

```
popfirst!(b) #> 1, b is now [2, 3, 4, 5, 6, 7, 100, 200, 42]
pushfirst!(b, 42) # b is now [42, 2, 3, 4, 5, 6, 7, 100, 200, 42]
```

To remove an element at a certain index, use the `splice!` function, as follows:

```
splice!(b,8) #> 100, b is now [42, 2, 3, 4, 5, 6, 7, 200, 42]
```

Checking whether an array contains an element is very easy with the `in` function:

```
in(42, b) #> true , in(43, b) #> false
```

To sort an array, use `sort!` if you want the array to be changed in place, or sort if the original array must stay the same:

```
sort(b) #> [2,3,4,5,6,7,42,42,200], but b is not changed:
println(b) #> [42,2,3,4,5,6,7,200,42]
```

```
sort!(b) #>                                             println(b) #> b is
now changed to [2,3,4,5,6,7,42,42,200]
```

To loop over an array, you can use a simple for loop:

```
for e in arr
    print("$e ") # or process e in another way
end
```

If a dot (`.`) precedes operators such as `+` or `*`, the operation is done element-wise, that is, on the corresponding elements of the arrays:

```
arr = [1, 2, 3]
arr .+ 2 #> [3, 4, 5]
arr * 2  #> [2, 10, 6]
```

As another example, if `a1 = [1, 2, 3]` and `a2 = [4, 5, 6]`, then `a1 .* a2` returns the array `[4, 10, 18]`. On the other hand, if you want the dot (or scalar) product of vectors, use the `LinearAlgebra.dot(a1, a2)` function, which returns `32`, so this gives the same result as `sum(a1 .* a2)`:

```
using LinearAlgebra
LinearAlgebra.dot(a1, a2) #> 32
sum(a1 .* a2)
```

Lots of other useful methods exist, such as `repeat([1, 2, 3], inner = [2])`, which produces `[1,1,2,2,3,3]`.

The `methodswith(Array)` function returns `47` methods. You can use help in the REPL, or search the documentation for more information.

When you assign an array to another array, and then change the first array, both the arrays change. Consider the following example:

```
a = [1,2,4,6]
a1 = a
show(a1) #> [1,2,4,6]
a[4] = 0
show(a) #> [1,2,4,0]
show(a1) #> [1,2,4,0]
```

This happens because they point to the same object in memory. If you don't want this, you have to make a copy of the array. Just use `b = copy(a)` or `b =`

`deepcopy(a)` if some elements of `a` are arrays that have to be copied recursively.

As we have seen, arrays are mutable (in contrast to strings), and as arguments to a function, they are passed by reference. As a consequence, the function can change them, as in this example:

```
a = [1,2,3]

function change_array(arr)
  arr[2] = 25
end

change_array(a)
println(a) #>[ 1, 25, 3]
```

Suppose you have an array `arr = ['a', 'b', 'c']`. Which function on `arr` do we need to return all characters in one string?

The function `join` will do the trick: `join(arr)` returns the string `"abc"`.

`string(arr)` does not: this returns `['a', 'b', 'c']`, but `string(arr...)` does return `"abc"`. This is because `...` is the **splice operator** (also known as splat). It causes the contents of `arr` to be passed as individual arguments, rather than passing `arr` as an array.

Dates and times

To get the basic time information, you can use the `time()` function that returns, for example, `1.408719961424e9`, which is the number of seconds since a predefined date called the **epoch** (normally, the 1st of January 1970 on a Unix system). This is useful for measuring the time interval between two events, for example, to benchmark how long a long calculation takes:

```
start_time = time()
# long computation
time_elapsed = time() - start_time
println("Time elapsed: $time_elapsed")
```

Use the `Dates` module that is built in into the standard library, with `Date` for days and `DateTime` for times down to milliseconds, to implement this. Additional time zone functionality can be added through the `Timezones.jl` package.

The `Date` and `DateTime` functions can be constructed as follows, or with simpler versions with less information:

- `d = Date(2014,9,1)` returns `2014-09-01`
- `dt = DateTime(2014,9,1,12,30,59,1)` returns `2014-09-01T12:30:59.001`

These objects can be compared and subtracted to get the duration. The `Date` function parts or fields can be retrieved through accessor functions, such as `year(d)`, `month(d)`, `week(d)`, and `day(d)`. Other useful functions exist, such as `dayofweek`, `dayname`, `daysinmonth`, `dayofyear`, `isleapyear`, and so on.

Scope and constants

The region in the program where a variable is known is called the **scope** of that variable. Until now, we have only seen how to create top-level or global variables that are accessible from anywhere in the program. By contrast, variables defined in a local scope can only be used within that scope. A common example of a local scope is the code inside a function. Using global scope variables is not advisable for several reasons, notably the performance. If the value and type can change at any moment in the program, the compiler cannot optimize the code.

So, restricting the scope of a variable to local scope is better. This can be done by defining them within a function or a control construct, as we will see in the following chapters. This way, we can use the same variable name more than once without name conflicts.

Let's take a look at the following code fragment:

```
# code in chapter 2\scope.jl
x = 1.0 # x is Float64
x = 1 # now x is Int
y::Float64 = 1.0
# ERROR: syntax: type declarations on global variables are not yet supported

function scopetest()
    println(x) # 1, x is known here, because it's in global scope
    y::Float64 = 1.0
# y must be Float64, this is not possible in global scope
end

scopetest()
#> 1
#> 1.0

println(y) #> ERROR: UndefVarError: y not defined
```

Variable `x` changes its type, which is allowed, but because it makes the code type unstable, it could be the source of a performance hit. From the definition of `y` in the third line, we can see that type annotations can only be used in local scope (here, in the `scopetest()` function).

Some code constructs introduce scope blocks. They support local variables. We have already mentioned functions, but `for`, `while`, `try`, `let`, and `type` blocks can all support a local scope. Any variable defined in a `for`, `while`, `try`, or `let` block will be local unless it is used by an enclosing scope before the block.

The following structure, called a **compound expression**, does not introduce a new scope. Several (preferably short) sub-expressions can be combined in one compound expression if you start it with begin, as in this example:

```
x = begin
    a = 5
    2 * a
end # now x is 10
println(a) #> a is 5
```

After end, `x` has the value `10` and `a` is `5`. This can also be written with `( )` as follows:

```
x = (a = 5; 2 * a)  #> 10
```

The value of a compound expression is the value of the last sub-expression. Variables introduced in it are still known after the expression ends.

Values that don't change during program execution are constants, which are declared with `const`. In other words, they are immutable, and their type is inferred. It is a good practice to write their name in uppercase letters, like this:

```
const GC = 6.67e-11 # gravitational constant in m3/kg s2
```

Julia defines a number of constants, such as `ARGS` (an array that contains the command-line arguments), `VERSION` (the version of Julia that is running), and `OS_NAME` (the name of the operating system such as Linux, Windows, or Darwin), mathematical constants (such as `pi` and `e`), and `Datetime` constants (such as `Friday`, `Fri`, `August`, and `Aug`).

If you try to give a global constant a new value, you get a warning, but if you change its type, you get an error, as follows:

```
julia> GC = 3.14
      Warning: redefining constant GC
```

```
julia> GC = 10
      ERROR: invalid redefinition of constant GC
```

Constants can only be assigned a value once, and their type cannot change, so they can be optimized. Use them whenever possible in the global scope.

So, global constants are more about type than value, which makes sense, because Julia gets its speed from knowing the correct types. If, however, the constant variable is of a mutable type (for example, `Array`, `Dict` (refer to Chapter 8, *I/O, Networking, and Parallel Computing*)), then you can't change it to a different array, but you can always change the contents of that variable:

```
julia> const ARR = [4,7,1]
julia> ARR[1] = 9
julia> show(ARR) #> [9,7,1]
julia> ARR = [1, 2, 3]
  Warning: redefining constant ARR
```

To review what we have learned in this chapter, we will play with characters, strings, and arrays in the following program (`strings_arrays.jl`):

```
using Statistics
# a newspaper headline:
str = "The Gold and Blue Loses a Bit of Its Luster"
println(str)
nchars = length(str)
println("The headline counts $nchars characters") # 43
str2 = replace(str, "Blue" => "Red")

# strings are immutable
println(str) # The Gold and Blue Loses a Bit of Its Luster
println(str2)
println("Here are the characters at position 25 to 30:")
subs = str[25:30]
print("-$(lowercase(subs))-") # "-a bit -"
println("Here are all the characters:")
for c in str
    println(c)
end
arr = split(str,' ')
show(arr)
#["The","Gold","and","Blue","Loses","a","Bit","of","Its","Luster"]
nwords = length(arr)
println("The headline counts $nwords words") # 10
println("Here are all the words:")
for word in arr
    println(word)
end
arr[4] = "Red"
show(arr) # arrays are mutable
println("Convert back to a sentence:")
nstr = join(arr, ' ')
println(nstr) # The Gold and Red Loses a Bit of Its Luster

# working with arrays:
```

```
println("arrays: calculate sum, mean and standard deviation ")
arr = collect(1:100)
typeof(arr) #> Array{Int64,1}
println(sum(arr)) #> 5050
println(mean(arr)) #> 50.5
```

Summary

In this chapter, we reviewed some basic elements of Julia, such as constants, variables, and types. We also learned how to work with the basic types such as numbers, characters, strings, and ranges, and encountered the very versatile array type. In the next chapter, we will look in-depth at functions and realize that Julia deserves to be called a functional language.

Functions

Julia is first and foremost a functional language because computations and data transformations are done through functions; they are first-class citizens in Julia. Programs are structured around defining functions and to overload them for different combinations of argument types. This chapter discusses this keystone concept, covering the following topics:

- Defining functions
- Optional and keyword arguments
- Anonymous functions
- First-class functions and closures
- Recursive functions
- Broadcasting
- Map, filter, and list comprehensions
- Generic functions and multiple dispatch

Defining functions

A function is an object that gets a number of arguments (the argument list, `arglist`) as the input, then does something with these values in the function body, and returns none, one, or more value(s). Multiple arguments are separated by commas (`,`) in an `arglist` (in fact, they form a tuple, as do the return values; refer to the *Tuples* section of Chapter 5, *Collection Types*). The arguments are also optionally typed, and the type(s) can be user-defined. The general syntax is as follows:

```
function fname(arglist)
    # function body...
    return value(s)
end
```

A function's argument list can also be empty; in this case, it is written as `fname()`.

The following is a simple example:

```
# code in functions101.jl
function mult(x, y)
       println("x is $x and y is $y")
       return x * y
end
```

Function names such as `mult` are, by convention, in lower-case. They can contain Unicode characters, which are useful in mathematical notations. The `return` keyword in the last line is optional; we could have written the line as `x * y`. In general, the value of the last expression in the function is returned, but writing `return` is mostly a good idea in multiline functions to increase readability.

The function called with `n = mult(3, 4)` returns `12`, and assigns the return value to a new variable, `n`. You can also execute a function just by calling `fname(arglist)` if you only need its side-effects (that is, how the function affects the program state; for instance, by changing the global variables).

The `return` keyword can also be used within a condition in other parts of the function body to exit the function earlier, as in this example:

```
function mult(x, y)
    println("x is $x and y is $y")
    if x == 1
      return y
    end
    x * y
end
```

In this case, `return` can also be used without a value so that the function returns `nothing`.

Functions are not limited to returning a single value. Here is an example with multiple return values:

```
function multi(n, m)
    n*m, div(n,m), n%m
end
```

This returns the tuple `(16,4,0)` when called with `multi(8, 2)`. The return values can be extracted to other variables such as `x, y, z = multi(8, 2)`; then `x` becomes `16`, `y` becomes `4`, and `z` becomes `0`. In fact, you can say that Julia always returns a single value, but this value can be a tuple that can be used to pass multiple variables back to the program.

We can also have a variable with a number of arguments using the ellipsis operator (`...`). An example of this operator is as follows:

```
function varargs(n, m, args...)
    println("arguments : $n $m $args")
   end
```

Here, `n` and `m` are just positional arguments (there can be more or none at all). The `args...` argument takes in all the remaining parameters in a tuple. If we call the function with `varargs(1, 2, 3, 4)`, then `n` is `1`, `m` is `2`, and `args` has the value `(3, 4)`. When there are still more parameters, the tuple can grow; or if there are none, it can be empty `()`.The same splat operator can also be used to unpack a tuple or an array into individual arguments. For example, we can define a second variable argument function as follows:

```
function varargs2(args...)
    println("arguments2: $args")
end
```

With `x = (3, 4)`, we can call `varargs2` as `varargs2(1, 2, x...)`. Now, `args` becomes the tuple `(1, 2, 3, 4)`; the tuple `x` was spliced. This also works for arrays. If `x = [10, 11, 12]`, then `args` becomes `(1, 2, 10, 11, 12)`. The receiving function does not need to be a variable argument function, but then the number of spliced parameters must exactly match the number of arguments.

It is important to realize that, in Julia, all arguments to functions (with the exception of plain data such as numbers and `chars`) are passed by reference. Their values are not copied when they are passed, which means they can be changed from inside the function, and the changes will be visible to the calling code.

For example, consider the following code:

```
function insert_elem(arr)
  push!(arr, -10)
end

arr = [2, 3, 4]
insert_elem(arr)
# arr is now [ 2, 3, 4, -10 ]
```

As this example shows, `arr` itself has been modified by the function.

Due to the way Julia compiles, a function must be defined by the time it is actually called (but it can be used before that in other function definitions).

It can also be useful to indicate the argument types, to restrict the kind of parameters passed when calling. Our `function` header for floating point numbers would then look like: `function mult(x::Float64, y::Float64)`. When this is the only `mult` function, and we call this function with `mult(5, 6)`, we receive an error, `ERROR: MethodError: no method matching mult(::Int64, ::Int64)`, proving that Julia is indeed a strongly typed language. It does not accept integer parameters for floating point arguments.

If we define a function without types, it is generic; the Julia JIT compiler is ready to generate versions called `methods` for different argument types when needed. Define the previous function `mult` in the REPL, and you will see the output as `mult (generic function with 1 method)`.

There is also a more compact, one-line function syntax (the assignment form) for short functions, for example, `mult(x, y) = x * y`. Use this, preferably, for simple one-line functions, as it will lend the code greater clarity. Because of this, mathematical functions can also be written in an intuitive form:

```
f(x, y) = x^3 - y + x * y; f(3, 2) #=> 31
```

A function defines its own scope. The set of variables that are declared inside a function are only known inside the function, and this is also true for the arguments. Functions can be defined as top-level (global) or nested (a function can be defined within another function). Usually, functions with related functionality are grouped in their own Julia file, which is included in a main file. Or, if the function is big enough, it can have its own file, preferably with the same name as the function.

Optional and keyword arguments

When defining functions, one or more arguments can be given a default value such as `f(arg = val)`. If no parameter is supplied for `arg`, then `val` is taken as the value of `arg`. The position of these arguments in the function's input is important, just as it is for normal arguments; that's why they are called **optional positional arguments**. Here is an example of an `f` function with an optional argument `b`:

```
# code in arguments.jl:
f(a, b = 5) = a + b
```

For example, if it's `f(1)`, then it returns `6`; `f(2, 5)` returns `7`; and `f(3)` returns `8`. However, calling it with `f()` or `f(1,2,3)` returns an error, because there is no matching function `f` with zero or three arguments. These arguments are still only defined by position: calling `f(2, b = 5)` raises an error as `ERROR: function f does not accept keyword arguments.`

Until now, arguments were only defined by position. For code clarity, it can be useful to explicitly call the by name, so they are called **optional keyword arguments**. Because the arguments are given explicit names, their order is irrelevant, but they must come last and be separated from the positional arguments by a semi-colon (`;`) in the argument list, as shown in this example:

```
 k(x; a1 = 1, a2 = 2) = x * (a1 + a2)
```

Now, `k(3, a2 = 3)` returns `12`, `k(3, a2 = 3, a1 = 0)` returns `9` (so their position doesn't matter), but `k(3)` returns `9` (demonstrating that the keyword arguments are optional). Normal, optional positional, and keyword arguments can be combined as follows:

```
function allargs(normal_arg, optional_positional_arg=2; keyword_arg="ABC")
    print("normal arg: $normal_arg" - )
    print("optional arg: $optional_positional_arg" - )
    print("keyword arg: $keyword_arg")
end
```

If we call `allargs(1, 3, keyword_arg=4)`, it prints `normal arg: 1 - optional arg: 3 - keyword arg: 4`.

A useful case is when the keyword arguments are splatted as follows:

```
function varargs2(;args...)
    args
end
```

Calling this with `varargs2(k1="name1", k2="name2", k3=7)` returns `pairs(::NamedTuple)` with three entries: `(:k1,"name1") (:k2,"name2") (:k3,7)`. Now, `args` is a collection of (`key`, `value`) tuples, where each key comes from the name of the keyword argument, and it is also a symbol (refer to the *Strings* section of `Chapter 2`, *Variables, Types, and Operations*) because of the colon (`:`) as prefix.

Anonymous functions

The function `f(x, y)` at the end of the *Defining functions* section can also be written with no name, as an anonymous function: `(x, y) -> x^3 - y + x * y`. We can, however, bind it to a name, such as `f = (x, y) -> x^3 - y + x * y`, and then call it, for example, as `f(3, 2)`. Anonymous functions are also often written using the following syntax (note the space before `(x)`):

```
  function (x)
      x + 2
  end
(anonymous function)
julia> ans(3)
5
```

Often, they are also written with a lambda expression as `(x) -> x + 2`. Before the stab character (`->`) are the arguments, and after the stab character we have the return value. This can be shortened to `x -> x + 2`. A function without arguments would be written as `() -> println("hello, Julia")`.

Here is an anonymous function taking three arguments: `(x, y, z) -> 3x + 2y - z`. When the performance is important, try to use named functions instead, because calling anonymous functions involves a huge overhead. Anonymous functions are mostly used when passing a function as an argument to another function, which is precisely what we will discuss in the next section.

First-class functions and closures

In this section, we will demonstrate the power and flexibility of functions (example code can be found in `Chapter 3\first_class.jl`). Firstly, functions have their own type: `Function`. Functions can also be assigned to a variable by their name:

```
julia> m = mult
julia> m(6, 6) #> 36
```

This is useful when working with anonymous functions, such as `c = x -> x + 2`, or as follows:

```
julia> plustwo = function (x)
                     x + 2
                 end
(anonymous function)
julia> plustwo(3)
5
```

Operators are just functions written with their arguments in an infix form; for example, `x + y` is equivalent to `+(x, y)`. In fact, the first form is parsed to the second form when it is evaluated. We can confirm it in the REPL: `+(3,4)` returns `7` and `typeof(+)` returns `Function`.

A function can take *a function* (or multiple functions) as its argument, which calculates the numerical derivative of a function `f`; as defined in the following function:

```
function numerical_derivative(f, x, dx=0.01)
    derivative = (f(x+dx) - f(x-dx))/(2*dx)
    return derivative
end
```

The function can be called as `numerical_derivative(f, 1, 0.001)`, passing an anonymous function `f` as an argument:

```
f = x -> 2x^2 + 30x + 9
println(numerical_derivative(f, 1, 0.001)) #> 33.99999999999537
```

A function can also return another function (or multiple functions) as its value. This is demonstrated in the following code, which calculates the derivative of a function (this is also a function):

```
function derivative(f)
    return function(x)
  # pick a small value for h
        h = x == 0 ? sqrt(eps(Float64)) : sqrt(eps(Float64)) * x
        xph = x + h
        dx = xph - x
        f1 = f(xph) # evaluate f at x + h
        f0 = f(x) # evaluate f at x
        return (f1 - f0) / dx  # divide by h
    end
end
```

As we can see, both are excellent use cases for anonymous functions.

Here is an example of a `counter` function that returns (a tuple of) two anonymous functions:

```
function counter()
    n = 0
    () -> n += 1, () -> n = 0
end
```

We can assign the returned functions to variables:

```
  (addOne, reset) = counter()
```

Notice that `n` is not defined outside the function:

```
julia> n
ERROR: n not defined
```

Then, when we call `addOne` repeatedly, we get the following output:

```
addOne() #=> 1
addOne() #=> 2
addOne() #=> 3
reset()  #=> 0
```

What we see is that, in the `counter` function, the variable `n` is captured in the anonymous functions. It can only be manipulated by the functions, `addOne` and `reset`. The two functions are said to be **closed** over the variable `n` and both have references to `n`. That's why they are called **closures**.

Currying (also called a partial application) is the technique of translating the evaluation of a function that takes multiple arguments (or a tuple of arguments) into evaluating a sequence of functions, each with a single argument. Here is an example of function currying:

```
function add(x)
    return function f(y)
        return x + y
    end
end
```

The output returned is `add (generic function with 1 method)`.

Calling this function with `add(1)(2)` returns `3`. This example can be written more succinctly as `add(x) = f(y) = x + y` or, with an anonymous function, as `add(x) = y -> x + y`. Currying is especially useful when passing functions around, as we will see in the *Map, filter, and list comprehensions* section.

functions

Functions can be nested, as demonstrated in the following example:

```
function a(x)
    z = x * 2
    function b(z)
        z += 1
    end
    b(z)
end

d = 5
a(d) #=> 11
```

A function can also be recursive, that is, it can call itself. To show some examples, we need to be able to test a condition in code. The simplest way to do this in Julia is to use the ternary operator `?` of the form `expr ? b : c` (ternary because it takes three arguments). Julia also has a normal `if` construct. (Refer to the *Conditional evaluation* section of Chapter 4, *Control Flow*.) `expr` is a condition and, if it is true, then `b` is evaluated and the value is returned, else `c` is evaluated. This is used in the following recursive definition to calculate the sum of all the integers up to and including a certain number:

```
sum(n) =  n > 1 ? sum(n-1) + n : n
```

The recursion ends because there is a base case: when `n` is `1`, this value is returned. Here is the famous function to calculate the n^{th} Fibonacci number that is defined as the sum of the two previous Fibonacci numbers:

```
fib(n) = n < 2 ? n : fib(n-1) + fib(n-2)
```

When using recursion, care should be taken to define a base case to stop the calculation. Also, although Julia can nest very deeply, watch out for stack overflows, because, until now, Julia has not done tail call optimization automatically.

Broadcasting

A function `f` can be broadcast over all elements of an array (or matrix) by using the dot notation `f.(matrix)`; for example:

```
arr = [1.0, 2.0, 3.0]
sin.(arr) #>
3-element Array{Float64,1}:
#  0.8414709848078965
#  0.9092974268256817
#  0.1411200080598672
```

Here is another example:

```
f(x,y) = x + 7y
f.(pi, arr)
#> 3-element Array{Float64,1}:
# 10.141592653589793
# 17.141592653589793
# 24.141592653589793
```

Broadcasting is very useful in Julia to write compact expressions with arrays and matrices.

Map, filter, and list comprehensions

Maps and filters are typical for functional languages. A `map` is a function of the form `map(func, coll)`, where `func` is a (often anonymous) function that is successively applied to every element of the `coll` collection, so `map` returns a new collection. Some examples are as follows:

- `map(x -> x * 10, [1, 2, 3])` returns `[10, 20, 30]`
- `cubes = map(x-> Base.power_by_squaring(x, 3), collect(1:5))` returns `[1, 8, 27, 64, 125]`

`power_by_squaring` is an internal function in `Base`, which means it is not exported, so it has to be qualified with `Base`.

The `map` function can also be used with functions that take more than one argument. In this case, it requires a collection for each argument; for example, `map(*, [1, 2, 3], [4, 5, 6])` works per element and returns `[4, 10, 18]`.

When the function passed to `map` requires several lines, it can be a bit unwieldy to write as an anonymous function. For instance, consider using the following function:

```
map( x-> begin
            if x == 0 return 0
            elseif iseven(x) return 2
            elseif isodd(x) return 1
            end
        end, collect(-3:3))
```

This function returns `[1,2,1,0,1,2,1]`. This can be simplified with a `do` block as follows:

```
map(collect(-3:3)) do x
    if x == 0 return 0
    elseif iseven(x) return 2
    elseif isodd(x) return 1
    end
end
```

The `do x` statement creates an anonymous function with the argument `x` and passes it as the first argument to `map`.

A `filter` is a function of the form `filter(func, coll)`, where `func` is a (often anonymous) Boolean function that is checked on each element of the collection `coll`. Filter returns a new collection with only the elements on which `func` is evaluated to be true. For example, the following code filters the even numbers and returns `[2, 4, 6, 8, 10]`:

```
filter( n -> iseven(n), collect(1:10))
```

An incredibly powerful and simple way to create an array is to use a list comprehension. This is a kind of implicit loop which creates the result array and fills it with values. Some examples are as follows:

- `arr = Float64[x^2 for x in 1:4]` creates `4-element Array{Float64,1}` with elements `1.0`, `4.0`, `9.0`, and `16.0`.
- `cubes = [x^3 for x in collect(1:5)]` returns `[1, 8, 27, 64, 125]`.
- `mat1 = [x + y for x in 1:2, y in 1:3]` creates a 2 x 3 array (`Array{Int64,2}`):

```
              2  3  4
              3  4  5
```

- `table10 = [x * y for x=1:10, y=1:10]` creates a 10 x 10 array (`Array{Int64,2}`), and returns the multiplication table of 10.
- `arrany = Any[i * 2 for i in 1:5]` creates `5-element Array{Any,1}` with elements `2`, `4`, `6`, `8`, and `10`.

For more examples, you can refer to the *Dictionaries* section in Chapter 5, *Collection Types*.

Constraining the type, as with `arr`, is often helpful for performance. Using typed comprehensions everywhere for explicitness and safety in production code is certainly the best practice.

Generic functions and multiple dispatch

We have already seen that functions are inherently defined as generic, that is, they can be used for different types of their arguments. The compiler will generate a separate version of the function each time it is called with arguments of a new type. In Julia, a concrete version of a function for a specific combination of argument types is called a **method**. To define a new method for a function (also called **overloading**), just use the same function name but a different signature, that is, with different argument types. A list of all the methods is stored in a virtual method table (`vtable`) on the function itself; methods do not belong to a particular type. When a function is called, Julia will lookup in `vtable` at runtime to find which concrete method it should call, based on the types of all its arguments; this is Julia's multiple dispatch mechanism, which Python, C++, or Fortran do not implement this. It allows open extensions where normal object-oriented code would have forced you to change a class or subclass to an existing class and thus change your library. Note that only positional arguments are taken into account for multiple dispatch, and not keyword arguments.

For each of these different methods, specialized low-level code is generated, targeted to the processor's instruction set. In contrast to **object-oriented** (**OO**) languages, `vtable` is stored in the function, and not in the type (or class). In OO languages, a method is called on a single object, `object.method()`, which is generally called **single dispatch**. In Julia, one can say that a function belongs to multiple types, or that a function is specialized or overloaded for different types. Julia's ability to compile code that reads like a high-level dynamic language into machine code that performs almost entirely like C is derived from its ability to do multiple dispatch.

To make this idea more concrete, a function such as `square(x) = x * x` actually defines a potentially infinite family of methods, one for each of the possible types of the argument `x`. For example, `square(2)` will call a specialized method that uses the CPU's native integer multiplication instruction, whereas `square(2.0)` will use the CPU's native floating point multiplication instruction.

Let's see multiple dispatch in action. We will define a function `f` that takes two arguments `n` and `m` returning a string, but, in some methods, the type of `n` or `m`, or both, is annotated. (`Number` is a supertype of `Integer`, refer to the *The type hierarchy – subtypes and supertypes* section in Chapter 6, *More on Types, Methods, and Modules*.) This can be seen in the following example:

```
f(n, m) = "base case"
f(n::Number, m::Number) = "n and m are both numbers"
f(n::Number, m) = "n is a number"
f(n, m::Number) = "m is a number"
f(n::Integer, m::Integer) = "n and m are both integers"
```

This returns `f (generic function with 5 methods).`

When `n` and `m` have no type, as in `"base case"`, then their type is `Any`, the supertype of all types. Let's take a look at how the most appropriate method is chosen in each of the following function calls:

- `f(1.5, 2)` returns `n and m are both numbers`
- `f(1, "bar")` returns `n is a number`
- `f(1, 2)` returns `n and m are both integers`
- `f("foo", [1,2])` returns `base case`

Calling `f(n, m)` will never result in an error, because if no other method matches, the base case will be invoked when we add a new method:

```
f(n::Float64, m::Integer) = "n is a float and m is an integer"
```

So, the call to `f(1.5,2)` now returns `n is a float and m is an integer.`

To get a quick overview of all the versions of a function, type `methods(fname)` into the REPL. For example, `methods(+)` shows a listing of 174 methods for a generic function `+`:

```
+(x::Bool) at bool.jl:36
+(x::Bool,y::Bool) at bool.jl:39
```

```
...
+(a,b,c) at operators.jl:82
+(a,b,c,xs...) at operators.jl:83
```

You can even take a look in the source code at how they are defined, as in `base/bool.jl` in the local Julia installation or at https://github.com/JuliaLang/julia/blob/master/base/bool.jl, where we can see the addition of `Bool` variables equal to the addition of integers: `+(x::Bool, y::Bool) = int(x) + int(y)`, where `int(false)` is `0` and `int(true)` is `1`.

As a second example, `methods(sort)` shows `# 6 methods for the generic function "sort"`.

The macro `@which` gives you the exact method that is used and where in the source code that method is defined, for example, `@which 2 * 2` returns `* (x::Int64, y::Int64) at int.jl:47`. This also works the other way around. If you want to know which methods are defined for a certain type, or use that type, ask `methodswith(Type)` from the `InteractiveUtils` module. For example, here is a part of the output of `InteractiveUtils .methodswith(String)`:

```
[18] getindex(s::String, r::UnitRange{Int64}) in Base at strings/string.jl:240
[19] getindex(s::String, i::Int64) in Base at strings/string.jl:205
[20] getindex(s::String, r::UnitRange{#s56} where #s56<:Integer) in Base at
strings/string.jl:237 ...
```

As already noted, type stability is crucial for optimal performance. A function is type-stable if the return type(s) of all the output variables can be deduced from the types of the inputs. So try to design your functions with type stability in mind.

Some crude performance measurements (execution time and memory used) on the execution of functions can be obtained from the macro `@time`, for example:

```
@time fib(35)
elapsed time: 0.115188593 seconds (6756 bytes allocated) 9227465
```

`@elapsed` only returns the execution time. `@elapsed fib(35)` returns `0.115188593`.

In Julia, the first call of a method invokes the LLVM JIT compiler backend (refer to the *How Julia works* section in Chapter 1, *Installing the Julia*

Platform), to emit machine code for it, so this warm-up call will take a bit longer. Start timing or benchmarking from the second call onward, after doing a dry run.

When writing a program with Julia, first write an easy version that works. Then, if necessary, improve the performance of that version by profiling it and then fixing performance bottlenecks. We'll come back to performance measurements in the *Performance tips* section of `Chapter 9`, *Running External Programs*.

Summary

In this chapter, we saw that functions are the basic building blocks of Julia. We explored the power of functions, their arguments and return values, closures, maps, filters, and comprehensions. However, to make the code in a function more interesting, we need to see how Julia does basic control flow, iterations, and loops. This is the topic of the next chapter.

Control Flow

Julia offers many control statements that are familiar to the other languages, while also simplifying the syntax for many of them. However, tasks are probably new; they are based on the coroutine concept to make computations more flexible.

We will cover the following topics in this chapter:

- Conditional evaluation
- Repeated evaluation
- Exception handling
- Scope revisited
- Tasks

Conditional evaluation

Conditional evaluation means that pieces of code are evaluated, depending on whether a Boolean expression is either true or false. The familiar `if...elseif...else...end` syntax is used here, which is as follows:

```
# code in Chapter 4\conditional.jl
var = 7
if var > 10
    println("var has value $var and is bigger than 10.")
elseif var < 10
    println("var has value $var and is smaller than 10.")
else
    println("var has value $var and is 10.")
end
# => prints "var has value 7 and is smaller than 10."
```

The `elseif` (of which there can be more than one) or `else` branches are optional. The condition in the first branch is evaluated, only the code in that branch is executed when the condition is true, and so on; so only one branch ever gets evaluated. No parentheses around condition(s) are needed, but they can be used for clarity. Each expression tested must effectively result in a true or false value, and no other values (such as `0` or `1`) are allowed.

Because every expression in Julia returns a value, so also does the `if` expression. We can use this expression to do an assignment depending on a condition. In the preceding case, the return value is nothing since that is what `println` returns.

However, in the following snippet, the value `15` is assigned to `z`:

```
a = 10
b = 15
z = if a > b  a
    else      b
    end
```

These kinds of expression can be simplified using the ternary operator `?` (which we introduced in the *Recursive functions* section in `Chapter 3`, *Functions*) as follows:

```
z = a > b ? a : b
```

Here, only `a` or `b` is evaluated and parentheses `( )` can be added around each clause, as they are necessary for clarity. The ternary operator can be chained, but then it often becomes harder to read. Our first example can be rewritten as follows:

```
var = 7
varout = "var has value $var"
cond = var > 10 ? "and is bigger than 10." : var < 10 ? "and is
     smaller than 10" : "and is 10."
println("$varout $cond") # var has value 7 and is smaller than 10
```

Using short-circuit evaluation (refer to the *Elementary mathematical functions* section in Chapter 2, *Variables, Types, and Operations*), the statements with `if...only` are often written as follows:

```
if <cond> <statement> end is written as <cond> && <statement
if !<cond> <statement> end is written as <cond> || <statement>
```

To make this clearer, the first can be read as `<cond>` *and then* `<statement>`, and the second as `<cond>` *or else* `<statement>`.

This feature can come in handy when guarding the parameter values passed into the arguments, which calculates the square root, like in the following function:

```
function sqroot(n::Int)
    n >= 0 || error("n must be non-negative")
    n == 0 && return 0
    sqrt(n)
end
sqroot(4) #=> 2.0
sqroot(0) #=> 0.0
sqroot(-6) #=> ERROR: LoadError: n must be non-negative
```

The `error` statement effectively throws an exception with the given message and stops the code execution (refer to the *Exception handling* section in this chapter).

Julia has no switch/case statement, and the language provides no built-in pattern matching (although one can argue that multiple dispatch is a kind of pattern matching that is based not on value, but on type).

Repeated evaluation

Julia has a `for` loop for iterating over a collection or repeating some code a certain number of times. You can use a `while` loop when the repetition depends on a condition, and you can influence the execution of both loops through `break` and `continue`.

for loops

We already encountered the `for` loop when iterating over the element `e` of a collection `coll` (refer to the *Strings*, *Ranges and Arrays* sections in Chapter 2, *Variables, Types, and Operations*). This takes the following general form:

```
# code in Chapter 4\repetitions.jl
for e in coll
   # body: process(e) executed for every element e in coll
end
```

Here, `coll` can be a range, a string, an array, or any other iterable collection (for other uses, also refer to Chapter 5, *Collection Types*). The variable `e` is not known outside the `for` loop. When iterating over a numeric range, often `=` (equal to) is used instead of `in`:

```
    for n = 1:10
       print(n^3)
    end
```

(This code can be a one-liner, but is spread over three lines for clarity.) The `for` loop is generally used when the number of repetitions is known.

Use `for i in 1:n` rather than `for i in [1:n]` since the latter allocates an array while the former uses a simpler range object.

You can also use `∊` instead of `in` or `=`.

If you need to know the index when iterating over the elements of an array, run the following code:

```
arr = [x^2 for x in 1:10]
for i = 1:length(arr)
   println("the $i-th element is $(arr[i])")
end
```

A more elegant way to accomplish this uses the `enumerate` function, as follows:

```
for (ix, val) in enumerate(arr)
   println("the $ix-th element is $val")
```

```
end
```

Nested `for` loops are possible, as in this code snippet, for a multiplication table:

```
for n = 1:5
    for m = 1:5
        println("$n * $m = $(n * m)")
    end
end
```

However, nested `for` loops can often be combined into a single outer loop, as follows:

```
for n = 1:5, m = 1:5
    println("$n * $m = $(n * m)")
end
```

while loops

When you want to use looping as long as a condition stays true, use the `while` loop, which is as follows:

```
a = 10; b = 15
while a < b
    # body: process(a)
    println(a)
    global a += 1
end
# prints on consecutive lines: 10 11 12 13 14
```

In the body of the loop, something has to change the value of `a` so that the initial condition becomes false and the loop ends. If the initial condition is false at the start, the body of the `while` loop is never executed. The `global` keyword makes `a` in the current scope refer to the global variable of that name.

If you need to loop over an array while adding or removing elements from the array, use a `while` loop, as follows:

```
arr = [1,2,3,4]
while !isempty(arr)
    print(pop!(arr), ", ")
end
```

The preceding code returns the output as `4, 3, 2, 1`.

The break statement

Sometimes, it is convenient to stop the loop repetition inside the loop when a certain condition is reached. This can be done with the `break` statement, which is as follows:

```
a = 10; b = 150
while a < b
    # process(a)
    println(a)
    global a += 1
    if a >= 50
        break
     end
end
```

This prints out the numbers 10 to 49, and then exits the loop when `break` is encountered. The following is an idiom that is often used; how to search for a given element in an array, and stop when we have found it:

```
arr = rand(1:10, 10)
println(arr)
# get the index of search in an array arr:
searched = 4
for (ix, curr) in enumerate(arr)
  if curr == searched
    println("The searched element $searched occurs on index $ix")
    break
  end
end
```

A possible output might be as follows:

```
[8,4,3,6,3,5,4,4,6,6]
The searched element 4 occurs on index 2
```

The `break` statement can be used in `for` loops as well as in `while` loops. It is, of course, mandatory in a `while true...end` loop.

The continue statement

What should you do when you want to skip one (or more) loop repetitions then, nevertheless, continue with the next loop iteration? For this, you need `continue`, as in this example:

```
for n in 1:10
  if 3 <= n <= 6
    continue # skip current iteration
  end
  println(n)
end
```

This prints out, `1 2 7 8 9 10`, skipping the numbers three to six, using a chained comparison.

There is no `repeat...until` or `do...while` construct in Julia. A `do...while` loop can be simulated as follows:

```
while true
# code
  condition || break
end
```

Exception handling

When executing a program, abnormal conditions can occur that force the Julia runtime to throw an exception or error, show the exception message and the line where it occurred, and then exit. For example (follow along with the code in `Chapter 4\errors.jl`):

- Using the wrong index for an array, for example, `arr = [1,2,3]`and then asking for `arr[0]` causes a program to stop with `ERROR: BoundsError()`
- Calling `sqrt()` on a negative value, for example, `sqrt(-3)` causes `ERROR: DomainError: sqrt will only return a complex result if called with a complex argument, try sqrt(complex(x))`; the `sqrt(complex(-3))` function gives the correct result `0.0 + 1.7320508075688772im`
- A syntax error in Julia code will usually result in `LoadError`

Similar to these, there are 18 predefined exceptions that Julia can generate (refer to `http://docs.julialang.org/en/latest/manual/control-flow/#man-exception-handling`). They are all derived from a base type, `Exception`.

How can you signal an error condition yourself? You can *call* one of the built-in exceptions by *throwing* such an exception; that is, calling the `throw` function with the exception as an argument. Suppose an input field, `code`, can only accept the codes listed in `codes = ["AO", "ZD", "SG", "EZ"]`. If `code` has the value, `AR`, the following test produces `DomainError`:

```
if code in codes
    println("This is an acceptable code")
else
    throw(DomainError())
end
```

A `rethrow()` statement can be useful to hand the current exception to a higher calling code level.

Note that you can't give your own message as an argument to `DomainError()`. This is possible with the `error(message)` function (refer to the *Conditional* *evaluation* section) with a `String` message. This results in a program stopping with an `ErrorException` function and an `ERROR: message` message.

Creating user-defined exceptions can be done by deriving from the base type, `Exception`, such as `mutable struct CustomException <: Exception end` (for an explanation of `<`, refer to the *The type hierarchy - subtypes and supertypes* section in Chapter 6, *More on Types, Methods, and Modules*). These can also be used as arguments to be thrown.

In order to catch and handle possible exceptions yourself so that the program can continue to run, Julia uses the familiar `try...catch...finally` construct, which includes the following:

- The dangerous code that comes in the `try` block
- The `catch` block that stops the exception and allows you to react to the code that threw the exception

Here is an example:

```
a = []
try
    pop!(a)
catch ex
    println(typeof(ex))
    showerror(STDOUT, ex)
end
```

This example prints the output, as follows:

```
ArgumentError
    array must be non-empty
```

Popping an empty array generates an exception. The variable, `ex`, contains the exception object, but a plain catch without a variable can also be used. The `showerror` function is a handy function; its first argument can be any I/O stream, so it could be a file.

To differentiate between the different types of exception in the `catch` block, you can use the following code:

```
try
  # try this code
```

```
catch ex
  if isa(ex, DomainError)
    # do this
  elseif isa(ex, BoundsError)
    # do this
  end
end
```

Similar to `if` and `while`, `try` is an expression, so you can assign its return value to a variable. So, run the following code:

```
ret = try
          global a = 4 * 2
      catch ex
      end
```

After running the preceding code, `ret` contains the value `8`.

Sometimes, it is useful to have a set of statements to be executed no matter what, for example, to clean up resources. Typical use cases are when reading from a file or a database. We want the file or the database connection to be closed after the execution, regardless of whether an error occurred while the file or database was being processed. This is achieved with the `finally` clause of a `try...catch...finally` construct, as in the following code snippet:

```
try
   global f = open("file1.txt") # returns an IOStream(<file file1.txt>)
# operate on file f
catch ex
finally
    close(f)
end
```

`f` must be defined as `global` in `try`, otherwise it is not known in the `finally` branch.

Here is a more concrete example:

```
try
  open("file1.txt", "r") do f
            k = 0
            while(!eof(f))
                a=readline(f)
                println(a)
                k += 1
            end
         println("\nNumber of lines in file: $k")
     end
```

```
catch ex
finally
    close(f)
end
```

The `try...catch...finally` full construct guarantees that the `finally` block is always executed, even when there is a return in `try`. In general, all three combinations of `try...catch`, `try...finally`, and `try...catch...finally` are possible.

It is important to realize that `try...catch` should not be used in performance bottlenecks, because the mechanism impedes performance. Whenever feasible, test a possible exception with normal conditional evaluation.

The preceding code can be written more idiomatically as follows:

```
open("file1.txt", "w") do f
    # operate on file f
end
```

`close(f)` is no longer needed: it is done implicitly with the `end`.

Scope revisited

A variable that is defined at the top level is said to have **global scope**.

The `for`, `while`, and `try` blocks (but not the `if` blocks) all introduce a new scope. Variables defined in these blocks are only known to that scope. This is called the **local scope**, and nested blocks can introduce several levels of local scope. However, global variables are not accessible in `for` and `while` loops.

Variables with the same name in different scopes can safely be used simultaneously. If a variable exists both in global and local scope, you can decide which one you want to use by prefixing them with the `global` or `local` keyword:

- `global`: This indicates that you want to use the variable from the outer, global scope. This applies to the whole of the current scope block.
- `local`: This means that you want to define a new variable in the current scope.

The following example will clarify this, as follows:

```
# code in Chapter 4\scope.jl
x = 9
function funscope(n)
  x = 0 # x is in the local scope of the function
  for i = 1:n
    local x # x is local to the for loop
    x = i + 1
    if (x == 7)
        println("This is the local x in for: $x") #=> 7
    end
  end
  x
  println("This is the local x in funscope: $x") #=> 0
  global x = 15
end

funscope(10)
println("This is the global x: $x") #=> 15
```

This prints out the following result:

```
This is the local x in for: 7
This is the local x in funscope: 0
This is the global x: 15
```

If the `local` keyword was omitted from the `for` loop, the second `print` statement would print out `11` instead of `7`, as follows:

```
This is the local x in for: 7
This is the local x in funscope: 11
This is the global x: 15
```

What is the output when the `global x = 15` statement is left out? In this situation, the program prints out this result:

```
This is the local x in for: 7
This is the local x in funscope: 11
This is the global x: 9
```

However, needless to say, such name conflicts obscure the code and are a source of bugs, so try to avoid them if possible.

If you need to create a new local binding for a variable, use the `let` block. Execute the following code snippet:

```
anon = Array{Any}(undef, 2)
for i = 1:2
  anon[i] = ()-> println(i)
  i += 1
end
```

Here, both `anon[1]` and `anon[2]` are anonymous functions. When they are called with `anon[1]()` and `anon[2]()`, they print `2` and `3` (the values of `i` when they were created plus one). What if you wanted them to stick with the value of `i` at the moment of their creation? Then, you have to use `let` and change the code to the following:

```
anon = Array{Any}(undef, 2)
for i = 1:2
  let i = i
      anon[i] = ()-> println(i)
  end
  i += 1
end
```

Now, `anon[1]()` and `anon[2]()` print `1` and `2`, respectively. Because of `let`, they kept the value of `i` the same as when they were created.

The `let` statement also introduces a new scope. You can, for example, combine it with `begin`, like this:

```
begin
    local x = 1
    let
       local x = 2
       println(x) #> 2
    end
    x
    println(x) #> 1
end
```

`for` loops and comprehensions differ in the way they scope an iteration variable. When `i` is initialized to `0` before a `for` loop, after executing `for i = 1:10 end`, the variable `i` is now `10`:

```
i = 0
for i = 1:10
end
println(i)  #> 10
```

After executing a comprehension such as `[i for i = 1:10]`, the variable `i` is still `0`:

```
i = 0
[i for i = 1:10 ]
println(i)  #> 0
```

Tasks

Julia has a built-in system for running tasks, which are, in general, known as **coroutines**. With this, a computation that generates values into a `Channel` (with a `put!` function) can be suspended as a task, while a consumer task can pick up the values (with a `take!` function). This is similar to the `yield` keyword in Python.

As a concrete example, let's take a look at a `fib_producer` function that calculates the first 10 Fibonacci numbers (refer to the *Recursive functions* section in Chapter 3, *Functions*), but it doesn't return the numbers, it produces them:

```
# code in Chapter 4\tasks.jl
 function fib_producer(c::Channel)
        a, b = (0, 1)
        for i = 1:10
            put!(c, b)
            a, b = (b, a + b)
        end
    end
```

Construct a `Channel` by providing this function as an argument:

```
chnl = Channel(fib_producer)
```

The task's state is now runnable. To get the Fibonacci numbers, start consuming them with `take!` until `Channel` is closed, and the task is finished (state is `:done`):

```
take!(chnl) #> 1
take!(chnl) #> 1
take!(chnl) #> 2
take!(chnl) #> 3
take!(chnl) #> 5
take!(chnl) #> 8
take!(chnl) #> 13
take!(chnl) #> 21
take!(chnl) #> 34
take!(chnl) #> 55
take!(chnl) #> ERROR: InvalidStateException("Channel is closed.", :closed)
```

It is as if the `fib_producer` function was able to return multiple times, once for each `take!` call. Between calls to `fib_producer`, its execution is suspended, and the consumer has control.

The same values can be more easily consumed in a `for` loop, where the loop variable becomes one by one the produced values:

```
for n in chnl
    println(n)
end
```

This produces: `1 1 2 3 5 8 13 21 34 55`.

There is a macro `@task` that does the same thing:

```
chnl = @task fib_producer(c::Channel)
```

Coroutines are not executed in different threads, so they cannot run on separate CPUs. Only one coroutine is running at once, but the language runtime switches between them. An internal scheduler controls a queue of runnable tasks and switches between them based on events, such as waiting for data, or data coming in.

Here is another example, which uses `@async` to start a task asynchronously, binds a channel to the task, and then prints out the contents of the channel:

```
fac(i::Integer) = (i > 1) ? i*fac(i - 1) : 1
c = Channel(0)
task = @async foreach(i->put!(c,fac(i)), 1:5)
bind(c,task)
for i in c
   @show i
end
```

This prints out the following:

```
i = 1
i = 2
i = 6
i = 24
i = 120
```

Tasks should be seen as a form of cooperative multitasking in a single thread. Switching between tasks does not consume stack space, unlike normal function calls. In general, tasks have very low overhead; so you can use lots of them if needed. Exception handling in Julia is implemented using `Tasks` as well as servers that accept many incoming connections (refer to the *Working with TCP sockets and servers* section in `Chapter 8`, *IO, Networking, and Parallel Computing*).

True parallelism in Julia is discussed in the *Parallel operations and computing* section of `Chapter 8`, *IO, Networking, and Parallel Computing*.

Summary

In this chapter, we explored different control constructs, such as `if` and `while`. We also saw how to catch exceptions with `try` or `catch`, and how to throw our own exceptions. Some subtleties of scope were discussed, and finally, we got an overview of how to use coroutines in Julia with tasks. Now we are well-equipped to explore more complex types that consist of many elements. This is the topic of the next chapter, *Collection Types*.

Collection Types

Collections of values appear everywhere in programs, and Julia has the most important built-in collection types. In `Chapter 2`, *Variables, Types, and Operations*, we introduced two important types of collection: **arrays** and **tuples**. In this chapter, we will look more deeply into multidimensional arrays (or matrices), and into the tuple type as well. A dictionary type, where you can look up a value through a key, is indispensable in a modern language, and Julia has this too. Finally, we will explore the set type. Like arrays, all these types are parameterized; the type of their elements can be specified at the time of object construction.

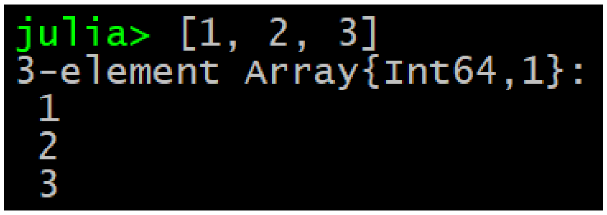
Collections are also iterable types, the types over which we can loop with `for` or an iterator producing each element of the collection successively. The iterable types include string, range, array, tuple, dictionary, and set.

So, the following are the topics for this chapter:

- Matrices
- Tuples
- Dictionaries
- Sets
- An example project—word frequency

Matrices

We know that the notation `[1, 2, 3]` is used to create an array. In fact, this notation denotes a special type of array, called a (column) **vector** in Julia, as shown in the following screenshot:



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

To create this as a row vector (`1 2 3`), use the notation `[1 2 3]` with spaces instead of commas. This array is of type `1 x 3 Array{Int64,2}`, so it has two dimensions. (The spaces used in `[1, 2, 3]` are for readability only, we could have written this as `[1,2,3]`).

A matrix is a two- or multidimensional array (in fact, a matrix is an alias for the two-dimensional case). We can write this as follows:

```
Array{Int64, 1} == Vector{Int64} #> true
Array{Int64, 2} == Matrix{Int64} #> true
```

As matrices are so prevalent in data science and numerical programming, Julia has an amazing range of functionalities for them.

To create a matrix, use space-separated values for the columns and semicolon-separated for the rows:

```
// code in Chapter 5\matrices.jl:
matrix = [1 2; 3 4]
    2x2 Array{Int64,2}:
    1  2
    3  4
```

So, the column vector from the beginning can also be written as `[1; 2; 3]`. However, you cannot use commas and semicolons together.

To get the value from a specific element in the matrix, you need to index it by row and then by column, for example, `matrix[2, 1]` returns the value `3` (second row, first column).

Using the same notation, one can calculate products of matrices such as `[1 2] * [3 ; 4]`; this is calculated as `[1 2] * [3 4]`, which returns the value `11` (which is equal to `1*3 + 2*4`). In contrast to this, conventional matrix multiplication is defined with the operator `.*`:

```
[1 2] .* [3 ; 4]
# 2 Array{Int64,2}:
#   3  6
#   4  8
```

To create a matrix from random numbers between 0 and 1, with three rows and five columns, use `ma1 = rand(3, 5)`, which shows the following results:

```
3x5 Array{Float64,2}:
 0.0626778  0.616528  0.60699   0.709196  0.900165
 0.511043   0.830033  0.671381  0.425688  0.0437949
 0.0863619  0.621321  0.78343   0.908102  0.940307
```

The `ndims` function can be used to obtain the number of dimensions of a matrix. Consider the following example:

```
 julia> ndims(ma1) #> 2
 julia> size(ma1) #> a tuple with the dimensions (3, 5)
```

To get the number of rows (`3`), run the following command:

```
julia>   size(ma1,1) #> 3
```

The number of columns (`5`) is given by:

```
julia> size(ma1,2) #> 5
julia> length(ma1) #> 15, the number of elements
```

That's why you will often see this: `nrows, ncols = size(ma)`, where `ma` is a matrix, `nrows` is the number of rows, and `ncols` is the number of columns.

If you need an identity matrix, where all the elements are zero, except for the elements on the diagonal that are `1.0`, use the `I` function (from the `LinearAlgebra` package) with the argument `3` for a 3 x 3 matrix:

```
using LinearAlgebra
  idm = Matrix(1.0*I, 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 easily work with parts of a matrix, known as **slices**; these are similar to those used in Python and NumPy as follows:

- `idm[1:end, 2]` or shorter `idm[:, 2]` returns the entire second column
- `idm[2, :]` returns the entire second row
- `idmc = idm[2:end, 2:end]` returns the output as follows:

```
        2x2 Array{Float64,2}
            1.0  0.0
            0.0  1.0
```

- `idm[2, :] .= 0` sets the entire second row to `0`
- `idm[2:end, 2:end] = [ 5 7 ; 9 11 ]` will change the matrix as follows:

```
        1.0  0.0  0.0
        0.0  5.0  7.0
        0.0  9.0  11.0
```

Slicing operations create views into the original array rather than copying the data, so a change in the slice changes the original array or matrix.

Any multidimensional matrix can also be seen as a one-dimensional vector in column order, as follows:

```
a = [1 2;3 4]
2 Array{Int64,2}:
 1  2
 3  4

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

To make an array of arrays (a **jagged** array), use an `Array` initialization, and then `push!` each array in its place, for example:

```
jarr = (Array{Int64, 1})[]
push!(jarr, [1,2])
push!(jarr, [1,2,3,4])
push!(jarr, [1,2,3])
#=>
3-element Array{Array{Int64,1},1}:
 [1,2]
 [1,2,3,4]
 [1,2,3]
```

If `ma` is a matrix, say `[1 2; 3 4]`, then `ma'` is the transpose matrix, that is `[1 3; 2 4]`:

```
ma:   1  2           ma'  1   3
      3  4                2   4
```

`ma'` is an operator notation for the `transpose(ma)` function.

Multiplication is defined between matrices, as in mathematics, so `ma * ma'` returns the 2 x 2 matrix or type `Array{Int64,2}` as follows:

```
5    11
11   25
```

If you need element-wise multiplication, use `ma .* ma'`, which returns `2 x 2 Array{Int64,2}`:

```
1   6
6  16
```

The inverse of a matrix `ma` (if it exists) is given by the `inv(ma)` function. The `inv(ma)` function returns `2 x 2 Array{Float64,2}`:

```
-2.0   1.0
1.5   -0.5
```

The inverse means that `ma * inv(ma)` produces the identity matrix:

```
1.0   0.0
0.0   1.0
```

Trying to take the inverse of a singular matrix (a matrix that does not have a well-defined inverse) will result in `LAPACKException` or `SingularException`, depending on the matrix type. Suppose you want to solve the `ma1 * X = ma2` equation, where `ma1`, `X`, and `ma2` are matrices. The obvious solution is `X = inv(ma1) * ma2`. However, this is actually not that good. It is better to use the built-in solver, where `X = ma1 \ ma2`. If you have to solve the `X * ma1 = ma2` equation, use the solution `X = ma2 / ma1`. Solutions that use `/` and `\` are much more numerically stable, and also much faster.

If `v = [1.,2.,3.]` and `w = [2.,4.,6.]`, and you want to form a 3 x 2 matrix with these two column vectors, then use `hcat(v, w)` (for horizontal concatenation) to produce the following output:

```
1.0  2.0
2.0  4.0
3.0  6.0
```

`vcat(v,w)` (for vertical concatenation) results in a one-dimensional array with all the six elements with the same result as `append!(v, w)`.

Thus, `hcat` concatenates vectors or matrices along the second dimension (columns), while `vcat` concatenates along the first dimension (rows). The more general `cat` can be used to concatenate multidimensional arrays along arbitrary dimensions.

There is an even simpler literal notation: to concatenate two matrices `a` and `b` with the same number of rows to a matrix `c`, just execute `c = [a b]`. now `b` is appended to the right of `a`. To put `b` beneath `c`, use `c = [a; b]`. The following is a concrete example, `a = [1 2; 3 4]`and `b = [5 6; 7 8]`:

`a`	`b`	`c = [a b]`	`c = [a; b]`

`1 2`
`3 4`
	`5 6`
	`7 8`
		`1 2 5 6`
		`3 4 7 8`
			`1 2`
			`3 4`
			`5 6`
			`7 8`

The `reshape` function changes the dimensions of a matrix to new values if this is possible, for example:

```
reshape(1:12, 3, 4) #> returns a 3x4 array with the values 1 to 12
3x4 Array{Int64,2}:
 1  4  7  10
 2  5  8  11
 3  6  9  12

a = rand(3, 3)  #> produces a 3x3 Array{Float64,2}
3x3 Array{Float64,2}:
 0.332401   0.499608  0.355623
 0.0933291  0.132798  0.967591
 0.722452   0.932347  0.809577

reshape(a, (9,1)) #> produces a 9x1 Array{Float64,2}:
9x1 Array{Float64,2}:
 0.332401
 0.0933291
 0.722452
 0.499608
 0.132798
 0.932347
 0.355623
 0.967591
 0.809577

reshape(a, (2,2)) #> does not succeed:
ERROR: DimensionMismatch("new dimensions (2,2) must be consistent
  with array size 9")
```

When working with arrays that contain arrays, it is important to realize that such an array contains references to the contained arrays, not their values. If you want to make a copy of an array, you can use the `copy()` function, but this produces only a *shallow copy* with references to the contained arrays. In order to make a complete copy of the values, we need to use the `deepcopy()` function.

The following example makes this clear:

```
x = Array{Any}(undef, 2) #> 2-element Array{Any,1}: #undef #undef
x[1] = ones(2) #> 2-element Array{Float64} 1.0 1.0
x[2] = trues(3) #> 3-element BitArray{1}: true true true
x #> 2-element Array{Any,1}: [1.0,1.0] Bool[true,true,true]
a = x
b = copy(x)
c = deepcopy(x)
# Now if we change x:
x[1] = "Julia"
x[2][1] = false
x #> 2-element Array{Any,1}: "Julia" Bool[false,true,true]
a #> 2-element Array{Any,1}: "Julia" Bool[false,true,true]
isequal(a, x) #> true, a is identical to x
b #> 2-element Array{Any,1}: [1.0,1.0] Bool[false,true,true]
isequal(b, x) #> false, b is a shallow copy of x
c #> 2-element Array{Any,1}: [1.0,1.0] Bool[true,true,true]
isequal(c, x) #> false
```

The value of `a` remains identical to `x` when this changes, because it points to the same object in the memory. The deep copy `c` function remains identical to the original `x`. The `b` value retains the changes in a contained array of `x`, but not if one of the contained arrays becomes another array.

To further increase performance, consider using the statically-sized and immutable vectors and matrices from the `ImmutableArrays` package, which is a lot faster, certainly for small matrices, and particularly for vectors.

Tuples

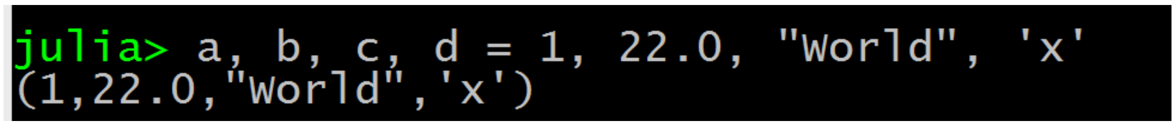
A **tuple** is a fixed-sized group of values, separated by commas and optionally surrounded by parentheses `( )`. The type of these values can be the same, but it doesn't have to be; a tuple can contain values of different types, unlike arrays. A tuple is a heterogeneous container, whereas an array is a homogeneous container. The type of a tuple is just a tuple of the types of the values it contains. So, in this sense, a tuple is very much the counterpart of an array in Julia. Also, changing a value in a tuple is not allowed; tuples are immutable.

In `Chapter 2`, *Variables, Types, and Operations*, we saw fast assignment, which is made possible by tuples:

```
// code in Chapter 5\tuples.jl:
a, b, c, d = 1, 22.0, "World", 'x'
```

This expression assigns `a` value `1`, `b` becomes `22.0`, `c` takes up the value `World`, and `d` becomes `x`.

The expression returns a tuple `(1, 22.0,"World",'x')`, as the REPL shows as follows:



```
julia> a, b, c, d = 1, 22.0, "World", 'x'
(1,22.0,"World",'x')
```

If we assign this tuple to a variable `t1` and ask for its type, we get the following result:

```
typeof(t1) #> Tuple{Int64,Float64,String,Char}
```

The argument list of a function (refer to the *Defining functions* section in `Chapter 3`, Functions) is, in fact, also a tuple. Similarly, Julia simulates the possibility of returning multiple values by packaging them into a single tuple, and a tuple also appears when using functions with variable argument lists. `( )` represents the empty tuple, and `(1,)` is a one-element tuple. The type of a tuple can be specified explicitly through a type annotation (refer to the *Types* section in Chapter 2, *Variables, Types, and Operations*), such as `('z', 3.14)::Tuple{Char, Float64}`.

The following snippet shows that we can index tuples in the same way as arrays by using brackets, indexing starting from `1`, slicing, and index control:

```
t3 = (5, 6, 7, 8)
t3[1] #> 5
t3[end] #> 8
t3[2:3] #> (6, 7)
t3[5] #> BoundsError: attempt to access (5, 6, 7, 8) at index [5]
t3[3] = 9 #> Error: 'setindex' has no matching ...
author = ("Ivo", "Balbaert", 62)
author[2] #> "Balbaert"
```

To iterate over the elements of a tuple, use a `for` loop:

```
for i in t3
    println(i)
end # #> 5  6  7  8
```

A tuple can be *unpacked* or deconstructed like this: `a, b = t3`; now `a` is `5` and `b` is `6`. Notice that we don't get an error despite the left-hand side not being able to take all the values of `t3`. To do this, we would have to write `a, b, c, d = t3`.

In the preceding example, the elements of the `author` tuple are unpacked into separate variables: `first_name, last_name,` and `age = author`.

So, tuples are nice and simple types that make a lot of things possible. We'll find them again in the next section as elements of a dictionary.

Dictionaries

When you want to store and look up values based on a unique key, then the dictionary type `Dict` (also called hash, associative collection, or map in other languages) is what you need. It is basically a collection of two-element tuples of the form `(key, value)`. To define a dictionary `d1` as a literal value, the following syntax is used:

```
// code in Chapter 5\dicts.jl:
d1 = Dict(1 => 4.2, 2 => 5.3)
```

It returns `Dict{Int64,Float64}` with two entries: `2 => 5.3` and `1 => 4.2`, so there are two key-value tuples here, `(1, 4.2)` and `(2, 5.3)`; the key appears before the `=>` symbol and the value appears after it, and the tuples are separated by commas.

To explicitly specify the types, use:

```
d1 = Dict{Int64,Float64}(1 => 4.2, 2 => 5.3)
```

If you use the former `[]` notation to try to define a dictionary, you now get `Array{Pairs{}}` instead:

```
d1 = [1 => 4.2, 2 => 5.3]
# 2-element Array{Pair{Int64,Float64},1}:
# 1 => 4.2
# 2 => 5.3
```

Here are some other examples:

```
d2 = Dict{Any,Any}("a"=>1, (2,3)=>true)
d3 = Dict(:A => 100, :B => 200)
```

The `Any` type is inferred when a common type among the keys or values cannot be detected.

So a `Dict` can have keys of different types, and the same goes for the values: their type is then indicated as `Any`. In general, dictionaries that have type `{Any, Any}` tend to lead to lower performance since the JIT compiler does not know the exact type of the elements. Dictionaries used in performance-critical parts of the code should therefore be explicitly typed. Notice that the `(key, value)` pairs are not returned (or stored) in the key order.

If the keys are of the `Char` or `String` type, you can also use `Symbol` as the key type, which could be more appropriate since `Symbols` are immutable, for example:

```
d3 = Dict{Symbol,Int64}(:A => 100, :B => 200)
```

Use the bracket notation, with a key as an index, to get the corresponding value: `d3[:B]` returns `200`. However, the key must exist, otherwise we will get an error, `d3[:Z]`, that returns `ERROR: KeyError: key not found: :Z`. To get around this, use the `get` method and provide a default value that is returned instead of the error, `get(d3, :Z, 999)` returns `999`.

Here is a dictionary that resembles an object, storing the field names as symbols in the keys:

```
dmus = [ :first_name => "Louis", :surname => "Armstrong",
    :occupation => "musician", :date_of_birth => "4/8/1901" ]
```

To test if a key is present in `Dict`, you can use the function `haskey` as follows:

- `haskey(d3, :Z)` returns `false`
- `haskey(d3, :B)` returns `true`

Dictionaries are mutable. If we tell Julia to execute `d3[:A] = 150`, then the value for key `:A` in `d3` has changed to `150`. If we do this with a new key, then that tuple is added to the dictionary:

```
d3[:C] = 300
```

`d3` is now `Dict(:A=>150,:B=>200,:C=>300)`, and it has three elements: `length(d3)` returns `3`.

`d4 = Dict()` is an empty dictionary of type `Any`, and you can start populating it in the same way as in the example with `d3`.

`d5 = Dict{Float64, Int64}()` is an empty dictionary with key type `Float64` and value type `Int64`. As to be expected, adding keys or values of another type to a typed dictionary is an error. `d5["c"] = 6` returns `ERROR: MethodError 'convert' has no method matching convert(::Type{Float64}, ::ASCIIString)` and `d3["CVO"] = 500` returns `ERROR: ArgumentError: CVO is not a valid key for type Symbol`.

Deleting a key mapping from a collection is also straightforward. `delete!(d3, :B)` removes `(:B, 200)` from the dictionary, and returns the collection that contains only `:A => 100`.

Keys and values – looping

To isolate the keys of a dictionary, use the `keys` function `ki = keys(d3)`, with `ki` being a `KeyIterator` object, which we can use in a `for` loop as follows:

```
for k in keys(d3)
    println(k)
end
```

Assuming `d3` is again `d3 = Dict(:A => 100, :B => 200)`, this prints out `A` and `B`. This also gives us an alternative way to test if a key exists with `in`. For example, `:A in keys(d3)` returns `true` and `:Z in keys(d3)` returns `false`.

If you want to work with an array of keys, use `collect(keys(d3))`, which returns a two-element `Array{Symbol,1}` that contains `:A` and `:B`. To obtain the values, use the `values` function: `vi = values(d3)`, with `vi` being a `ValueIterator` object, which we can also loop through with `for`:

```
for v in values(d3)
    println(v)
end
```

This returns `100` and `200`, but the order in which the values or keys are returned is undefined.

Creating a dictionary from arrays with `keys` and `values` is trivial because we have a `Dict` constructor that can use these; as in the following example:

```
keys1 = ["J.S. Bach", "Woody Allen", "Barack Obama"] and
values1 =  [ 1685, 1935, 1961]
```

Then, `d5 = Dict(zip(keys1, values1))` results in a `Dict{String,Int64}` with three entries as follows:

```
"J.S. Bach"    => 1685
"Woody Allen"  => 1935
"Barack Obama" => 1961
```

Working with both the `key` and `value` pairs in a loop is also easy. For instance, the `for` loop over `d5` is as follows:

```
for (k, v) in d5
       println("$k was born in $v")
   end
```

This will print the following output:

```
J.S. Bach was born in 1685
Barack Obama was born in 1961
Woody Allen was born in 1935
```

Alternatively, we can use an index in the tuple:

```
for p in d5
  println("$(p[1]) was born in $(p[2])")
end
```

Here are some more neat tricks, where `dict` is a dictionary:

- Copying the keys of a dictionary to an array with a list comprehension:

```
        arrkey = [key for (key, value) in dict]
```

  This is the same as `collect(keys(dict))`.

- Copying the values of a dictionary to an array with a list comprehension:

```
        arrval = [value for (key, value) in dict]
```

  This is the same as `collect(values(dict))`

Sets

Array elements are ordered, but can contain duplicates, that is, the same value can occur at different indices. In a dictionary, keys have to be unique, but the values do not, and the keys are not ordered. If you want a collection where order does not matter, but where the elements have to be unique, then use a **Set**. Creating a set is as easy as this:

```
// code in Chapter 5\sets.jl:
s = Set([11, 14, 13, 7, 14, 11])
```

The `Set()` function creates an empty set `Set(Any[])`. The preceding line returns `Set([7, 14, 13, 11])`, where the duplicates have been eliminated.

Operations from the set theory are also defined for `s1 = Set([11, 25])` and `s2 = Set([25, 3.14])` as follows:

- `union(s1, s2)` produces `Set([3.14,25,11])`
- `intersect(s1, s2)` produces `Set([25])`
- `setdiff(s1, s2)` produces `Set{Any}([11])`, whereas `setdiff(s2, s1)` produces `Set([ 3.14])`
- `issubset(s1, s2)` produces `false`, but `issubset(s1, Set([11, 25, 36]))` produces `true`

To add an element to a set is easy: `push!(s1, 32)` adds `32` to set `s1`. Adding an existing element will not change the set. To test whether a set contains an element, use `in`. For example, `in(32, s1)` returns true and `in(100, s1)` returns false.

`Set([1,2,3])` produces a set of integers `Set([2,3,1])` of the `Set{Int64}` type. To get a set of arrays, use `Set([[1,2,3]])`, which returns `Set(Array{Int64,1}[[1, 2, 3]])`.

Sets are commonly used when we need to keep track of objects in no particular order. For instance, we might be searching through a graph. We can then use a set to remember which nodes of the graph we have already visited in order to avoid visiting them again. Checking whether an element is present in a set is independent of the size of the set. This is extremely useful for very large sets of data, for example:

```
x = Set(collect(1:100))
@time 2 in x
#> 0.003186 seconds (33 allocations: 2.078 KiB)
x2 = Set(collect(1:1000000))
@time 2 in x2
# 0.000003 seconds (4 allocations: 160 bytes)
```

The second statement executes much faster using much less memory, despite the fact that `x2` is four orders of magnitude larger than `x`.

Take a look at the `Collections` module if you need more specialized containers. It contains a priority queue as well as some lower-level heap functions.

An example project – word frequency

A lot of the concepts and techniques that we have seen so far in this book come together in this little project. Its aim is to read a text file, remove all characters that are not used in words, and count the frequency of the words in the remaining text. This can be useful, for example, when counting the word density on a web page, the frequency of DNA sequences, or the number of hits on a website that came from various IP addresses. This can be done in some ten lines of code. For example, when `words1.txt` contains the sentence `to be, or not to be, that is the question!`, then this is the output of the program:

```
Word : frequency

be : 2
is : 1
not : 1
or : 1
question : 1
that : 1
the : 1
to : 2
```

Here is the code with comments:

```
# code in chapter 5\word_frequency.jl:
# 1- read in text file:
str = read("words1.txt", String)
# 2- replace non alphabet characters from text with a space:
nonalpha = r"(\W\s?)" # define a regular expression
str = replace(str, nonalpha => ' ')
digits = r"(\d+)"
str = replace(str, digits => ' ')
# 3- split text in words:
word_list = split(str, ' ')
# 4- make a dictionary with the words and count their frequencies:
word_freq = Dict{String, Int64}()
for word in word_list
    word = strip(word)
    if isempty(word) continue end
    haskey(word_freq, word) ?
      word_freq[word] += 1 :
      word_freq[word] = 1
end
```

```
# 5- sort the words (the keys) and print out the frequencies:
println("Word : frequency \n")
words = sort!(collect(keys(word_freq)))
for word in words
    println("$word : $(word_freq[word])")
end
```

The `strip()` function removes white space from a string at the front/back.

The `isempty` function is quite general and can be used on any collection.

Try the code out with the example text files `words1.txt` or `words2.txt`. See the output in `results_words1.txt` and `results_words2.txt`.

Summary

In this chapter, we looked at the built-in collection types Julia has to offer. We saw the power of matrices, the elegance of dictionaries, and the usefulness of tuples and sets. However, to dig deeper into the fabric of Julia, we need to learn how to define new types, which is another concept necessary that we need to organize code. We must know how types can be constructed, and how they are used in multiple dispatch. This is the main topic of the next chapter, where we will also see modules, which serve to organize code, but at an even higher level than types.

More on Types, Methods, and Modules

Julia has a rich built-in type system, and most data types can be parameterized, such as `Array{Float64, 2}` or `Dict{Symbol, Float64}`. Typing a variable (or more exactly the value it is bound to) is optional. However, indicating the type of some variables, although they are not statically checked, can provide some of the advantages of static-type systems as in C++, Java, or C#. A Julia program can run without any indication of types, which can be useful in a prototyping stage, and it will still run fast. However, some type indications can increase the performance by allowing more specialized multiple dispatch. Type assertions also help the LLVM compiler to create more compact, better optimized code. Moreover, typing function parameters makes the code easier to read and understand. The robustness of the program is also enhanced by throwing exceptions, in cases where certain type operations are not allowed. These failures will manifest themselves during testing, or the code can provide an exception handling mechanism.

All functions in Julia are inherently generic or polymorphic, that is, they can operate on different types of their arguments. The most appropriate method (an implementation of the function where argument types are indicated) will be chosen at runtime to be executed, depending on the type of arguments passed to the function. As we will see in this chapter, you can also define your own types, and Julia provides a limited form of abstract types and subtyping.

A lot of these topics have already been discussed in previous chapters; for example, refer to the *Generic functions and multiple dispatch* section in `Chapter 3`, *Functions*. In this chapter, we broaden the previous discussions by covering the following topics:

- Type annotations

- The type hierarchy—subtypes and supertypes
- Concrete and abstract types
- User-defined and composite types
- Types and collections—inner constructors
- Type unions
- Parametric types and methods
- Standard modules and paths

Type annotations

As we saw in `Chapter 2`, *Variables, Types, and Operations*, type-annotating a variable is done with the `::` operator, such as in the function definition `function write(io::IO, s::String) #... end`, where the parameter `io` has to be of type `IO`, and `s` of type `String`. To put it differently, `io` has to be an instance of type `IO`, and `s` an instance of type `String`. The `::` operator is, in fact, an assertion that affirms that the value on the left is of the type on the right. If this is not true, a `typeassert` error is thrown. Try this out in the REPL:

```
# see the code in Chapter 6\conversions.jl:
(31+42)::Float64
```

We get an `ERROR: TypeError: in typeassert, expected Float64, got Int64` error message.

This is, in addition to the method specialization for multiple dispatch, an important reason why type annotations are used in function signatures.

The operator `::` can also be used in the sense of a type declaration, but only in local scope, such as in functions, as follows:

```
n::Int16 or local n::Int16 or n::Int16 = 5
```

Every value assigned to `n` will be implicitly converted to the indicated type with the `convert` function.

Type conversions and promotions

The `convert` function can also be used explicitly in the code as `convert(Int64, 7.0)`, which returns `7`.

In general, `convert(Type, x)` will attempt to put the `x` value in an instance of `Type`. In most cases, `type(x)` will also do the trick, as in `Int64(7.0)`, returning `7`.

The conversion, however, doesn't always work:

- When precision is lost—`Int64(7.01)`returns an `ERROR: InexactError()` error message
- When the target type is incompatible with the source value—`convert(Int64, "CV")` returns an ``ERROR: MethodError: Cannot `convert` an object of type String to an object of type Int64`` error message

This last error message really shows us how multiple dispatch works; the types of the input arguments are matched against the methods available for that function.

We can define our own conversions by providing new methods for the `convert` function. For example, for information on how to do this, refer to `http://docs.julialang.org/en/latest/manual/conversion-and-promotion/#conversion`.

Julia has a built-in system called **automatic type promotion** to promote arguments of mathematical operators and assignments to a common type: in `4 + 3.14`, the integer `4` is promoted to a `Float64` value, so that the addition can take place and results in `7.140000000000001`. In general, promotion refers to the conversion of values of different types to one common type. This can be done with the `promote` function, which takes a number of arguments, and returns a tuple of the same values, converting them to a common type. An exception is thrown if promotion is not possible. Some examples are as follows:

- `promote(1, 2.5, 3//4)` returns `(1.0, 2.5, 0.75)`

- `promote(1.5, im)` returns `(1.5 + 0.0im, 0.0 + 1.0im)`
- `promote(true, 1.0)` returns `(1.0, 1.0)`

Thanks to the automatic type promotion system for numbers, Julia doesn't have to define, for example, the `+` operator for any combinations of numeric types. Instead, it is defined as `+(x::Number, y::Number) = +(promote(x,y)...)`.

It basically says: first, promote the arguments to a common type, and then perform the addition. `Number` is a common supertype for all values of numeric types. To determine the common promotion type of the two types, use `promote_type(Int8, UInt16)` to find whether it returns `UInt16`.

This is because, somewhere in the standard library, the following `promote_rule` function was defined as `promote_rule(::Type{Int8}, ::Type{Uint16}) = UInt16`.

You can take a look at how promoting is defined in the source code Julia in `base/promotion.jl`. These kinds of promotion rules can be defined for your own types too if needed.

The type hierarchy – subtypes and supertypes

In Julia, every value has a type, for example, `typeof(2)` is `Int64` (or `Int32` on 32-bit systems). Julia has a lot of built-in types, in fact, a whole hierarchy starting from the type `Any` at the top. Every type in this structure also has a type, namely, `DataType`, so it is very consistent. `typeof(Any)`, `typeof(Int64)`, `typeof(Complex{Int64})`, and `typeof(DataType)` all return `DataType`. So, types in Julia are also objects; all concrete types, except tuple types, which are a tuple of the types of its arguments, are of type `DataType`.

Follow along with the code in `type_hierarchy.jl`.

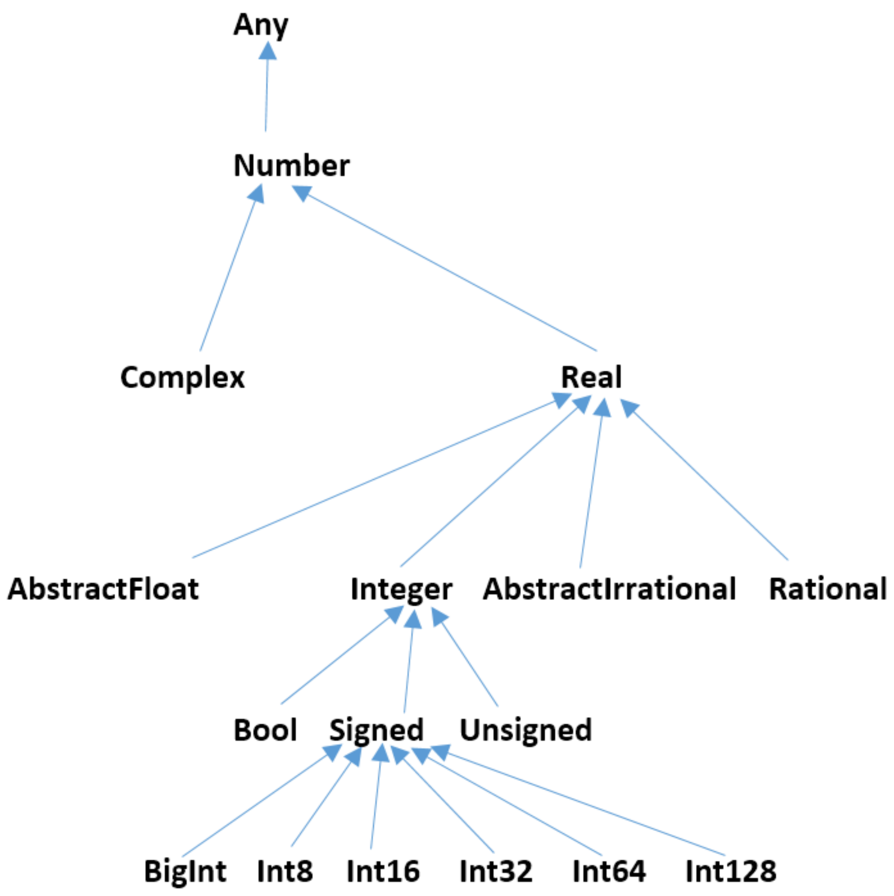
This type hierarchy is like a tree; each type has one parent given by the `supertype` function:

- `supertype(Int64)` returns `Signed`
- `supertype(Signed)` returns `Integer`
- `supertype(Integer)` returns `Real`
- `supertype(Real)` returns `Number`
- `supertype(Number)` returns `Any`
- `supertype(Any)` returns `Any`

A type can have a lot of children or `subtypes` (a function from the `InteractiveUtils` package) as follows:

- `subtypes(Integer)` form `3-element Array{Any,1}`, which contains `Bool`, `Signed`, and `Unsigned`
- `subtypes(Signed)` form `6-element Array{Any,1}`, which contains `BigInt, Int128`, `Int16`, `Int32`, `Int64`, and `Int8`
- `subtypes(Int64)` is `0-element Array{Any,1}`, which has no subtypes

To indicate the subtype relationship, the operator `<` is used: `Bool <: Integer` and `Bool <: Any` returns true, while `Bool <: Char` is false. The following is a visualization of part of this type tree:



Concrete and abstract types

In this hierarchy, some types, such as `Number`, `Integer`, and `Signed`, are abstract, which means that they have no concrete objects or values of their own. Instead, objects or values are of concrete types given by the result of applying `typeof(value)`, such as `Int8`, `Float64`, and `String`. For example, the concrete type of the value `5` is `Int64` given by `typeof(5)` (on a 64-bit machine). However, a value also has the type of all of its supertypes; for example, `isa(5, Number)` returns true (we introduced the `isa` function in the *Types* section of `Chapter 2`, *Variables, Types, and Operations*).

Concrete types have no subtypes and might only have abstract types as their supertypes. Schematically, we can differentiate them as follows:

Type	**Instantiate**	**Subtypes**
`concrete`	`Y`	`N`
`abstract`	`N`	`Y`

An abstract type (such as `Number` and `Real`) is only a name that groups multiple subtypes together, but it can be used as a type annotation or used as a type in array literals. These types are the nodes in the type hierarchy that mainly serve to support the type tree. Also, note that an abstract type cannot have any fields.

The abstract type `Any` is the supertype of all types, and all the objects are also instances of `Any`.

At the other end is `Union{}`: this type has no values and no subtypes. All types are supertypes of `Union{}` , and no object is an instance of `Union{}`. It is unlikely that you will ever have to use this type. The `Nothing` type has one value called `nothing`.

When a function is only used for its side-effects, convention dictates that it returns `nothing`. We have seen this with the `println` function, where the printing is the side-effect, for instance:

```
x = println("hello") #> hello
x == nothing #> true
```

User-defined and composite types

In Julia, as a developer you can define your own types to structure data used in applications. For example, if you need to represent points in a three-dimensional space, you can define a type `Point`, as follows:

```
# see the code in Chapter 6\user_defined.jl:
mutable struct Point
    x::Float64
    y::Float64
    z::Float64
end
```

`mutable` here means that `Point` values can be modified. If your type values cannot be changed, simply use `struct`.

The type `Point` is a concrete type. Objects of this type can be created as `p1 = Point(2, 4, 1.3)`, and it has no subtypes: `typeof(p1)` returns `Point (constructor with 2 methods)`, `subtypes(Point)`returns `0-element Array{Any,1}`.

Such a user-defined type is composed of a set of named fields with an optional type annotation; that's why it is a composite type, and its type is also `DataType`. If the type of a named field is not given, then it is `Any`. A composite type is similar to `struct` in C, or a class without methods in Java.

Unlike in other object-oriented languages such as Python or Java, where you call a function on an object such as `object.func(args)`, Julia uses the `func(object, args)` syntax.

Julia has no classes (as types with functions belong to that type); this keeps the data and functions separate. Functions and methods for a type will be defined outside that type. Methods cannot be tied to a single type, because multiple dispatch connects them with different types. This provides more flexibility, because when adding a new method for a type, you don't have to change the code of the type itself, as you would have to do with the code of the class in object-oriented languages.

The names of the fields that belong to a composite type can be obtained with the `names` function of the type or object: `fieldnames(Point)` or `fieldnames(typeof(p1))` returns `(:x, :y, :z)`.

A user-defined type has two default implicit *constructors* that have the same name as the type and take an argument for each field. You can see this by asking for the methods of `Point`: `methods(Point)`; it returns two methods for generic function `Point`: `Point(x::Float64, y::Float64, z::Float64)` and `Point(x ,y ,z)`. Here, the field values can be of type `Any`.

You can now make objects simply like this:

```
orig = Point(0, 0, 0)
p1 = Point(2, 4, 1.3).
```

Fields that together contain the state of the object can be accessed by name, as in: `p1.y`, which returns `4.0`.

Objects of such a type are mutable, for example, I can change the `z` field to a new value with `p1.z = 3.14`, resulting in `p1` now having the value `Point(2.0, 4.0, 3.14)`. Of course, types are checked: `p1.z = "A"` results in an error.

Objects as arguments to functions are *passed by reference*, so that they can be changed inside the function (for example, refer to the function `insert_elem(arr)` in the *Defining types* section of `Chapter 3`, *Functions*).

If you don't want your objects to be changeable, replace `type` with the keyword `struct`, for example:

```
struct Vector3D
    x::Float64
    y::Float64
    z::Float64
end
```

Calling `p = Vector3D(1, 2, 3)` returns `Vector3D(1.0, 2.0, 3.0)` and `p.y = 5` returns `ERROR: type Vector3D is immutable`.

Immutable types enhance performance, because Julia can optimize the code for them. Another big advantage of immutable types is thread safety: an immutable object can be shared between threads without needing synchronization. If, however, such an *immutable type contains a mutable field such as an array, the contents of that field can be changed. So, define your immutable types without mutable fields.*

A type once defined cannot be changed. If we try to define a new type `Point` with fields of type `Int64`, or with added fields, we get an `ERROR: invalid redefinition of constant TypeName` error message.

A new type that is exactly the same as an existing type can be defined as an *alias* through a simple assignment, for instance, `Point3D = Point`. Now, objects of type `Point3D` can also be created: `p31 = Point3D(1, 2, 3)` returns `Point(1.0, 2.0, 3.0)`. Julia also uses this internally; the alias `Int` is used for either `Int64` or `Int32`, depending on the architecture of the system that is being used.

When are two values or objects equal or identical?

Whether two values are equal or not can be decided by the `==` operator, for example, `5 == 5` and `5 == 5.0` are both `true`. Equivalent to this operator is the `isequal()` function:

```
isequal(5, 5)   #> true
isequal(5, 5.0) #> true
```

Both the preceding statements return `true`, because objects such as numbers are immutable and they are compared at the bits level.

To see whether the two objects `x` and `y` are identical, they must be compared with the `===` operator. The result is a `Bool` value, `true` or `false`: `x === y -> Bool`, for example:

```
5 === 5 #> true
5 === 5.0 #> false
```

For objects that are more complex, such as strings, arrays, or objects that are constructed from composite types, the addresses in the memory are compared to check whether they point to the same memory location. For immutable object such as `struct`, this gets optimized so that instances with the same value point to the same object:

```
struct Vector3D
    x::Float64
    y::Float64
    z::Float64
end

q = Vector3D(4.0, 3.14, 2.71)
r = Vector3D(4.0, 3.14, 2.71)
isequal(q, r) #> true
q === r #> true
```

However, if objects are mutable, they are different objects even if they have the same value, as follows:

```julia
mutable struct MVector3D
    x::Float64
    y::Float64
    z::Float64
end

q = MVector3D(4.0, 3.14, 2.71)
r = MVector3D(4.0, 3.14, 2.71)
isequal(q, r) #> false
q === r #> false
```

A multiple-dispatch example

Let's now explore an example about people working in a company to show multiple dispatch in action. Let's define an abstract type `Employee` and a type `Developer` that is a subtype:

```
abstract type Employee
end

mutable struct Developer <: Employee
    name::String
    iq
    favorite_lang::String
end
```

We cannot make objects from an abstract type: calling `Employee()` only returns an `ERROR: MethodError: no constructors have been defined for Employee` error message.

The type `Developer` has two implicit constructors, but we can define another outer constructor, which uses a default constructor as follows:

```
Developer(name, iq) = Developer(name, iq, "Java")
```

Outer constructors provide additional convenient methods to construct objects. Now, we can make the following two developer objects:

- `devel1 = Developer("Bob", 110)`, which returns `Developer("Bob",110,"Java")`
- `devel2 = Developer("William", 145, "Julia")`, which returns `Developer("William",145,"Julia")`

Similarly, we can define a type `Manager` and an instance of it as follows:

```
mutable struct Manager
    name::String
    iq
    department::String
end
man1 = Manager("Julia", 120, "ICT")
```

Concrete types, such as `Developer` or `Manager`, cannot be subtyped:

```
mutable struct MobileDeveloper <: Developer
  platform
end
```

This returns `ERROR: invalid subtyping in definition of MobileDeveloper`.

If we now define a function `cleverness` as `cleverness(emp::Employee) = emp.iq`, then `cleverness(devel1)` returns `110`, but `cleverness(man1)` returns an `ERROR: MethodError: `cleverness` has no method matching cleverness(::Manager)` error message; the function has no method for a manager.

The following function makes us think that managers are always cleverer, which is, of course, not true:

```
function cleverer(m::Manager, e::Employee)
    println("The manager $(m.name) is cleverer!")
end

cleverer(man1, devel1) #> The manager Julia is cleverer!
cleverer(man1, devel2) #> The manager Julia is cleverer!
```

Suppose we introduce a function `cleverer` with the following argument types:

```
function cleverer(d::Developer, e::Employee)
    println("The developer $(d.name) is cleverer I think!")
end
```

The `cleverer(devel1, devel2)` call will now print `"The developer Bob is cleverer I think!"`. (Clearly, the function isn't yet coded right.) It matches a method because `devel2` is also an employee.

However, `cleverer(devel1, man1)` will give an `ERROR: MethodError: `cleverer` has no method matching cleverer(::Developer,::Manager)`) error message, as a manager is not an employee, and a method with this signature was not defined.

We should now define another method for `cleverer` as follows:

```
function cleverer(e::Employee, d::Developer)
    if e.iq <= d.iq
        println("The developer $(d.name) is cleverer!")
    else
        println("The employee $(e.name) is cleverer!")
    end
end
```

If we now call `cleverer(devel1, devel2)` an ambiguity arises; Julia detects a problem in the definitions and gives us the following error:

```
#> ERROR: MethodError: cleverer(::Developer, ::Developer) is ambiguous.
Candidates:
 cleverer(e::Employee, d::Developer) in Main at REPL[32]:2
 cleverer(d::Developer, e::Employee) in Main at REPL[29]:2
Possible fix, define
 cleverer(::Developer, ::Developer)
```

The ambiguity is that, if `cleverer` is called with `e` being a `Developer`, which of the two defined methods should be chosen? To remove this ambiguity, we will now define the more specific (and correct) method, as follows:

```
function cleverer(d1::Developer, d2::Developer)
    if d1.iq <= d2.iq
        println("The developer $(d2.name) is cleverer!")
    else
        println("The developer $(d1.name) is cleverer!")
    end
end
```

Now, `cleverer(devel1, devel2)` prints `"The developer William is cleverer!"` as well as `cleverer(devel2, devel1)`.This illustrates multiple dispatching. When defined, the more specific method definition (here, the second method `cleverer`) is chosen; more specific means the method with more specialized type annotations for its arguments. More specialized doesn't only mean subtypes, it can also mean using type aliases.

TIP *Always avoid method ambiguities by specifying an appropriate method for the intersection case.*

Types and collections – inner constructors

Here is another type with only default constructors:

```
# see the code in Chapter 6\inner_constructors.jl
mutable struct Person
    firstname::String
    lastname::String
    sex::Char
    age::Float64
    children::Array{String, 1}
end

p1 = Person("Alan", "Bates", 'M', 45.5, ["Jeff", "Stephan"])
```

This example demonstrates that an object can contain collections, such as arrays or dictionaries. Custom types can also be stored in a collection, just like built-in types, for example:

```
people = Person[]
```

This returns `0-element Array{Person,1}`:

```
push!(people, p1)
push!(people, Person("Julia", "Smith", 'F', 27, ["Viral"]))
```

The `show(people)` function now returns the following output:

```
Person[Person("Alan", "Bates", 'M', 45.5, ["Jeff", "Stephan"]),
       Person("Julia", "Smith", 'F', 27.0, ["Viral"])]
```

Now we can define a function `fullname` on type `Person`. You will notice that the definition stays outside the type's code:

```
fullname(p::Person) = "$(p.firstname) $(p.lastname)"
```

Or, slightly more performant:

```
fullname(p::Person) = string(p.firstname, " ", p.lastname)
```

Now `print(fullname(p1))` returns `Alan Bates`.

If you need to include error checking or transformations as part of the type construction process, you can use inner constructors (so-called because they are defined inside the type itself), as shown in the following example:

```
mutable struct Family
     name::String
     members::Array{String, 1}
     big::Bool
  Family(name::String) = new(name, String[], false)
  Family(name::String, members) = new(name, members,
    length(members) > 4)
end
```

We can make a `Family` object as follows:

```
fam = Family("Bates-Smith", ["Alan", "Julia", "Jeff", "Stephan",
  "Viral"])
```

Then the output is as follows:

```
Family("Bates-Smith",String["Alan","Julia","Jeff","Stephan","Viral"],true)
```

The keyword `new` can only be used in an inner constructor to create an object of the enclosing type. The first constructor takes one argument and generates a default for the other two values. The second constructor takes two arguments and infers the value of `big`. Inner constructors give you more control over how the values of the type can be created. Here, they are written with the short function notation, but if they are multiline, they will use the normal function syntax.

Note that when you use inner constructors, there are no default constructors any more. Outer constructors, calling a limited set of inner constructors, are often the best practice.

Type unions

In geometry, a two-dimensional point and a vector are not the same, even if they both have an `x` and `y` component. In Julia, we can also define them as different types, as follows:

```
# see the code in Chapter 6\unions.jl
mutable struct Point
    x::Float64
    y::Float64
end

mutable struct Vector2D
    x::Float64
    y::Float64
end
```

Here are the two objects:

- `p = Point(2, 5)` that returns `Point(2.0, 5.0)`
- `v = Vector2D(3, 2)` that returns `Vector2D(3.0, 2.0)`

Suppose we want to define the sum for these types as a point which has coordinates as the sum of the corresponding coordinates:

```
+(p, v)
```

This results in an `ERROR: MethodError: `+` has no method matching +(::Point, ::Vector2D)` error message.

To define a `+` method here, first do an `import Base.+`

Even after defining the following, `+(p, v)` still returns the same error because of multiple dispatch. Julia has no way of knowing that `+(p,v)` should be the same as `+(v,p)`:

```
+(p::Point, q::Point) = Point(p.x + q.x, p.y + q.y)
+(u::Vector2D, v::Vector2D) = Point(u.x + v.x, u.y + v.y)
+(u::Vector2D, p::Point) = Point(u.x + p.x, u.y + p.y)
```

Only when we define the type matching method as `+(p::Point, v::Vector2D) = Point(p.x + v.x, p.y + v.y)`, do we get a result `+(p, v)`, which returns `Point(5.0,7.0)`.

Now you can ask the question: Don't multiple dispatch and many types give rise to code duplication, as is the case here?

The answer is no, because, in such a case, we can define a union type, `VecOrPoint`:

```
VecOrPoint = Union{Vector2D, Point}
```

If `p` is a point, it is also of type `VecOrPoint`, and the same is true for `v` which is `Vector2D`. `isa(p, VecOrPoint)` and `isa(v, VecOrPoint)`; both return `true`.

Now we can define one `+` method that works for any of the preceding four cases:

```
+(u::VecOrPoint, v:: VecOrPoint) = VecOrPoint(u.x + v.x, u.y +
 v.y)
```

So, now we only need one method instead of four.

Parametric types and methods

An array can take elements of different types. Therefore, we can have, for example, arrays of the following types: `Array{Int64,1}`, `Array{Int8,1}`, `Array{Float64,1}`, or `Array{String, 1}`, and so on. That is why an `Array` is a parametric type; its elements can be of any arbitrary type `T`, written as `Array{T, 1}`.

In general, types can take type parameters, so that type declarations actually introduce a whole family of new types. Returning to the `Point` example of the previous section, we can generalize it to the following:

```
# see the code in Chapter 6\parametric.jl
mutable struct Point{T}
  x::T
  y::T
end
```

This is conceptually similar to the generic types in Java or templates in C++.

This abstract type creates a whole family of new possible concrete types (but they are only compiled as needed at runtime), such as `Point{Int64}`, `Point{Float64}`, and `Point{String}`.

These are all subtypes of `Point`: `Point{String} <: Point` returns `true`. However, this is not the case when comparing different `Point` types, whose parameter types are subtypes of one another: `Point{Float64} <: Point{Real}` returns `false`.

To construct objects, you can indicate the type `T` in the constructor, as in `p = Point{Int64}(2, 5)`, but this can be shortened to `p = Point(2, 5)`. Or let's consider another example: `p = Point("London", "Great-Britain")`.

If you want to restrict the parameter type `T` to only the subtypes of `Real`, this can be written as follows:

```
mutable struct Point{T <: Real}
  x::T
  y::T
end
```

Now, the statement `p = Point("London", "Great-Britain")` results in an `ERROR: MethodError: `Point{T<:Real}` has no method matching Point{T<:Real}(::String, :String)` error message, because `String` is not a subtype of `Real`.

In much the same way, methods can also optionally have type parameters immediately after their name and before the tuple of arguments. For example, to constrain two arguments to be of the same type `T`, run the following command:

```
add(x::T, y::T) where T = x + y
```

Now, `add(2, 3)` returns `5` and `add(2, 3.0)` returns an `ERROR: MethodError: `add` has no method matching add(::Int64, ::Float64)` error message.

Here, we restrict `T` to be a subtype of `Number` in `add` as follows:

```
add(x::T, y::T) where T <: Number = x + y
```

As another example, here is how to check whether a `vecfloat` function only takes a vector of floating point numbers as the input. Simply define it with a type parameter `T` as follows:

```
function vecfloat(x::Vector{T}) where T <: AbstractFloat
    # code
  end
```

Inner constructors can also take type parameters in their definition.

Standard modules and paths

The code for Julia packages (also called **libraries**) is contained in a module whose name starts with an uppercase letter by convention, like this:

```
# see the code in Chapter 6\modules.jl
module Package1

export Type1, perc

include("file1.jl")
include("file2.jl")

# code
mutable struct Type1
    total
end

perc(a::Type1) = a.total * 0.01

end
```

This serves to separate all its definitions from those in other modules so that no name conflicts occur. Name conflicts are solved by qualifying the function by the module name. For example, the packages `Winston` and `Gadfly` both contain a function plot. If we needed these two versions in the same script, we would write it as follows:

```
import Winston
import Gadfly
Winston.plot(rand(4))
Gadfly.plot(x=[1:10], y=rand(10))
```

All variables defined in the `global` scope are automatically added to the `Main` module. Thus, when you write `x = 2` in the REPL, you are adding the variable `x` to the `Main` module.

Julia starts with `Main` as the current top-level module. The module `Core` is a set of non-Julia sources (in the `src` directory of the GitHub source); for example, C/C++ and Femtolisp, which are used to create `libjulia`, are used by the Julia source to interface to the OS through the API. The standard library is also available. The code for the standard library (the contents of `/base`) is contained in the following modules:

- `Base64`
- `FileWatching`
- `LinearAlgebra`
- `Printf`
- `Serialization`
- `SuiteSparse`
- `CRC32c`
- `Future`
- `Logging`
- `Profile`
- `SharedArrays`
- `Test`
- `Dates`
- `InteractiveUtils`
- `Markdown`
- `REPL`
- `Sockets`
- `UUIDs`
- `DelimitedFiles`
- `LibGit2`
- `Mmap`
- `Random`
- `SparseArrays`
- `Unicode`
- `Distributed`
- `Libdl`
- `Pkg`
- `SHA`
- `Statistics`

The type of a module is `Module: typeof(Base)`, which returns `Module`. If we call `names(Main)`, we get, for example, `5-element Array{Symbol,1}: :ans, :Main, :Core, :Base`, and `:InteractiveUtils`. If you have defined other variables or functions in the REPL, these would also show up.

All the top-level defined variables and functions, together with the default modules, are stored as symbols. The `varinfo()` function lists these objects with their types:

```
name                   size     summary
---------------- ----------- -------
Base                         Module
Core                         Module
InteractiveUtils 157.063 KiB Module
Main                         Module
```

This can also be used for another module. For example, `varinfo(Winston)` lists all the exported names from the module `Winston`.

A module can make some of its internal definitions (such as constants, variables, types, functions, and so on) visible to other modules (as if making them public) by declaring them with `export`. This can be seen in the following example:

```
export Type1, perc
```

For the preceding example, using `Package1` will make the type `Type1` and function `perc` available in other modules that import them through this statement. All the other definitions remain invisible (or private).

As we saw in `Chapter 1`, *Installing the Julia Platform*, a module can also include other source files in their entirety with `include("file1.jl")`. However, this means that the included files are not modules. Using `include("file1.jl")` is, to the compiler, no different from copying `file1.jl` and pasting it directly in the current file or the REPL.

In general, use `import` to import definitions from another module in the current module:

- After `import.LibA`, you can use all definitions from `LibA` inside the current module by qualifying them with `LibA.`, such as `LibA.a`

- The `import LibB.varB` or `import LibD.funcD` statement only imports one name; the function `funcD` must be used as `LibD.funcD`.

- Use `importall LibE` to import all the exported names from `LibE` in the current module.

Here is a more concrete example. Suppose we define a `TemperatureConverter` module as follows:

```
#code in Chapter 6\temperature_converter.jl
module TemperatureConverter

  function as_celsius(temperature, unit)
    if unit == :Celsius
      return temperature
    elseif unit == :Kelvin
      return kelvin_to_celsius(temperature)
    end
  end

  function kelvin_to_celsius(temperature)
    # 'private' function
    return temperature + 273
  end

end
```

We can now use this module in another program as follows:

```
#code in Chapter 6\using_module.jl
include("temperature_converter.jl")

println("$(TemperatureConverter.as_celsius(100, :Celsius))")
#> 100
println("$(TemperatureConverter.as_celsius(100, :Kelvin))")
#> 373
println("$(TemperatureConverter.kelvin_to_celsius(0))") #> 273
```

Imported variables are read-only, and the current module cannot create variables with the same names as the imported ones. A source file can contain many modules, or one module can be defined in several source files. If a module contains a function `__init__()`, this will be executed when the module is first loaded.

The variable `LOAD_PATH` contains a list of directories where Julia looks for (module) files when running the `using`, `import`, or `include` statements. Put this statement in the file `~/.julia/config/startup.jl` to extend `LOAD_PATH` on every Julia startup:

```
push!(LOAD_PATH, "new/path/to/search")
```

Summary

In this chapter, we delved into types and type hierarchies in Julia. We got a much better understanding of types and how functions work on them through multiple dispatch. The next chapter will reveal another power tool in Julia: metaprogramming and macros.

Metaprogramming in Julia

Everything in Julia is an expression that returns a value when executed. Every piece of the program code is internally represented as an ordinary Julia data structure, also called an **expression**. In this chapter, we will see how, by working on expressions, a Julia program can transform and even generate new code. This is a very powerful characteristic, also called **homoiconicity**. It inherits this property from **Lisp**, where code and data are just lists, and where it is commonly referred to with the phrase: *"Code is data and data is code"*.

In homoiconic languages, code can be expressed in terms of the language syntax. This is the case for the Lisp-like family of languages: Lisp, Scheme and, more recently, Clojure, which use s-expressions. Julia is homoiconic, as are others such as Prolog, IO, Rebol, and Red. As such, these are able to generate code during runtime, which can be subsequently executed.

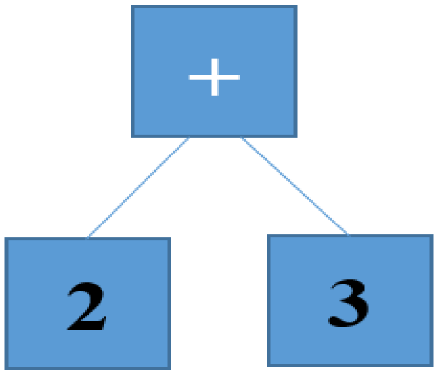
We will explore this metaprogramming power by covering the following topics:

- Expressions and symbols
- Evaluation and interpolation
- Defining macros
- Built-in macros
- Reflection capabilities

Expressions and symbols

An **abstract syntax tree** (**AST**) is a tree representation of the abstract syntactic structure of source code written in a programming language. When Julia code is parsed by its LLVM JIT compiler, it is internally represented as an abstract syntax tree. The nodes of this tree are simple data structures of the type expression `Expr`. For more information on abstract syntax trees, refer to `http://en.wikipedia.org/wiki/Abstract_syntax_tree`.

An expression is simply an object that represents Julia code. For example, `2 + 3` is a piece of code, which is an expression of type `Int64` (follow along with the code in `Chapter 7\expressions.jl`). Its syntax tree can be visualized as follows:



To make Julia see this as an expression and block its evaluation, we have to quote it, that is, precede it by a colon (`:`) as in `:(2 + 3)`. When you evaluate `:(2 + 3)` in the REPL, it just returns `:(2 + 3)`, which is of type `Expr`: `typeof(:(2 + 3))` returns `Expr`. In fact, the `:` operator (also called the **quote** operator) sets out to treat its argument as data, not as code.

If this code is more than one line, enclose the lines between the `quote` and `end` keywords to turn the code into an expression. For example, this expression just returns itself:

```
quote
    a = 42
    b = a^2
    a - b
end
```

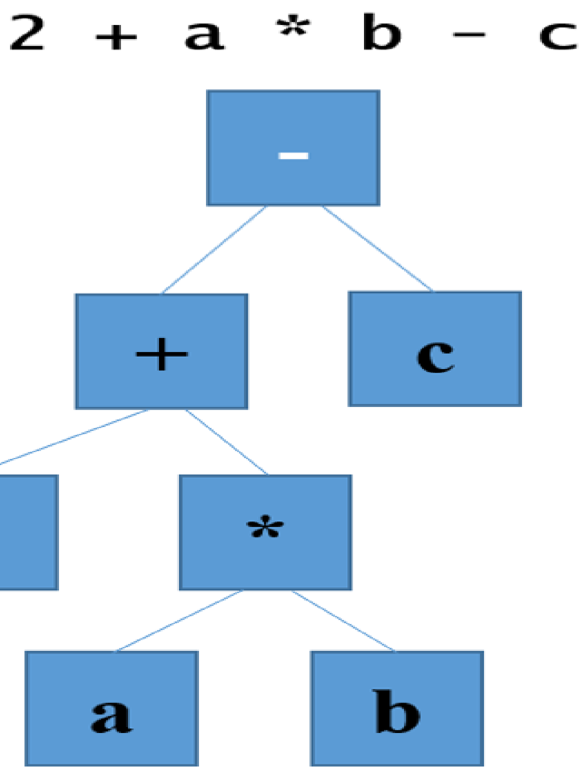
In fact, this is the same as `:(a = 42; b = a^2; a - b)`. `quote...end` is just another way to convert blocks of code into expressions.

We can give an expression such as this a name, such as `e1 = :(2 + 3)`. We can ask for the following information:

- `e1.head` returns `:call`, indicating the kind of expression, which here is a function call
- `e1.args` returns `3-element Array{Any,1}: :+ 2 3`

Indeed the expression `2 + 3` is, in fact, a call of the `+` function with the argument `2`, and `3`: `2 + 3 == + (2, 3)` returns `true`. The `args` argument consists of a symbol `:+`, and two literal values, `2` and `3`. Expressions are made up of symbols and literals. More complicated expressions will consist of literal values, symbols, and sub- or nested expressions, which can, in turn, be reduced to symbols and literals.

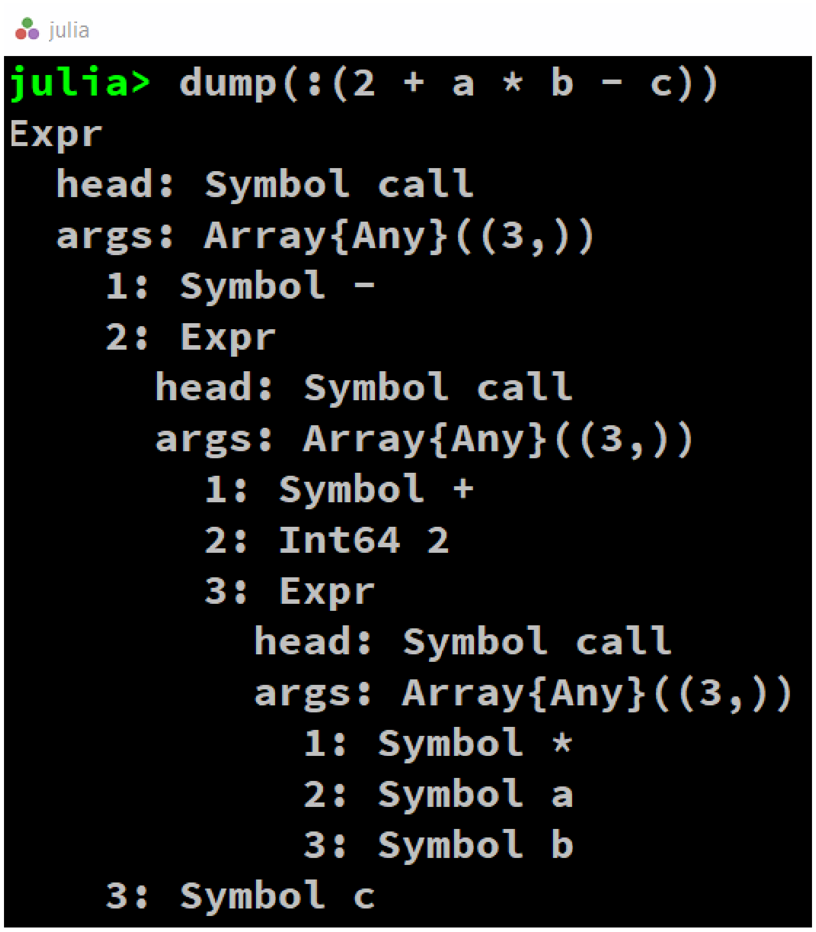
For example, consider the expression `e2 = :(2 + a * b - c)`, which can be visualized by the following syntax tree:



`e2` consists of `e2.args`, which is a `3-element Array{Any,1}` that contains `:-` and `:c`, which are symbols, and `:(2 + a * b)`, which is also an expression. This last expression, in turn, is itself an expression with `args:+`, `2`, and `:(a * b)`; `:(a * b)` is an expression with arguments and symbols: `:*`, `:a`, and `:b`. We can see that this works recursively; we can simplify every subexpression in the same way until we end up with elementary symbols and literals.

In the context of an expression, *symbols are used to indicate access to variables*; they represent the variable in the tree structure for the code. In fact, the **prevent evaluation** character of the `quote` operator (`:`) is already at work with symbols: after `x = 5`, `x` returns `5`, but `:x` returns `:x`.

The `dump` function presents the abstract syntax tree for its argument in a nice way. For example, `dump(:(2 + a * b - c))` returns the output, as shown in the following screenshot:



```
julia> dump(:(2 + a * b - c))
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol -
    2: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol +
        2: Int64 2
        3: Expr
          head: Symbol call
          args: Array{Any}((3,))
            1: Symbol *
            2: Symbol a
            3: Symbol b
    3: Symbol c
```

Evaluation and interpolation

With the definition of type `Expr` from the preceding section, we can also build expressions directly from the constructor for `Expr`. For example: `e1 = Expr(:call, *, 3, 4)` returns `:((*)(3, 4))` (follow along with the code in `Chapter 7\eval.jl`).

The result of an expression can be computed with the `eval` function, `eval(e1)`, which returns `12` in this case. At the time an expression is constructed, not all the symbols have to be defined, but they have to be defined at the time of evaluation, otherwise an error occurs.

For example, `e2 = Expr(:call, *, 3, :a)` returns `:((*)(3, a))`, and `eval(e2)` then gives `ERROR: UndefVarError: a not defined`. Only after we say, for example, `a = 4` does `eval(e2)` return `12`.

Expressions can also change the state of the execution environment, for example, the expression `e3 = :(b = 1)` assigns a value to `b` when evaluated, and even defines `b`, if it doesn't exist already.

To make writing expressions a bit simpler, we can use the `$` operator to do **interpolation** in expressions, as with `$` in strings, and this will evaluate immediately when the expression is made. The expressions `a = 4` and `b = 1`, `e4 = :(a + b)` return `:(a + b)`, and `e5 = :($a + b)` returns `:(4 + b)`; both expressions evaluate to `5`. So, there are two kinds of evaluation here:

- Expression interpolation (with `$`) evaluates when the expression is constructed (at parse time)
- Quotation (with `:` or `quote`) evaluates only when the expression is passed to `eval` at runtime

We now have the capability to build code programmatically. Inside a Julia program, we can construct arbitrary code while it is running, and then evaluate this with `eval`. So, Julia can generate the code from inside itself during the normal program execution.

This happens all the time in Julia and it is used, for example, to do things such as generating bindings for external libraries, to reducing the repetitive boilerplate code needed to bind big libraries, or generating lots of similar routines in other situations. Also, in the field of robotics, the ability to generate another program and then run it is very useful. For example: a chirurgical robot learns how to move by perceiving a human surgeon demonstrating a procedure. Then, the robot generates the program code from that perception, so that it is able to perform the procedure by itself.

One of the most powerful Julia tools emerging from what we discussed before is **macros**, which exist in all the languages of the Lisp family.

Julia version 1.0 also introduces the concept of **generated functions**: such functions are prefixed by the `@generated` macro and, instead of normal values, they return expressions. We won't discuss this advanced concept here in this book.

Defining macros

In previous chapters, we have already used macros, such as `@printf`, in Chapter 2, *Variables, Types, and Operations*, and `@time` in Chapter 3, *Functions*. Macros are like functions, but instead of values they take expressions (which can also be symbols or literals) as input arguments. When a macro is evaluated, the input expression is expanded, that is, the macro returns a modified expression. This expansion occurs at parse time when the syntax tree is being built, not when the code is actually executed.

The following descriptions highlight the difference between macros and functions when they are called or invoked:

- **Function**: It takes the input values and returns the computed values at runtime
- **Macro**: It takes the input expressions and returns the modified expressions at parse time

In other words, a macro is a custom program transformation. Macros are defined with the keyword as follows:

```
macro mname
# code returning expression
end
```

It is invoked as `@mname exp1 exp2` or `@mname(exp1, exp2)` (the `@` sign distinguishes it from a normal function call). The macro block defines a new scope. Macros allow us to control when the code is executed.

Here are some examples:

- A first simple example is `macint` macro, which does the interpolation of its argument expression `ex`:

```
        # see the code in Chapter 7\macros.jl)
        macro macint(ex)
            quote
                println("start")
                $ex
                println("after")
            end
        end
```

`@macint println("Where am I?")` will result in:

```
start
Where am I?
after
```

- The second example is an `assert` macro that takes an expression `ex` and tests whether it is true or not; in the last case, an error is thrown:

```
macro assert(ex)
:($ex ? nothing : error("Assertion failed: ",
      $(string(ex))))
end
```

For example: `@assert 1 == 1.0` returns `nothing`. `@assert 1 == 42` returns `ERROR: Assertion failed: 1 == 42.`

The macro replaces the expression with a ternary operator expression, which is evaluated at runtime. To examine the resulting expression, use the `macroexpand` function as follows:

```
macroexpand(Main, :(@assert 1 == 42))
```

This returns the following expression:

```
:(if 1 == 42
        nothing
  else
        (Base.throw)((Base.AssertionError)("1 == 42"))
  end)
```

This `assert` function is just a macro example, using the built-in assert function in the production code. (Refer to the *Testing* subsection of the *Built-in macros* section.)

- The third example mimics an `unless` construct, where `branch` is executed if the condition `test_cond` is not true:

```
macro unless(test_cond, branch)
    quote
        if !$test_cond
```

```
            $branch
        end
    end
end
```

Suppose `arr = [3.14, 42, 'b']`, then `@unless 42 in arr println("arr does not contain 42")` returns `nothing`, but `@unless 41 in arr println("arr does not contain 41")` prints out the following command:

```
arr does not contain 41
```

Here, `macroexpand(Main, :(@unless 41 in arr println("arr does not contain 41")))` returns the following output:

```
quote
    #= REPL[49]:3 =#
    if !(41 in Main.arr)
        #= REPL[49]:4 =#
        (Main.println)("arr does not contain 41")
    end
end
```

Unlike functions, macros inject the code directly into the namespace in which they are called, possibly this is also in a different module than the one in which they were defined. It is therefore important to ensure that this generated code does not clash with the code in the module in which the macro is called. When a macro behaves appropriately like this, it is called a **hygienic macro**. The following rules are used when writing hygienic macros:

- Declare the variables used in the macro as `local`, so as not to conflict with the outer variables
- Use the escape function `esc` to make sure that an interpolated expression is not expanded, but instead is used literally
- Don't call `eval` inside a macro (because it is likely that the variables you are evaluating don't even exist at that point)

These principles are applied in the following `timeit` macro, which times the execution of an expression `ex` (like the built-in macro `@time`):

```
macro timeit(ex)
    quote
        local t0 = time()
        local val = $(esc(ex))
        local t1 = time()
        print("elapsed time in seconds: ")
        @printf "%.3f" t1 - t0
        val
    end
end
```

The expression is executed through `$`, and `t0` and `t1` are respectively the start and end times.

`@timeit factorial(10)` returns `elapsed time in seconds: 0.0003628800`.

`@timeit a^3` returns `elapsed time in seconds: 0.0013796416`.

Hygiene with macros is all about differentiating between the macro context and the calling context.

Macros are valuable tools which save you a lot of tedious work, and, with the quoting and interpolation mechanism, they are fairly easy to create. You will see them being used everywhere in Julia for lots of different tasks. Ultimately, they allow you to create **domain-specific languages** (**DSLs**). To get a better idea of this concept, we suggest you experiment with the other examples in the accompanying code file.

Built-in macros

Needless to say, the Julia team has put macros to good use. To get help information about a macro, enter a `?` in the REPL, and type `@macroname` after the `help>` prompt. Apart from the built-in macros we encountered in the examples in the previous chapters, here are some other very useful ones (refer to the code in `Chapter 7\built_in_macros.jl`).

Testing

The `@assert` macro actually exists in the standard library. The standard version also allows you to give your own error message, which is printed after `ERROR: assertion failed`.

The `Test` library contains some useful macros to compare the numbers:

```
using Test
@test 1 == 3
```

This returns the following:

```
Test Failed at REPL[5]:1
  Expression: 1 == 3
    Evaluated: 1 == 3
ERROR: There was an error during testing.
```

`@test` with the `≈` operator tests whether the two numbers are approximately equal. `@test 1 ≈ 1.1` returns `Test Failed` because they are not equal within the machine tolerance. However, you can give the interval as the last argument within which they should be equal: `@test 1 ≈ 1.1 atol=0.2`, which returns `Test Passed`, so `1` and `1.1` are within `0.2` from each other.

Debugging

If you want to look up in the source code where and how a particular method is defined, use `@which`. For example: if `arr = [1, 2]` then `@which sort(arr)` returns `sort(v::AbstractArray{T,1}) where T) in Base.Sort at sort.jl:683 at sort.jl:334`.

`@show` shows the expression and its result, which is handy for checking the embedded results: `456 * 789 + (@show 2 + 3)` gives `2 + 3 => 5     359789`.

Benchmarking

For benchmarking purposes, we already know `@time` and `@elapsed`; `@timed` gives you the `@time` results as a tuple:

`@time [x^2 for x in 1:1000]` prints `elapsed time: 3.911e-6 seconds (8064 bytes allocated)` and returns `1000-element Array{Int64,1}: ....`

`@timed [x^2 for x in 1:1000]` returns the following:

```
([1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ... 982081, 984064, 986049, 988036, 990025,
992016, 994009, 996004, 998001, 1000000], 3.911e-6, 8064, 0.0)
```

`@elapsed [x^2 for x in 1:1000]` returns `3.422e-6`.

If you are specifically interested in the allocated memory, use `@allocated [x^2 for x in 1:1000]`, which returns `8064`.

If you are looking for a package, consult `BenchmarkingTools`. This has some macros and also a good method for benchmarking.

Starting a task

Tasks (refer to the *Tasks* section in `Chapter 4`, *Control Flow*) are independent units of code execution. Often, we want to start executing them, and then continue executing the main code without waiting for the task result. In other words, we want to start the task *asynchronously*. This can be done with the `@async` macro:

```
a = @async 1 + 2 # Task (done) @0x000000002d70faf0
```

Reflection capabilities

We saw in this chapter that code in Julia is represented by expressions that are data structures of the `Expr` type. The structure of a program and its types can therefore be explored programmatically just like any other data. This means that a running program can dynamically discover its own properties, which is called **reflection**. We have already encountered some of these macros or functions before:

- Use the `@isdefined` macro to check whether a variable is already declared, for example if `a` is not declared, you get:

```
@isdefined a #> false
```

- Use the `typeof` and `InteractiveUtils.subtypes` to query the type `hierarchy` (refer to Chapter 6, *More on Types, Methods, and Modules*)
- Use the `methods(f)` to see all the methods of a function `f` (refer to Chapter 3, *Functions*)
- `names` and `types`: given a type `Person`:

```
mutable struct Person
    name:: String
    height::Float64
end
```

  Then, `fieldnames(Person)` returns the field names as a tuple of symbols: `(:name, :height)`

  `Person.types` returns a tuple with the field types `(String, Float64)`.

- To inspect how a function is represented internally, you can use `code_lowered`:

```
code_lowered(+, (Int, Int))
```

  This returns the following output:

```
1-element Array{Core.CodeInfo,1}:
  CodeInfo(
 53 1 ─ %1 = Base.add_int(%%x, %%y)::Any
│
    └──      return %1
│
 )
```

Or you can use `code_typed` to see the type-inferred form:

```
code_typed(+, (Int, Int))
```

This returns the following:

```
1-element Array{Any,1}:
 1-element Array{Any,1}:
  CodeInfo(
 53 1 ─ %1 = Base.add_int(%%x, %%y)::Int64
│
    └──      return %1
│
 ) => Int64
```

Using `code_typed` can show you whether your code is type-optimized for performance: if the `Any` type is used instead of an appropriate specific type that you would expect, then the type annotation in your code can certainly be improved, leading most likely to speeding up the program's execution.

- To inspect the code generated by the LLVM engine, use `code_llvm`, and, to see the assembly code generated, use `code_native` (refer to the *How Julia works* section in `Chapter 1`, *Installing the Julia Platform*).

While reflection is not necessary for many of the programs that you will write, it is very useful for IDEs to be able to inspect the internals of an object, as well as for tools generating automatic documentation, and for profiling tools. In other words, reflection is indispensable for tools that need to inspect the internals of code objects programmatically.

You should also look at the `MacroTools` package (from Mike Innes) which has some good examples of macros.

Summary

In this chapter, we explored the expression format in which Julia is parsed. Because this format is a data structure, we can manipulate this in the code, and this is precisely what macros can do. We explored a number of them, and also some built-in ones that can be useful.

In the next chapter, we will extend our vision to the network environment in which Julia runs, and we will explore its powerful capabilities for parallel execution.

I/O, Networking, and Parallel Computing

In this chapter, we will explore how Julia interacts with the outside world, reading from standard input and writing to standard output, files, networks, and databases. Julia provides asynchronous networking I/O using the `libuv` library. We will see how to handle data in Julia. We will also explore Julia's parallel processing model.

In this chapter, the following topics are covered:

- Basic input and output
- Working with files (including CSV files)
- Using DataFrames
- Working with TCP sockets and servers
- Interacting with databases
- Parallel operations and computing

Basic input and output

Julia's vision on **input/output** (**I/O**) is **stream-oriented**, that is, reading or writing streams of bytes. We will introduce different types of stream, file streams, in this chapter. **Standard input** (**stdin**) and **standard output** (**stdout**) are constants of the `TTY` type (an abbreviation for the old term, `Teletype`) that can be read from and written to in Julia code (refer to the code in `Chapter 8\io.jl`):

- `read(stdin, Char)`: This command waits for a character to be entered, and then returns that character; for example, when you type in `J`, this returns `'J'`
- `write(stdout, "Julia")`: This command types out `Julia5` (the added `5` is the number of bytes in the output stream; it is not added if the command ends in a semicolon `;`)

`stdin` and `stdout` are simply streams and can be replaced by any stream object in read/write commands. `readbytes` is used to read a number of bytes from a stream into a vector:

- `read(stdin,3)`: This command waits for an input, for example, `abe` reads three bytes from it, and then returns `3-element Array{Uint8,1}: 0x61 0x62 0x65`.
- `readline(stdin)`: This command reads all the input until a newline character, `\n`, is entered. For example, type `Julia` and press *Enter*; this returns `"Julia\r\n"` on Windows and `"Julia\n"` on Linux.

If you need to read all the lines from an input stream, use the `eachline` method in a `for` loop, for example:

```
stream = stdin
for line in eachline(stream)
    println("Found $line")
    # process the line
end
```

Include the following REPL dialog as an example:

```
First line of input
Found First line of input
2nd line of input
Found 2nd line of input
3rd line...
Found 3rd line...
```

To test whether you have reached the end of an input stream, use `eof(stream)` in combination with a `while` loop, as follows:

```
while !eof(stream)
     x = read(stream, Char)
     println("Found: $x")
# process the character
end
```

We can experiment with the preceding code by replacing `stream` with `stdin` in these examples.

Working with files

To work with files, we need the `IOStream` type. `IOStream` is a type with the `IO` supertype and has the following characteristics:

- The fields are given by `fieldnames(IOStream)`:

```
        (:handle, :ios, :name, :mark)
```

- The types are given by `IOStream.types`:

```
        (Ptr{Nothing}, Array{UInt8,1}, AbstractString, Int64)
```

The file handle is a pointer of the `Ptr` type, which is a reference to the file object.

Opening and reading a line-oriented file with the `example.dat` name is very easy:

```
// code in Chapter 8\io.jl
fname = "example.dat"
f1 = open(fname)
```

`fname` is a string that contains the path to the file, using the escaping of special characters with `\` when necessary. For example, in Windows, when the file is in the `test` folder on the `D:` drive, this would become `d:\\test\\example.dat`. The `f1` variable is now an `IOStream(<file example.dat>)` object.

To read all lines one after another in an array, use `data = readlines(f1)`, which returns `3-element Array{String,1}`:

```
"this is line 1."
"42; 3.14"
"this is line 3."
```

For processing line by line, now only a simple loop is needed:

```
for line in data
  println(line) # or process line
```

```
end
close(f1)
```

Always close the `IOStream` object to clean and save resources. If you want to read the file into one string, use `readall` (for example, see the `word_frequency` program in `Chapter 5`, *Collection Types*). Use this only for relatively small files because of the memory consumption; this can also be a potential problem when using `readlines`.

There is a convenient shorthand with the `do` syntax for opening a file, applying a function process, and closing it automatically. This goes as follows (`file` is the `IOStream` object in this code):

```
open(fname) do file
    process(file)
end
```

As you can recall, in the *Map, filter, and list comprehensions* section in `Chapter 3`, *Functions*, `do` creates an anonymous function, and passes it to `open`. Thus, the previous code example would have been equivalent to `open(process, fname)`. Use the same syntax for processing a `fname` file line by line without the memory overhead of the previous methods, for example:

```
open(fname) do file
    for line in eachline(file)
        print(line) # or process line
    end
end
```

Writing a file requires first opening it with a `"w"` flag, writing strings to it with `write`, `print`, or `println`, and then closing the file handle that flushes the `IOStream` object to the disk:

```
fname =   "example2.dat"
f2 = open(fname, "w")
write(f2, "I write myself to a file\n")
# returns 24 (bytes written)
println(f2, "even with println!")
close(f2)
```

Opening a file with the `"w"` option will clear the file if it exists. To append to an existing file, use `"a"`.

To process all the files in the current folder (or a given folder as an argument to `readdir()`), use this `for` loop:

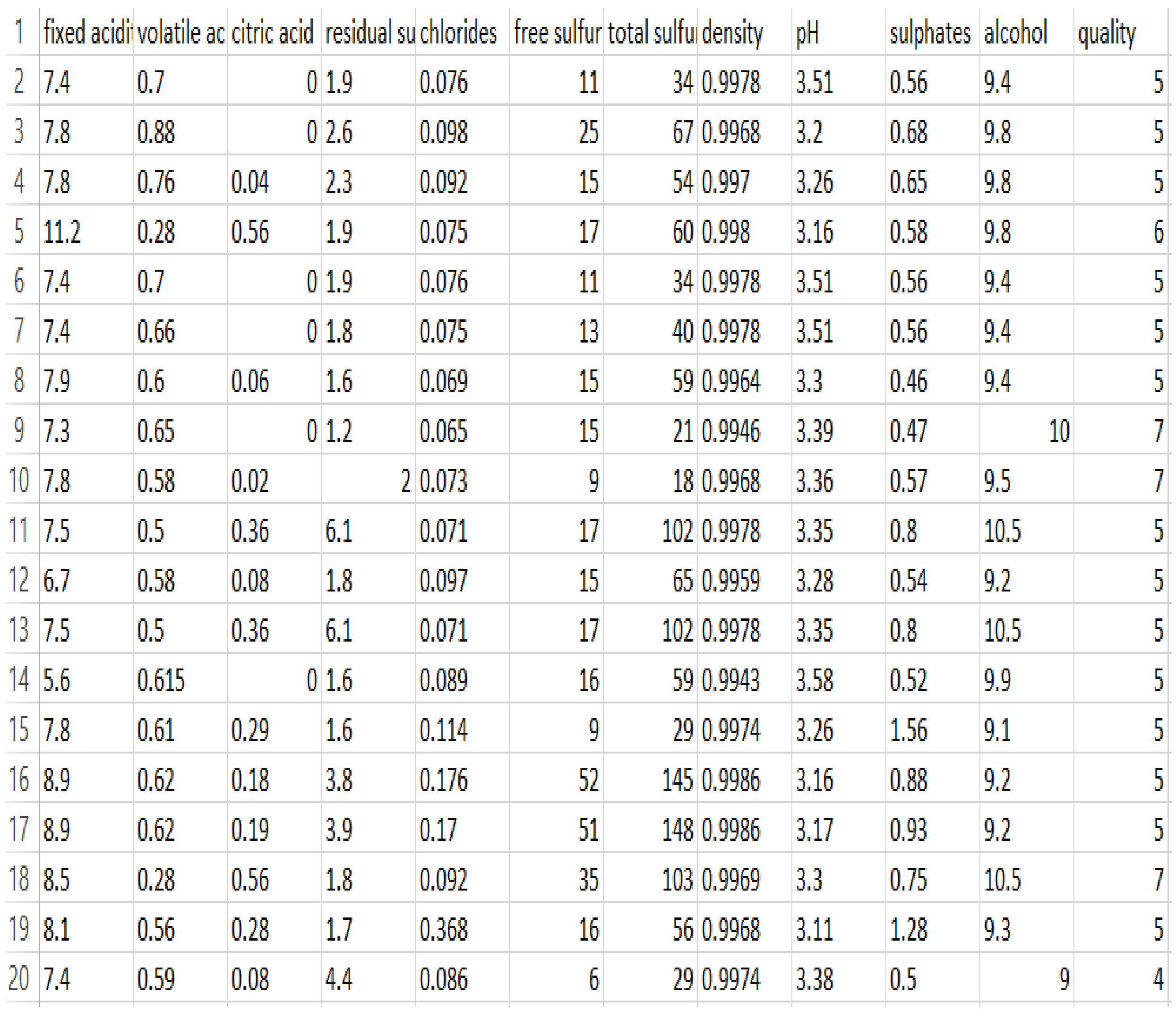
```
for file in readdir()
  # process file
end
```

For example, try out this code, which goes up to the parent directory, prints them out, and then comes back:

```
mydir = pwd()
cd("..")

for fn in readdir()
    println(fn)
end
cd(mydir)
```

Reading and writing CSV files

A **CSV** file is a **comma-separated file**. The data fields in each line are separated by commas, `,`, or another delimiter, such as semicolons, `;`. These files are the de-facto standard for exchanging small and medium amounts of tabular data. Such files are structured so that one line contains data about one data object, so we need a way to read and process the file line by line. As an example, we will use the `Chapter 8\winequality.csv` datafile, which contains 1,599 sample measurements, 12 data columns, such as `pH` and `alcohol`, per sample, separated by a semicolon. In the following screenshot, you can see the top 20 rows:



1	fixed acidi	volatile ac	citric acid	residual su	chlorides	free sulfur	total sulfu	density	pH	sulphates	alcohol	quality
2	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
3	7.8	0.88	0	2.6	0.098	25	67	0.9968	3.2	0.68	9.8	5
4	7.8	0.76	0.04	2.3	0.092	15	54	0.997	3.26	0.65	9.8	5
5	11.2	0.28	0.56	1.9	0.075	17	60	0.998	3.16	0.58	9.8	6
6	7.4	0.7	0	1.9	0.076	11	34	0.9978	3.51	0.56	9.4	5
7	7.4	0.66	0	1.8	0.075	13	40	0.9978	3.51	0.56	9.4	5
8	7.9	0.6	0.06	1.6	0.069	15	59	0.9964	3.3	0.46	9.4	5
9	7.3	0.65	0	1.2	0.065	15	21	0.9946	3.39	0.47	10	7
10	7.8	0.58	0.02	2	0.073	9	18	0.9968	3.36	0.57	9.5	7
11	7.5	0.5	0.36	6.1	0.071	17	102	0.9978	3.35	0.8	10.5	5
12	6.7	0.58	0.08	1.8	0.097	15	65	0.9959	3.28	0.54	9.2	5
13	7.5	0.5	0.36	6.1	0.071	17	102	0.9978	3.35	0.8	10.5	5
14	5.6	0.615	0	1.6	0.089	16	59	0.9943	3.58	0.52	9.9	5
15	7.8	0.61	0.29	1.6	0.114	9	29	0.9974	3.26	1.56	9.1	5
16	8.9	0.62	0.18	3.8	0.176	52	145	0.9986	3.16	0.88	9.2	5
17	8.9	0.62	0.19	3.9	0.17	51	148	0.9986	3.17	0.93	9.2	5
18	8.5	0.28	0.56	1.8	0.092	35	103	0.9969	3.3	0.75	10.5	7
19	8.1	0.56	0.28	1.7	0.368	16	56	0.9968	3.11	1.28	9.3	5
20	7.4	0.59	0.08	4.4	0.086	6	29	0.9974	3.38	0.5	9	4

In general, the `readdlm` function from the `DelimitedFiles` package is used to read in the data from the CSV files:

```
# code in Chapter 8\csv_files.jl:
fname = "winequality.csv"
using DelimitedFiles
data = DelimitedFiles.readdlm(fname, ';')
```

The second argument is the delimiter character (here, it is `;`). The resulting data is a `1600x12 Array{Any,2}` array of the `Any` type because no common type could be found:

```
 "fixed acidity"    "volatile acidity"      "alcohol"    "quality"
   7.4                0.7                     9.4          5.0
   7.8                0.88                    9.8          5.0
   7.8                0.76                    9.8          5.0
...
```

The problem with what we have done so far is that the header (the column titles) was read as part of the data. Fortunately, we can pass the `header=true` argument to let Julia put the first line in a separate array. It then naturally gets the correct datatype, `Float64`, for the data array. We can also specify the type explicitly, such as this:

```
data3 = DelimitedFiles.readdlm(fname, ';', Float64, '\n', header=true)
```

The third argument here is the type of data, which is a numeric type, `String` or `Any`. The next argument is the line-separator character, and the fifth indicates whether or not there is a header line with the field (column) names. If so, then `data3` is a tuple with the data as the first element and the header as the second, in our case, `([7.4 0.7 ... 9.4 5.0; 7.8 0.88 ... 9.8 5.0; ... ; 5.9 0.645 ... 10.2 5.0; 6.0 0.31 ... 11.0 6.0], AbstractString["fixed acidity" "volatile acidity" ... "alcohol" "quality"])` (there are other optional arguments to define `readdlm`; use `? DelimitedFiles.readdlm`). In this case, the actual data is given by `data3[1]` and the header by `data3[2]`.

Let's continue working with variable data. The data forms a matrix, and we can get the rows and columns of data using the normal array-matrix syntax (refer to the *Matrices* section in Chapter 5, *Collection Types*). For example, the third row is given by `row3 = data[3, :]` with `data: 7.8 0.88 0.0 2.6 0.098 25.0 67.0 0.9968 3.2 0.68 9.8 5.0`, representing the measurements for all the characteristics of a certain wine.

The measurements of a certain characteristic for all wines are given by a `data` column; for example, `col3 = data[ :, 3]` represents the measurements of citric acid and returns a `1600-element Array{Any,1}: "citric acid" 0.0 0.0 0.04 0.56 0.0 0.0 ... 0.08 0.08 0.1 0.13 0.12 0.47` column vector.

If we need columns two to four (`volatile acidity` to `residual sugar`) for all wines, extract the data with `x = data[:, 2:4]`. If we need these measurements only for the wines on rows 70-75, get these with `y = data[70:75, 2:4]`, returning a `6 x 3 Array{Any,2}` output, as follows:

```
0.32   0.57  2.0
0.705  0.05  1.9
...
0.675  0.26  2.1
```

To get a matrix with the data from columns 3, 6, and 11, execute the following command:

```
z = [data[:,3] data[:,6] data[:,11]]
```

This includes the headers; if you don't want these, use the following:

```
z = [data[2:end,3] data[2:end,6] data[2:end,11]]
```

It would be useful to create a `Wine` type in the code.

For example, if the data is to be passed around functions, it will improve the code quality to encapsulate all the data in a single data type, like this:

```
struct Wine
    fixed_acidity::Array{Float64}
    volatile_acidity::Array{Float64}
    citric_acid::Array{Float64}
    # other fields
    quality::Array{Float64}
end
```

Then, we can create objects of this type to work with them, like in any other object-oriented language, for example, `wine1 = Wine(data[1, :]...)`, where the elements of the row are splatted with the `...` operator into the `Wine` constructor.

To write to a CSV file, the simplest way is to use the `writecsv` function for a comma separator, or the `writedlm` function if you want to specify another separator. For example, to write an array data to a `partial.dat` file, you need to execute the following command:

```
writedlm("partial.dat", data, ';')
```

If more control is necessary, you can easily combine the more basic functions from the previous section. For example, the following code snippet writes 10 tuples of three numbers each to a file:

```
// code in Chapter 8\tuple_csv.jl
fname = "savetuple.csv"
csvfile = open(fname,"w")
# writing headers:
write(csvfile, "ColName A, ColName B, ColName C\n")
for i = 1:10
  tup(i) = tuple(rand(Float64,3)...)
```

```
    write(csvfile, join(tup(i),","), "\n")
end
close(csvfile)
```

Using DataFrames

If you measure `n` variables (each of a different type) of a single object, then you get a table with `n` columns for each object row. If there are `m` observations, then we have `m` rows of data. For example, given the student grades as data, you might want to know `compute the average grade for each socioeconomic group`, where `grade` and `socioeconomic group` are both columns in the table, and there is one row per student.

`DataFrame` is the most natural representation to work with such a (*m x n*) table of data. They are similar to Pandas `DataFrames` in Python or `data.frame` in R. `DataFrame` is a more specialized tool than a normal array for working with tabular and statistical data, and it is defined in the `DataFrames` package, a popular Julia library for statistical work. Install it in your environment by typing in `add DataFrames` in the REPL. Then, import it into your current workspace with `using DataFrames`. Do the same for the `DataArrays` and `RDatasets` packages (which contain a collection of example datasets mostly used in the R literature).

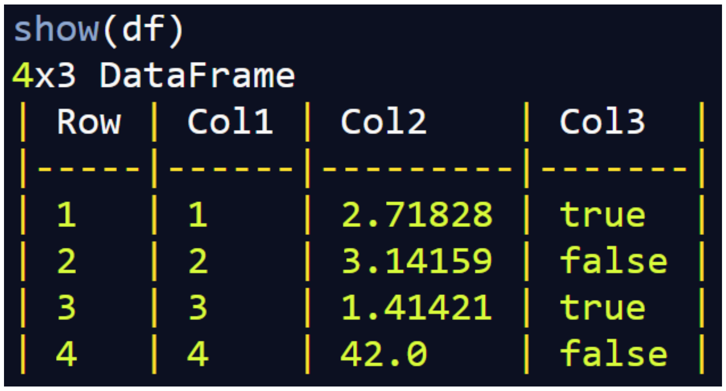
A common case in statistical data is that data values can be missing (the information is not known). The `Missings` package provides us with a unique value, `missing`, which represents a non-existing value, and has the `Missing` type. The result of the computations that contain the `missing` values mostly cannot be determined, for example, `42 + missing` returns `missing`.

`DataFrame` is a kind of in-memory database, versatile in the various ways you can work with data. It consists of columns with names such as `Col1`, `Col2`, and `Col3`. All of these columns are `DataArrays` that have their own type, and the data they contain can be referred to by the column names as well, so we have substantially more forms of indexing. Unlike two-dimensional arrays, columns in `DataFrame` can be of different types. One column might, for instance, contain the names of students and should therefore be a string. Another column could contain their age and should be an integer.

We construct `DataFrame` from the program data as follows:

```
// code in Chapter 8\dataframes.jl
using DataFrames, Missings
# constructing a DataFrame:
df = DataFrame()
df[:Col1] = 1:4
df[:Col2] = [exp(1), pi, sqrt(2), 42]
df[:Col3] = [true, false, true, false]
show(df)
```

Notice that the column headers are used as symbols. This returns the following `4 x 3 DataFrame` object:



```
show(df)
4x3 DataFrame
| Row | Col1 | Col2    | Col3  |
|-----|------|---------|-------|
| 1   | 1    | 2.71828 | true  |
| 2   | 2    | 3.14159 | false |
| 3   | 3    | 1.41421 | true  |
| 4   | 4    | 42.0    | false |
```

`show(df)` produces a nicely formatted output (whereas `show(:Col2)` does not). This is because there is a `show()` routine defined in the package for the entire contents of `DataFrame`.

We could also have used the full constructor, as follows:

```
df = DataFrame(Col1 = 1:4, Col2 = [e, pi, sqrt(2), 42],
     Col3 = [true, false, true, false])
```

You can refer to columns either by an index (the column number) or by a name; both of the following expressions return the same output:

```
show(df[2])
show(df[:Col2])
```

This gives the following output:

```
[2.718281828459045, 3.141592653589793, 1.4142135623730951,42.0]
```

To show the rows or subsets of rows and columns, use the familiar splice (`:`) syntax, for example:

- To get the first row, execute `df[1, :]`. This returns `1x3 DataFrame`:

```
| Row | Col1 | Col2    | Col3 |
|-----|------|---------|------|
| 1   | 1    | 2.71828 | true |
```

- To get the second and third row, execute `df [2:3, :]`.
- To get only the second column from the previous result, execute `df[2:3, :Col2]`. This returns `[3.141592653589793, 1.4142135623730951]`.
- To get the second and third columns from the second and third row, execute `df[2:3, [:Col2, :Col3]]`, which returns the following output:

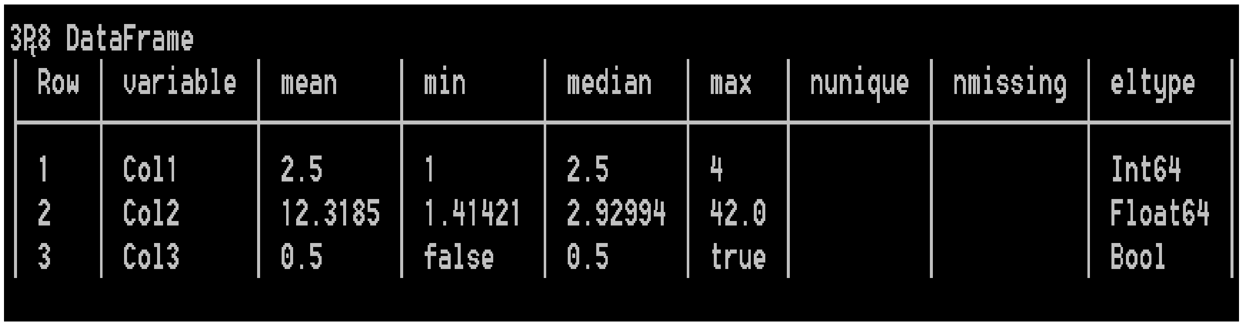
```
2x2 DataFrame
 | Row | Col2    | Col3  |
 |---- |-----   -|-------|
 | 1   | 3.14159 | false |
 | 2   | 1.41421 | true  |
```

The following functions are very useful when working with `DataFrames`:

- The `head(df)` and `tail(df)` functions show you the first six and the last six lines of data, respectively. You can see this in the following example:

```
df0 = DataFrame(i = 1:10, x = rand(10),
                y = rand(["a", "b",  "c"], 10))
head(df0
```

- The `names` function gives the names of the `names(df)` columns. It returns `3-element Array{Symbol,1}: :Col1 :Col2 :Col3`.
- The `eltypes` function gives the data types of the `eltypes(df)` columns. It gives the output as `3-element Array{Type{T<:Top},1}: Int64 Float64 Bool`.
- The `describe` function tries to give some useful summary information about the data in the columns, depending on the type. For example, `describe(df)` gives for column 2 (which is numeric) the minimum, maximum, median, mean, number of unique, and the number of missing:

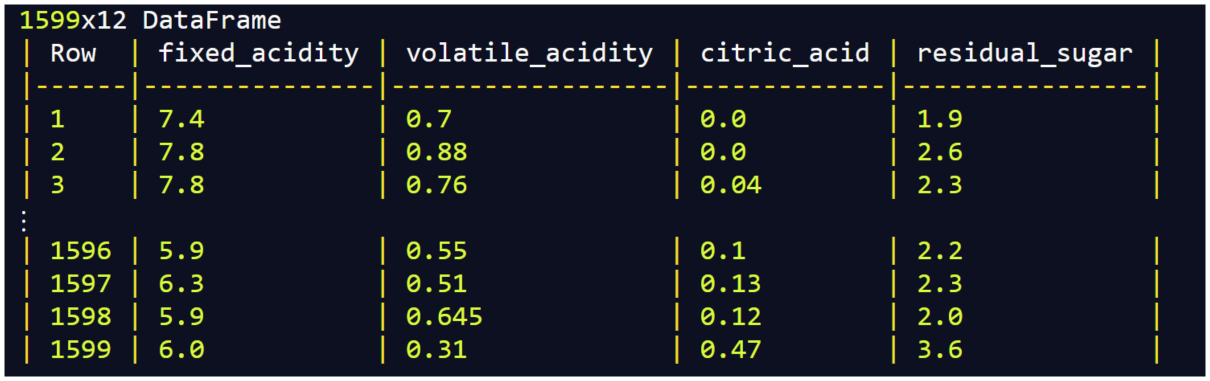


```
3R8 DataFrame
│ Row │ variable │ mean    │ min     │ median  │ max  │ nunique │ nmissing │ eltype  │
├─────┼──────────┼─────────┼─────────┼─────────┼──────┼─────────┼──────────┼─────────┤
│ 1   │ Col1     │ 2.5     │ 1       │ 2.5     │ 4    │         │          │ Int64   │
│ 2   │ Col2     │ 12.3185 │ 1.41421 │ 2.92994 │ 42.0 │         │          │ Float64 │
│ 3   │ Col3     │ 0.5     │ false   │ 0.5     │ true │         │          │ Bool    │
```

To load in data from a local CSV file, use the `read` method from the `CSV` package (the following are the docs for that package: https://juliadata.github.io/CSV.jl/stable/). The returned object is of the `DataFrame` type:

```
// code in Chapter 8\dataframes.jl
using DataFrames, CSV
fname = "winequality.csv"
data = CSV.read(fname, delim = ';')
typeof(data) # DataFrame
size(data) # (1599,12)
```

Here is a fraction of the output:



```
1599x12 DataFrame
| Row  | fixed_acidity | volatile_acidity | citric_acid | residual_sugar |
|------|---------------|------------------|-------------|----------------|
| 1    | 7.4           | 0.7              | 0.0         | 1.9            |
| 2    | 7.8           | 0.88             | 0.0         | 2.6            |
| 3    | 7.8           | 0.76             | 0.04        | 2.3            |
⋮
| 1596 | 5.9           | 0.55             | 0.1         | 2.2            |
| 1597 | 6.3           | 0.51             | 0.13        | 2.3            |
| 1598 | 5.9           | 0.645            | 0.12        | 2.0            |
| 1599 | 6.0           | 0.31             | 0.47        | 3.6            |
```

The `readtable` method also supports reading in the `gzip` CSV files.

Writing `DataFrame` to a file can be done with the `CSV.write` function, which takes the filename and `DataFrame` as arguments, for example, `CSV.write ("dataframe1.csv", df, delim = ';')`. By default, `write` will use the delimiter specified by the filename extension and write the column names as headers.

Both `read` and `write` support numerous options for special cases. Refer to the docs for more information.

To demonstrate some of the power of `DataFrames`, here are some queries you can do:

- Make a vector with only quality information, `data[:quality]`.
- Give wines whose alcohol percentage is equal to `9.5`, for example, `data[ data[:alcohol] .== 9.5, :]`.

  Here, we use the `.==` operator, which does an element-wise comparison. `data[:alcohol] .== 9.5` returns an array of Boolean values (true for `datapoints`, where `:alcohol` is `9.5`, and false otherwise). `data[boolean_array, : ]` selects those rows where `boolean_array` is true.

- Count the number of wines grouped by quality with `by(data, :quality, data -> size(data, 1))`, which returns the following:

```
6x2 DataFrame
| Row | quality | x1  |
|-----|---------|-----|
| 1   | 3       | 10  |
| 2   | 4       | 53  |
| 3   | 5       | 681 |
| 4   | 6       | 638 |
| 5   | 7       | 199 |
| 6   | 8       | 18  |
```

- The `DataFrames` package contains the `by` function, which takes in three arguments:
  - `DataFrame`, here it takes `data`
  - A column to split `DataFrame` on, here it takes `quality`
  - A function or an expression to apply to each subset of `DataFrame`, here `data -> size(data, 1)`, which gives us the number of wines for each quality value

Another easy way to get the distribution among quality is to execute the histogram `hist` function, `hist(data[:quality])`, which gives the counts over the range of quality (`2.0:1.0:8.0,[10,53,681,638,199,18]`). More precisely, this is a tuple with the first element corresponding to the edges of the histogram bins, and the second denoting the number of items in each bin. So there are, for example, 10 wines with quality between 2 and 3.

To extract the counts as a `count` variable of the `Vector` type, we can execute `_, count = hist(data[:quality])`; _; this means that we neglect the first element of the tuple. To obtain the quality classes as a `DataArray` class, we execute the following:

```
class = sort(unique(data[:quality]))
```

We can now construct `df_quality`, a `DataFrame` type with `class` and `count` columns as `df_quality = DataFrame(qual=class, no=count)`. This gives the following output:

```
6x2 DataFrame
| Row | qual | no  |
|-----|------|-----|
| 1   | 3    | 10  |
| 2   | 4    | 53  |
| 3   | 5    | 681 |
| 4   | 6    | 638 |
| 5   | 7    | 199 |
| 6   | 8    | 18  |
```

To deepen your understanding and learn about the other features of Julia's `DataFrames` (such as joining, reshaping, and sorting), refer to the documentation available at http://juliadata.github.io/DataFrames.jl/latest/index.html.

Other file formats

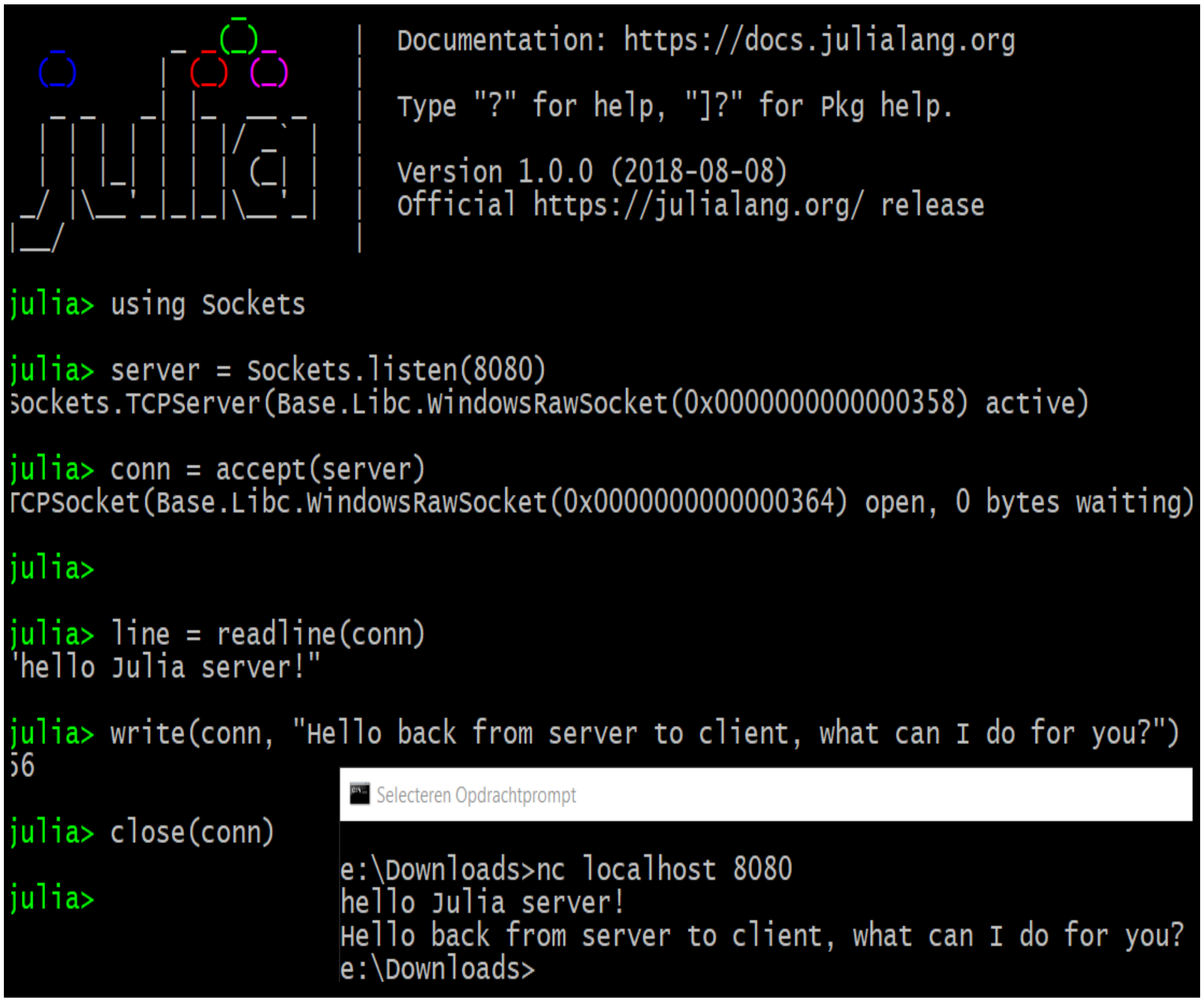
Julia can work with other human-readable file formats through specialized packages:

- For JSON, use the `JSON` package. The `parse` method converts JSON strings into dictionaries, and the `json` method turns any Julia object into a JSON string.
- For XML, use the `LightXML` package.
- For YAML, use the `YAML` package.
- For HDF5 (a common format for scientific data), use the `HDF5` package.
- For working with Windows `INI` files, use the `IniFile` package.

Working with TCP sockets and servers

To send data over a network, the data has to conform to a certain format or protocol. The **Transmission Control Protocol / Internet Protocol** (**TCP/IP**) is one of the core protocols to be used on the internet.

The following screenshot shows how to communicate over TCP/IP between a Julia TCP server and a client (see the code in `Chapter 8\tcpserver.jl`):



```
Documentation: https://docs.julialang.org

Type "?" for help, "]?" for Pkg help.

Version 1.0.0 (2018-08-08)
Official https://julialang.org/ release

julia> using Sockets

julia> server = Sockets.listen(8080)
Sockets.TCPServer(Base.Libc.WindowsRawSocket(0x0000000000000358) active)

julia> conn = accept(server)
TCPSocket(Base.Libc.WindowsRawSocket(0x0000000000000364) open, 0 bytes waiting)

julia>

julia> line = readline(conn)
"hello Julia server!"

julia> write(conn, "Hello back from server to client, what can I do for you?")
56

julia> close(conn)

julia>
```

Selecteren Opdrachtprompt

```
e:\Downloads>nc localhost 8080
hello Julia server!
Hello back from server to client, what can I do for you?
e:\Downloads>
```

The server (in the upper-left corner) is started in a Julia session with `server = Sockets.listen(8080)`, which returns a `TcpServer` object listening on port `8080`. The `conn = accept(server)` line waits for an incoming client to make a connection. In a second terminal (in the lower-right corner), we start the **netcat** (**nc**) tool at the prompt to make a connection with the Julia server on port `8080`, for example, `nc localhost 8080`. Then, the `accept` function creates a `TcpSocket` object on which the server can read or write.

Then, the server issues the `line = readline(conn)` command, blocking the server until it gets a full line (ending with a newline character) from the client. The client types `"hello Julia server!"` followed by *Enter*, which appears at the server console. The server can also write text to the client over the TCP connection with the `write(conn, "message ")` function, which then appears at the client side. The server can, when finished, close the `TcpSocket` connection to close the TCP connection with `close(conn)`; this also closes the `netcat` session.

Of course, a normal server must be able to handle multiple clients. Here, you can see the code for a server that echoes back to the clients everything they send to the server:

```
// code in Chapter8\echoserver.jl
using Sockets
server = Sockets.listen(8080)
while true
  conn = accept(server)  @async begin
    try
      while true
        line = readline(conn)
        println(line)  # output in server console
        write(conn,line)
      end
    catch ex
      print("connection ended with error $ex")
    end
  end # end coroutine block
end
```

To achieve this, we place the `accept()` function within an infinite `while` loop, so that each incoming connection is accepted. The same is true for reading and writing to a specific client; the server only stops listening to that client when the client disconnects. Because the network communication with the clients is a possible source of errors, we have to surround it within a `try`/`catch` expression. When an error occurs, it is bound to the `ex` object. For example, when a client terminal exits, you get the `connection ended with error ErrorException("stream is closed or unusable")` message.

However, we also see an `@async` macro here; what is its function? The `@async` macro starts a new coroutine (refer to the *Tasks* section in Chapter 4, *Control Flow*) in the local process to handle the execution of the `begin...end` block that starts right after it. So, the `@async` macro handles the connection with each particular client in a separate coroutine. Thus, the `@async` block returns immediately, enabling the server to continue accepting new connections through the outer `while` loop. Because coroutines have a very low overhead, making a new one for each connection is perfectly acceptable. If it weren't for the `async` block, the program would block it until it was done with its current client before accepting a new connection.

On the other hand, the `@sync` macro is used to enclose a number of `@async` (or `@spawn` or `@parallel` calls, refer to the *Parallel operations and computing* section), and the code execution waits at the end of the `@sync` block until all the enclosed calls are finished.

Start this server example by typing the following command:

```
julia echoserver.jl
```

We can experiment with a number of netcat sessions in separate terminals. Client sessions can also be made by typing in a Julia console:

```
  using Sockets
  conn = Sockets.connect(8080)
#> TCPSocket(Base.Libc.WindowsRawSocket(0x0000000000000340) open, 0 bytes waiting)
  write(conn, "Do you hear me?\n")
```

The `listen` function has some variants, for example, `listen(IPv6(0),2001)` creates a TCP server that listens on port `2001` on all IPv6 interfaces. Similarly, instead of `readline`, there are also simpler `read` methods:

- `read(conn, UInt8)`: This method blocks until there is a byte to read from `conn`, and then returns it. Use `convert(Char, n)` to convert a `UInt8` value into `Char`. This will let you see the ASCII letter for `UInt8` you read in.
- `read(conn, Char)`: This method blocks until there is a byte to read from `conn`, and then returns it.

The important aspect about the communication API is that the code looks like synchronous code executing line by line, even though the I/O is actually happening asynchronously through the use of tasks. We don't have to worry about writing callbacks as in some other languages. For more details about possible methods, refer to the *I/O and Network* section at `https://docs.julialang.org/en/latest/base/io-network/`.

Interacting with databases

Open Database Connectivity (**ODBC**) is a low-level protocol for establishing connections with the majority of databases and datasources (for more details, refer to `http://en.wikipedia.org/wiki/Open_Database_Connectivity`).

Julia has an `ODBC` package that enables Julia scripts to talk to ODBC data sources. Install the package through `Pkg.add("ODBC")`, and at the start of the code, run it `using ODBC`.

The package can work with a system **Data Source Name** (**DSN**) that contains all the concrete connection information, such as server name, database, credentials, and so on. Every operating system has its own utility to make DSNs. In Windows, the ODBC administrator can be reached by navigating to Control Panel | Administrative Tools | ODBC Data Sources; on other systems, you have IODBC or Unix ODBC.

For example, suppose we have a database called `pubs` running in a SQL Server or a MySQL Server, and the connection is described with a DSN `pubsODBC`. (Included in the code download is a script, `instpubs.sql`, to install the `pubs` database with only the `titles` table in a SQL Server; the script can be easily adapted for MySQL.)

Now, I can connect to this database as follows:

```
// code in Chapter 8\odbc.jl
using ODBC
ODBC.DSN("pubsODBC",<user>,<password>)
```

This returns an output as follows:

```
Connection Data Source: pubsODBC
pubsODBC Connection Number: 1
    Contains resultset? No
```

You can also store this DSN object in a `dsn` variable, as follows:

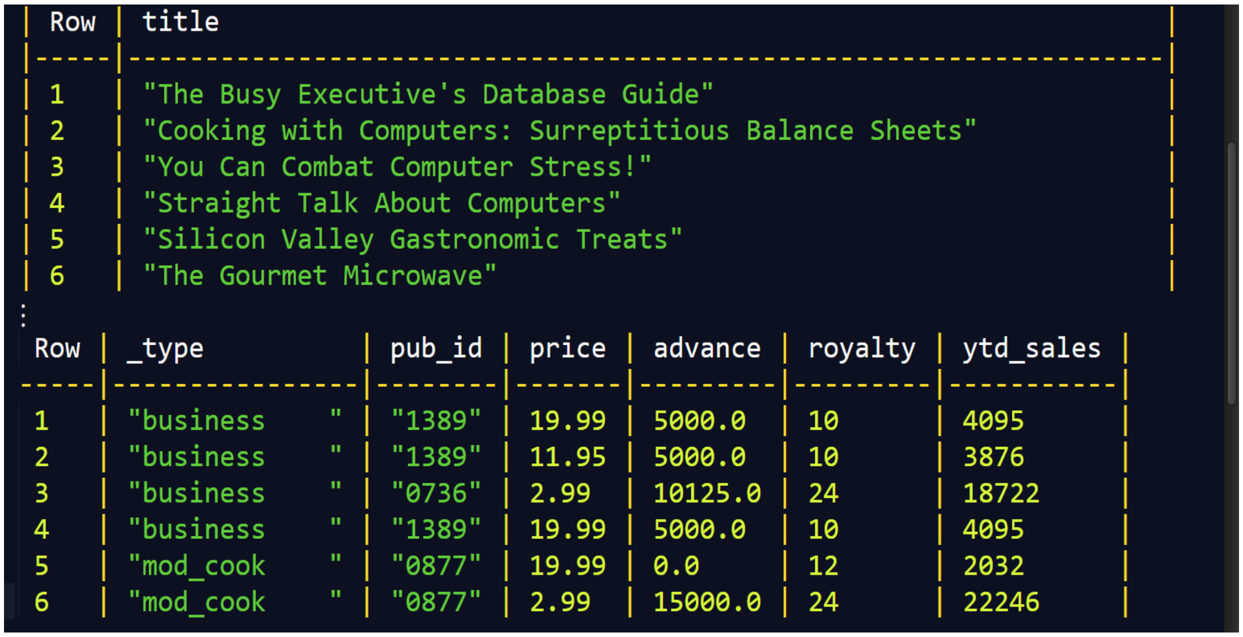
```
dsn = ODBC.DSN("pubsODBC",<user>,<password>)
```

This way, you are able to close the connection when necessary through `ODBC.disconnect!(dsn)` to save database resources, or handle multiple connections.

To launch a query on the `titles` table, you only need to use the `query` function, as follows:

```
results = ODBC.query(dsn, "select * from titles")
```

The result is of the `DataFrame` type and the dimensions are 18 x 10, because the table contains 18 rows and 10 columns, for example; here are some of the columns:



```
| Row | title                                                           |
|-----|-----------------------------------------------------------------|
| 1   | "The Busy Executive's Database Guide"                           |
| 2   | "Cooking with Computers: Surreptitious Balance Sheets"          |
| 3   | "You Can Combat Computer Stress!"                               |
| 4   | "Straight Talk About Computers"                                 |
| 5   | "Silicon Valley Gastronomic Treats"                             |
| 6   | "The Gourmet Microwave"                                         |
⋮
 Row | _type              | pub_id | price | advance | royalty | ytd_sales |
-----|--------------------|--------|-------|---------|---------|-----------|
 1   | "business        " | "1389" | 19.99 | 5000.0  | 10      | 4095      |
 2   | "business        " | "1389" | 11.95 | 5000.0  | 10      | 3876      |
 3   | "business        " | "0736" | 2.99  | 10125.0 | 24      | 18722     |
 4   | "business        " | "1389" | 19.99 | 5000.0  | 10      | 4095      |
 5   | "mod_cook        " | "0877" | 19.99 | 0.0     | 12      | 2032      |
 6   | "mod_cook        " | "0877" | 2.99  | 15000.0 | 24      | 22246     |
```

If you haven't stored the query results in a variable, you can always retrieve them from `conn.resultset`, where `conn` is an existing connection. Now we have all the functionalities of `DataFrames` at our disposal to work with this data. Launching data-manipulation queries works in the same way:

```
updsql = "update titles set type = 'psychology' where
   title_id='BU1032'"
stmt = ODBC.prepare(dsn, updsql)
ODBC.execute!(stmt)
```

In order to see which ODBC drivers are installed on the system, ask for `ODBC.listdrivers()`. The already available DSNs are listed with `ODBC.listdsns()`.

Julia already has database drivers for Memcache, FoundationDB, MongoDB, Redis, MySQL, SQLite, and PostgreSQL (for more information, refer to `https://github.com/JuliaDatabases`).

Parallel operations and computing

In our multicore CPU and clustered computing world, it is imperative for a new language to have excellent parallel computing capabilities. This is one of the main strengths of Julia: providing an environment based on message-passing between multiple processes that can execute on the same machine or on remote machines.

In that sense, it implements the actor model (as Erlang, Elixir, and Pony do), but we'll see that the actual coding happens on a higher level than receiving and sending messages between processes, or workers (processors) as Julia calls them. The developer only needs to explicitly manage the main process from which all other workers are started. The message send and receive operations are simulated by higher-level operations that look like function calls.

Creating processes

To start with processes, add and make available the `Distributed` package with `add Distributed`, and `using Distributed`.

Julia can be started as a REPL or as a separate application with a number of workers, `n`, available. The following command starts `n` processes on the local machine (this command includes the `Distributed` package automatically):

```
// code in Chapter 8\parallel.jl
julia -p n   # starts REPL with n workers
```

These workers are different processes, not threads, so they do not share memory.

To get the most out of a machine, set `n` equal to the number of processor cores. For example, when `n` is `8`, you have, in fact, nine workers: one for the REPL shell itself, and eight others that are ready to do parallel tasks. Every worker has its own integer identifier, which we can see by calling the `workers` function, `workers()`. This returns the following:

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

Process `1` is the REPL worker. We can now iterate over the workers with the following command:

```
for pid in workers()
  # do something with each process (pid = process id)
end
```

Each worker can get its own process ID with the `myid()` function. If you need more workers, adding new ones is easy:

```
addprocs(5)
```

This returns `5-element Array{Any,1}`, which contains their process identifiers, `10 11 12 13 14`. The default method adds workers on the local machine, but the `addprocs` method accepts arguments to start processes on remote machines via

SSH. This is the secure shell protocol that enables you to execute commands on a remote computer via a shell in a totally encrypted manner.

The number of available workers is given by `nprocs()`; in our case, this is `14`. A worker can be removed by calling `rmprocs()` with its identifier; for example, `rmprocs(3)` stops the worker with the ID `3`.

All these workers communicate via TCP ports and run on the same machine, which is why it is called a local cluster. To activate workers on a cluster of computers, start Julia as follows:

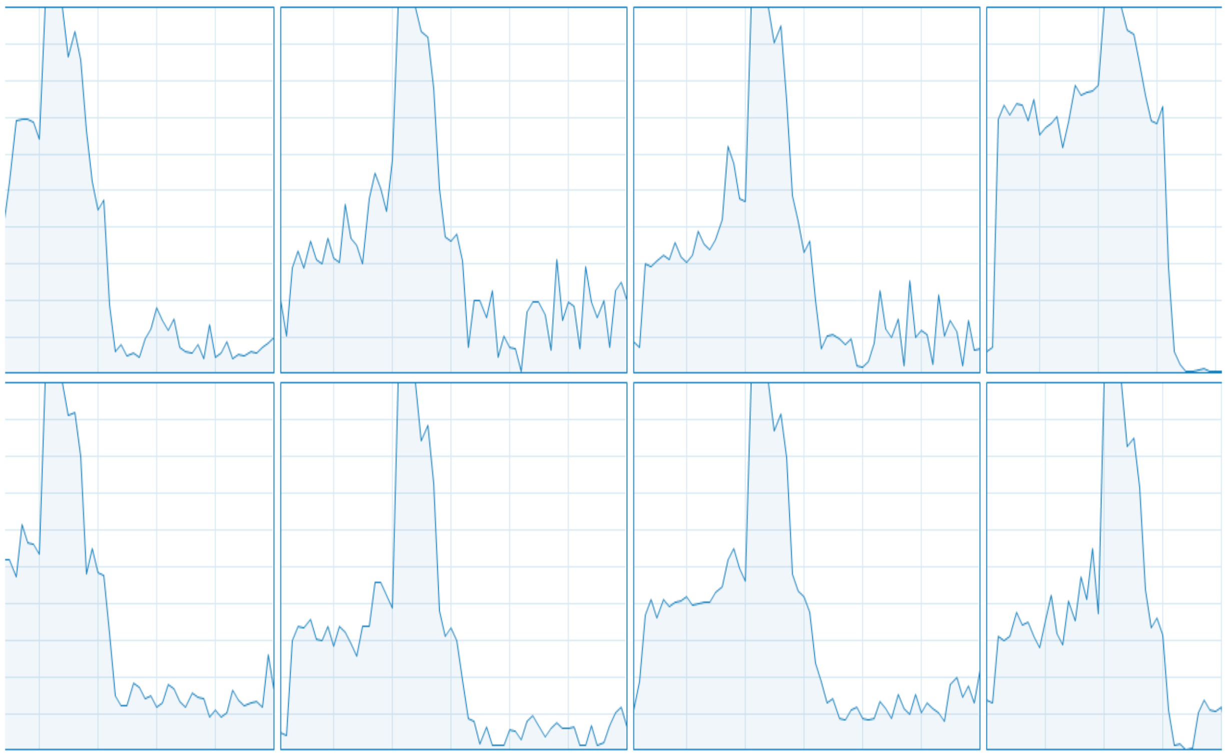
```
julia --machine-file machines driver.jl
```

Here, `machines` is a file that contains the names of the computers you want to engage, as follows:

```
node01
node01
node02
node02
node03
```

Here `node01`, `node02`, and `node03` are the three names of computers in the cluster, and we want to start two workers each on `node01` and `node02`, and one worker on `node03`.

The `driver.jl` file is the script that runs the calculations and has a process identifier of `1`. This command uses a password-less SSH login to start the worker processes on the specified machines. The following screenshot shows all the eight processors on an eight-core machine when engaged in a parallel operation:



The horizontal axis is time, and the vertical is the CPU usage. On each core, a worker process is engaged in a long-running Fibonacci calculation.

Processors can be dynamically added or removed to a master Julia process, locally on symmetric multiprocessor systems, remotely on a computer cluster, as well as in the cloud. If more versatility is needed, you can work with the `ClusterManager` type (see http://docs.julialang.org/en/latest/manual/parallel-computing/).

Using low-level communications

Julia's native parallel computing model is based on two primitives: **remote calls** and **remote references**. At this level, we can give a certain worker a function with arguments to execute with `remotecall`, and get the result back with fetch. As a trivial example in the following code, we call upon worker 2 to execute a square function on the number `1000`:

```
r1 = remotecall(x -> x^2, 2, 1000)
```

This returns `Future(2, 1, 15, nothing)`.

The arguments are: the worker ID, the function, and the function's arguments. Such a remote call returns immediately, thus not blocking the main worker (the REPL in this case). The main process continues executing while the remote worker does the assigned job. The `remotecall` function returns a variable, `r1`, of the `Future` type, which is a reference to the computed result, which we can get using `fetch`:

```
fetch(r1)
```

This returns `1000000`.

The call to `fetch` will block the main process until worker 2 has finished the calculation. The main processor can also run `wait(r1)`, which also blocks until the result of the remote call becomes available. If you need the remote result immediately in the local operation, use the following command:

```
remotecall_fetch(sin, 5, 2pi)
```

Which returns `-2.4492935982947064e-16`.

This is more efficient than `fetch(remotecall(..))`.

You can also use the `@spawnat` macro, which evaluates the expression in the second argument on the worker specified by the first argument:

```
r2 = @spawnat 4 sqrt(2) # lets worker 4 calculate sqrt(2)
  fetch(r2)  # returns 1.4142135623730951
```

This is made even easier with `@spawn`, which only needs an expression to evaluate, because it decides for itself where it will be executed: `r3 = @spawn sqrt(5)` returns `RemoteRef(5,1,26)` and `fetch(r3)` returns `2.23606797749979`.

To execute a certain function on all the workers, we can use a comprehension:

```
r = [@spawnat w sqrt(5) for w in workers()]
fetch(r[3]) # returns 2.23606797749979
```

To execute the same statement on all the workers, we can also use the `@everywhere` macro:

```
@everywhere println(myid()) 1
        From worker 2: 2
        From worker 3: 3
        From worker 4: 4
        From worker 7: 7
        From worker 5: 5
        From worker 6: 6
        From worker 8: 8
        From worker 9: 9
```

All the workers correspond to different processes; they therefore do not share variables, for example:

```
x = 5 #> 5
@everywhere println(x) #> 5
  # exception on worker 2: ERROR: UndefVarError: x not defined ...
       ...and 11 more exception(s)
```

The `x` variable is only known in the main process, all the other workers return the `ERROR: x not defined` error message.

`@everywhere` can also be used to make the data, such as the `w` variable, available to all processors, for example, `@everywhere w = 8`.

The following example makes a `defs.jl` source file available to all the workers:

```
@everywhere include("defs.jl")
```

Or, more explicitly, a `fib(n)` function, as follows:

```
@everywhere function fib(n)
  if (n < 2) then
    return n
  else return fib(n-1) + fib(n-2)
  end
end
```

In order to be able to perform its task, a remote worker needs access to the function it executes. You can make sure that all workers know about the functions they need by loading the `functions.jl` source code with `include`, making it available to all workers:

```
include("functions.jl")
```

In a cluster, the contents of this file (and any files loaded recursively) will be sent over the network.

A best practice is to separate your code into two files: one file (`functions.jl`) that contains the functions and parameters that need to be run in parallel, and the other file (`driver.jl`) that manages the processing and collects the results. Use the `include("functions.jl")` command in `driver.jl` to import the functions and parameters to all processors.

An alternative is to specify that the files load on the command line. If you need the `file1.jl` and `file2.jl` source files on all the `n` processors at startup time, use the `julia -p n -L file1.jl -L file2.jl driver.jl` syntax, where `driver.jl` is the script that organizes the computations.

Data-movement between workers (such as when calling `fetch`) needs to be reduced as much as possible in order to get performance and scalability.

If every worker needs to know the `d` variable, this can be broadcast to all processes with the following code:

```
for pid in workers()
    remotecall(pid, x -> (global d; d = x; nothing), d)
end
```

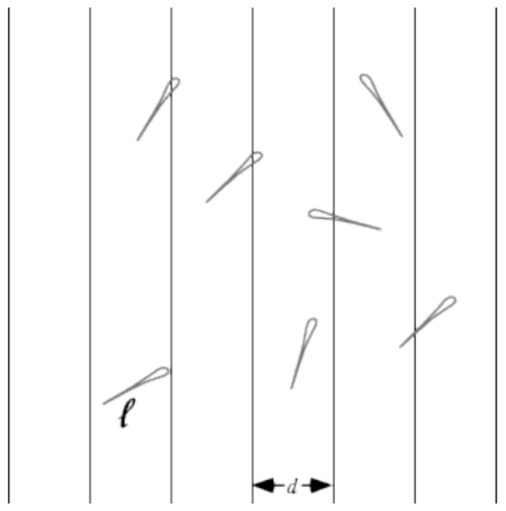
Each worker then has its local copy of data. Scheduling the workers is done with tasks (refer to the *Tasks* section of Chapter 4, *Control Flow*), so that no locking is required; for example, when a communication operation such as `fetch` or `wait` is executed, the current task is suspended, and the scheduler picks another task to run. When the wait event completes (for example, the data shows up), the current task is restarted.

In many cases, however, you do not have to specify or create processes to do parallel programming in Julia, as we will see in the next section.

Parallel loops and maps

A `for` loop with a large number of iterations is a good candidate for parallel execution, and Julia has a special construct to do this: the `@parallel` macro, which can be used for the `for` loops and comprehensions.

Let's calculate an approximation for `π` using the famous Buffon's needle problem. If we drop a needle onto a floor with equal parallel strips of wood, what is the probability that the needle will cross a line between two strips? Let's take a look at the following screenshot:



Without getting into the mathematical intricacies of this problem (if you are interested, see http://en.wikipedia.org/wiki/Buffon's_needle), a `buffon(n)` function can be deduced from the model assumptions that returns an approximation for `π` when throwing the needle `n` times (assuming the length of the needle, `l`, and the width, `d`, between the strips both equal `1`):

```
// code in Chapter 8\parallel_loops_maps.jl
function buffon(n)
  hit = 0
  for i = 1:n
    mp = rand()
    phi = (rand() * pi) - pi / 2 # angle at which needle falls
    xright = mp + cos(phi)/2 # x location of needle
    xleft = mp - cos(phi)/2
    # does needle cross either x == 0 or x == 1?
    p = (xright >= 1 || xleft <= 0) ? 1 : 0
    hit += p
  end
  miss = n - hit
  piapprox = n / hit * 2
end
```

With ever-increasing `n`, the calculation time increases, because the number of the `for` iterations that have to be executed in one thread on one processor increases, but we also get a better estimate for `π`:

```
@time buffon(100000)
   0.208500 seconds (504.79 k allocations: 25.730 MiB, 7.10% gc time)
 3.1441597233139444

@time buffon(100000000)
  4.112683 seconds (5 allocations: 176 bytes)
   3.141258861373451
```

However, what if we could spread the calculations over the available processors? For this, we have to rearrange our code a bit. In the sequential version, the variable hit is increased on every iteration inside the `for` loop with the `p` amount (which is `0` or `1`). In the parallel version, we rewrite the code, so that this `p` is exactly the result of the `for` loop (one calculation) done on one of the involved processors.

Julia also provides a `@distributed` macro that acts on a `for` loop, splitting the range and distributing it to each process. It optionally takes a "reducer" as its first argument. If a reducer is specified, the results from each remote procedure will be aggregated using the reducer. In the following example, we use the `(+)` function as a reducer, which means that the last values of the parallel blocks on each worker will be summed to calculate the final value of `hit`:

```
function buffon_par(n)
  hit = @distributed (+) for i = 1:n
      mp = rand()
      phi = (rand() * pi) - pi / 2
      xright = mp + cos(phi)/2
      xleft = mp - cos(phi)/2
       (xright >= 1 || xleft <= 0) ? 1 : 0
    end
  miss = n - hit
  piapprox = n / hit * 2
end
```

On my machine with eight processors, this gives the following results:

```
@time buffon_par(100000)
  1.058487 seconds (951.35 k allocations: 48.192 MiB, 2.04% gc time)
 3.15059861373661

@time buffon_par(100000000)
  0.735853 seconds (1.84 k allocations: 133.156 KiB)
 3.1418106012575633
```

We see much better performance for the higher number of iterations (a factor of `5.6` in this case). By changing a normal `for` loop into a parallel-reducing version, we were able to get substantial improvements in the calculation time, at the cost of higher memory consumption. In general, always test whether the parallel version really is an improvement over the sequential version in your specific case!

The first argument of `@distributed` is the reducing operator (here, `(+)`), the second is the `for` loop, which must start on the same line. The calculations in the loop must be independent of one another, because the order in which they run is arbitrary, given that they are scheduled over the different workers. The actual reduction (summing up in this case) is done on the calling process.

Any variables used inside the parallel loop will be copied (broadcasted) to each process. Because of this, the code, such as the following, will fail to initialize the `arr` array, because each process has a copy of it:

```
arr = zeros(100000)
@distributed for i=1:100000
  arr[i] = i
end
```

Afterward the loop, `arr` still contains all the zeros, because it is the copy on the master worker.

If the computational task is to apply a function to all elements in some collection, you can use a **parallel map** operation through the `pmap` function. The `pmap` function takes the following form: `pmap(f, coll)`, applies an `f` function on each element of the `coll` collection in parallel, but preserves the order of the collection in the result. Suppose we have to calculate the rank of a number of large matrices. We can do this sequentially, as follows:

```
using LinearAlgebra
function rank_marray()
  marr = [rand(1000,1000) for i=1:10]
  for arr in marr
      println(LinearAlgebra.rank(arr))
  end
end

@time rank_marray() # prints out ten times 1000
7.310404 seconds (91.33 k allocations: 162.878 MiB, 1.15% gc time)
```

In the following, parallelizing also gives benefits (a factor of `1.6`):

```
function prank_marray()
  marr = [rand(1000,1000) for i=1:10]
  println(pmap(LinearAlgebra.rank, marr))
end

@time prank_marray()
5.966216 seconds (4.15 M allocations: 285.610 MiB, 2.15% gc time)
```

The `@distributed` macro and `pmap` are both powerful tools to tackle **map-reduce** problems.

Julia's model for building a large parallel application works by means of a global distributed address space. This means that you can hold a reference to an object that lives on another machine participating in a computation. These references are easily manipulated and passed around between machines, making it simple to keep track of what's being computed where. Also, machines can be added in mid-computation when needed.

Summary

In this chapter, we explored a lot of material. We learned how the I/O system in Julia is constructed, how to work with files and `DataFrames`, and how to connect with databases using ODBC. The basics of network programming in Julia were also discussed, and then we got an overview of the parallel computing functionality, from primitive operations to map-reduce functions and distributed arrays.

In the next chapter, we will take a look at how Julia interacts with the command line and with other languages, and discuss a number of performance tips.

Running External Programs

Sometimes, your code needs to interact with programs in the outside world, be it the operating system in which it runs or other languages such as C or Fortran. This chapter shows how straightforward it is to run external programs from Julia and covers the following topics:

- Running shell commands—interpolation and pipelining
- Calling C and Fortran
- Calling Python
- Performance tips

Running shell commands

To interact with the operating system from within the Julia REPL, there are a few helper functions available, as follows:

- `pwd()`: this function prints the current directory, for example, `"d:\\test"`
- `cd("d:\\test\\week1")`: this function helps to navigate to subdirectories
- `;`: in the interactive shell, you can also use shell mode using the `;` modifier, for example: `; cd folder`: navigates to `folder`

However, what if you want to run a shell command by using the operating system (the OS)? Julia offers efficient shell integration through the `run` function, which takes an object of type `Cmd`, defined by enclosing a command string in backticks (``).

The following are some examples for Linux or macOS X (at the time of writing: September 2018):

```
# Code in Chapter 9\shell.jl:
cmd = `echo Julia is smart`
   typeof(cmd) #> Cmd
   run(cmd) # returns Julia is smart
   run(`date`) #> Sat Jul 14 09:44:50 GMT 2018
   cmd = `cat file1.txt`
   run(cmd) # prints the contents of file1.txt
```

The preceding code does not work on Windows, although it worked until version 0.6. This is a bug in version 1.0 and is expected to be resolved in the near future.

Be careful to enclose the command text in backticks (`), not single quotes (').

If the execution of `cmd` by the OS goes wrong, `run` throws a `failed process` error. You might want to test the command first before running it; `success(cmd)` will return `true` if it executes successfully, otherwise it returns `false`.

Julia forks commands as **child processes** from the Julia process. Instead of immediately running the command in the shell, backticks create a `Cmd` object to represent the command. This can then be run, connected to other commands via pipes, and read or written to.

Interpolation

String interpolation with the `$` operator is allowed in a command object, like this:

```
  file = "file1.txt"
cmd = `cat $file` # equivalent to `cat file1.txt`
run(cmd) #> prints the contents of file1.txt
```

This is very similar to the string interpolation with `$` in strings (refer to the *Strings* section in `Chapter 2`, *Variables, Types, and Operations*).

Pipelining

Julia defines a `pipeline` function to redirect the output of a command as the input to the following command:

```
run(pipeline(`cat $file`,"test.txt"))
```

This writes the contents of the file referred to by `$file` into `test.txt`, which is shown as follows:

```
run(pipeline("test.txt",`cat`))
```

This `pipeline` function can even take several successive commands, as follows:

```
run(pipeline(`echo $("\nhi\nJulia")`,`cat`,`grep -n J`)) #>
       3:Julia
```

If the file `tosort.txt` contains `B`, `A`, and `C` on consecutive lines, then the following command will sort the lines:

```
run(pipeline(`cat "tosort.txt"`,`sort`))  # returns A B C
```

Another example is to search for the word `is` in all the text files in the current folder. Use the following command:

```
run(`grep is $(readdir())`)
```

To capture the result of a command in Julia, use `read` or `readline`:

```
a = read(pipeline(`cat "tosort.txt"`,`sort`))
```

Now `a` has the value `A\r\nB\r\nC\n`.

Multiple commands can be run in parallel with the `&` operator:

```
run(`cat "file1.txt"` & `cat "tosort.txt"`)
```

This will print the lines of the two files intermingled, because the printing happens concurrently.

Using this functionality requires careful testing, and, probably, the code will differ according to the operating system on which your Julia program runs. You can obtain the OS from the variable `Sys.KERNEL`, or use the functions `iswindows`, `isunix`, `islinux`, and `isosx` from the `Sys` package, which was specifically designed to handle platform variations. For example, let's say we want to execute the function `fun1()` (unless we are on Windows, in which case the function is `fun2()`). We can write this as follows:

```
Sys.iswindows() ? fun1 : fun2
```

Or we can write it with the more usual `if...else` keyword:

```
if Sys.iswindows()
    fun1
else
    fun2
end
```

Calling C and Fortran

While Julia can rightfully claim to obviate the need to write some C or Fortran code, it is possible that you will need to interact with the existing C or Fortran shared libraries. Functions in such a library can be called directly by Julia, with no glue code, boilerplate code, or compilation needed. Because Julia's LLVM compiler generates native code, calling a C function from Julia has exactly the same overhead as calling the same function from C code itself. However, first, we need to know a few more things:

- For calling out to C, we need to work with pointer types; a native pointer `Ptr{T}` is nothing more than the memory address for a variable of type `T`. You can use `Cstring` if the value is null-terminated.
- At this lower level, the term `primitive` is also used. `primitive` is a concrete type whose data consists of bits, such as `Int8`, `UInt8`, `Int32`, `Float64`, `Bool`, and `Char`.
- To pass a string to C, it is converted to a contiguous byte array representation with the function `unsafe_string()`; given `Ptr` to a C string, it returns a Julia string.

Here is how to call a C function in a shared library (calling Fortran is done similarly). Suppose we want to know the value of an environment variable in our system, say, the language; we can obtain this by calling the C function `getenv` from the shared library `libc`:

```
# code in Chapter 9\callc.jl:
lang = ccall( (:getenv, "libc"), Cstring, (Cstring,), "LANG")
```

This returns a `Cstring`. To see its string contents, execute `unsafe_string(lang)`, which returns `en_US`.

In general, `ccall` takes the following arguments:

- A (`:function, "library"`) tuple, where the name of the C function (here, `getenv`) is used as a symbol, and the library name (here, `libc`) as a string
- The return type (here, `Cstring`), which can also be any `primitive`, or `Ptr`

- A `Cstring` as input arguments: note the tuple notation `(Cstring,)`
- The actual arguments, if there are any (here, `"LANG"`)

It is generally advisable to test for the existence of a library before doing the call. This can be tested like this: `find_library(["libc"])`, which returns `"libc"` when the library is found, or `" "` when it cannot find the library.

When calling a Fortran function, all inputs must be passed by reference. Arguments to C functions are, in general, automatically converted, and the returned values in C types are also converted to Julia types. Arrays of Booleans are handled differently in C and Julia and cannot be passed directly, so they must be manually converted. The same applies for some system-dependent types.

The `ccall` function will also automatically ensure that all of its arguments will be preserved from garbage collection until the call returns. C types are mapped to Julia types. For example, `short` is mapped to `Int16`, and `double` to `Float64`.

A complete table of these mappings, as well as a lot more intricate details, can be found in the Julia docs at http://docs.julialang.org/en/latest/manual/calling-c-and-fortran-code/. The other way around is also possible, by calling Julia functions from C code (or embedding Julia in C); refer to http://docs.julialang.org/en/latest/manual/embedding/. Julia and C can also share array data without copying.

If you have the existing C code, you must compile it as a shared library to call it from Julia. With GCC, you can do this using the `-shared -fPIC` command-line arguments. Support for C++ is more limited and is provided by the `Cpp` and `Clang` packages.

Calling Python

The `PyCall` package provides us with the ability to call Python from Julia code. As always, add this package to your Julia environment with `add PyCall`. Then, you can start using it in the REPL, or in a script as follows:

```
using PyCall
py"10*10"             #> 100
@pyimport math
math.sin(math.pi / 2) #> 1.0
```

As we can see with the `@pyimport` macro, we can easily import any Python library; functions inside such a library are called with the familiar dot notation.

For more details, refer to https://github.com/stevengj/PyCall.jl.

Performance tips

Throughout this book, we have paid attention to performance. Here, we summarize some highlighted performance topics and give some additional tips. These tips need not always be used, and you should always benchmark or profile the code and the effect of a tip. However, applying some of them can often yield a remarkable performance improvement. Using type annotations everywhere is certainly *not* the way to go; Julia's type inferring engine does that work for you:

- **Refrain from using global variables**. If unavoidable, make them constant with `const`, or at least annotate the types. It is better to use local variables instead; they are often only kept on the stack (or even in registers), especially if they are immutable.
- Use a `main()` function to structure your code.
- Use functions that do their work on local variables via function arguments, rather than mutating global objects.
- Type stability is very important:
  - Avoid changing the types of variables over time
  - The return type of a function should only depend on the type of the arguments

  Even if you do not know the types that will be used in a function, but you do know it will always be of the same type `T`, then functions should be defined keeping that in mind, as in the following code snippet:

```
function myFunc(a::T, c::Int) where T
  # code
end
```

- If large arrays or dictionaries are needed, indicate their final size with `sizehint!` from the start (refer to the *Ranges and arrays* section of Chapter 2, *Variables, Types, and Operations*). The following is an example of its use:

```
d1 = Dict();
sizehint!(d1, 10000);
for i in [1:10000] d1[string(i)] = 2*i; end;
```

- If `arr` is a very large array that you no longer need, you can free the memory it occupies by setting `arr = nothing`. The occupied memory will be released the next time the garbage collector runs. You can force this to happen by invoking `GC.gc()`.
- In certain cases (such as real-time applications), disabling garbage collection (temporarily) with `GC.enable(false)` can be useful.
- Use named functions instead of anonymous functions.
- In general, use small functions.
- Don't test for the types of arguments inside a function, use an argument type annotation instead.
- If necessary, code different versions of a function (several methods) according to the types, so that multiple dispatch applies. Normally, this won't be necessary, because the JIT compiler is optimized to deal with types as they come.
- Use types for keyword arguments; avoid using the splat operator (`...`) for dynamic lists of keyword arguments.
- Using mutating APIs (functions with `!` at the end) is helpful, for example, to avoid copying large arrays.
- Prefer array operations to comprehensions, for example, `x.^2` is considerably faster than `[val^2 for val in x]`.
- Don't use `try`/`catch` in the inner loop of a calculation.
- Use immutable types (`cfr.` package `ImmutableArrays`).
- Avoid using type `Any`, especially in collection types.
- Avoid using abstract types in a collection.
- Type annotate fields in composite types.
- Avoid using a large number of variables, large temporary arrays, and collections, because this provokes a great deal of garbage collection. Also, don't make copies of variables if you don't have to.
- Avoid using string interpolation (`$`) when writing to a file, just write the values.
- Devectorize your code, that is, use explicit `for` loops on array elements instead of simply working with the arrays and matrices. (This is the exact opposite of advice commonly given to R, MATLAB, or Python users.)

- If appropriate, use a parallel reducing form with `@distributed` instead of a normal `for` loop (refer to `Chapter 8`, *IO, Networking, and Parallel Computing*).
- Reduce data movement between workers in a parallel execution as much as possible (refer to `Chapter 8`, *IO, Networking, and Parallel Computing*).
- Fix deprecation warnings.
- Use the macro `@inbounds` so that no array bounds checking occurs in expressions (if you are absolutely certain that no `BoundsError` occurs!).
- Avoid using `eval` at runtime.

In general, split your code into functions. Data types will be determined at function calls, and when a function returns. Types that are not supplied will be inferred, but the `Any` type does not translate to efficient code. If types are stable (that is, variables stick to the same type) and can be inferred, then your code will run quickly.

Tools to use

Execute a function with certain parameter values, and then use `@time` (refer to the *Generic functions and multiple dispatch* section in Chapter 3, *Functions*) to measure the elapsed time and memory allocation. If too much memory is allocated, investigate the code for type problems.

Experiment with different tips and techniques in the script `array_product_benchmark.jl`. Use `code_typed` (refer to the *Reflection capabilities* section in Chapter 7, *Metaprogramming in Julia*) to see if type `Any` is inferred.

A **profiler** tool is available in the standard library to measure the performance of your running code and identify possible bottleneck lines. This works through calling your code with the `@profile` macro (refer to https://docs.julialang.org/en/latest/manual/profile/).

The `ProfileView` package provides a nice graphical browser to investigate profile results (follow the tutorial at https://github.com/timholy/ProfileView.jl).

`BenchmarkingTools` is an excellent package with macros and tools for benchmarking your code.

For more tips, examples, and argumentation about performance, look up http://docs.julialang.org/en/latest/manual/performance-tips/.

Summary

In this chapter, we saw how easy it is to run commands at the operating system level. Interfacing with C is not that much more difficult, although it is somewhat specialized. Finally, we reviewed the best practices at our disposal to make Julia perform at its best. In the previous chapter, we got to know some of the more important packages when using Julia in real projects.

The Standard Library and Packages

In this final chapter of our mini tour of Julia, we will look anew at the standard library and explore the ever-growing ecosystem of packages for Julia. We will discuss the following topics:

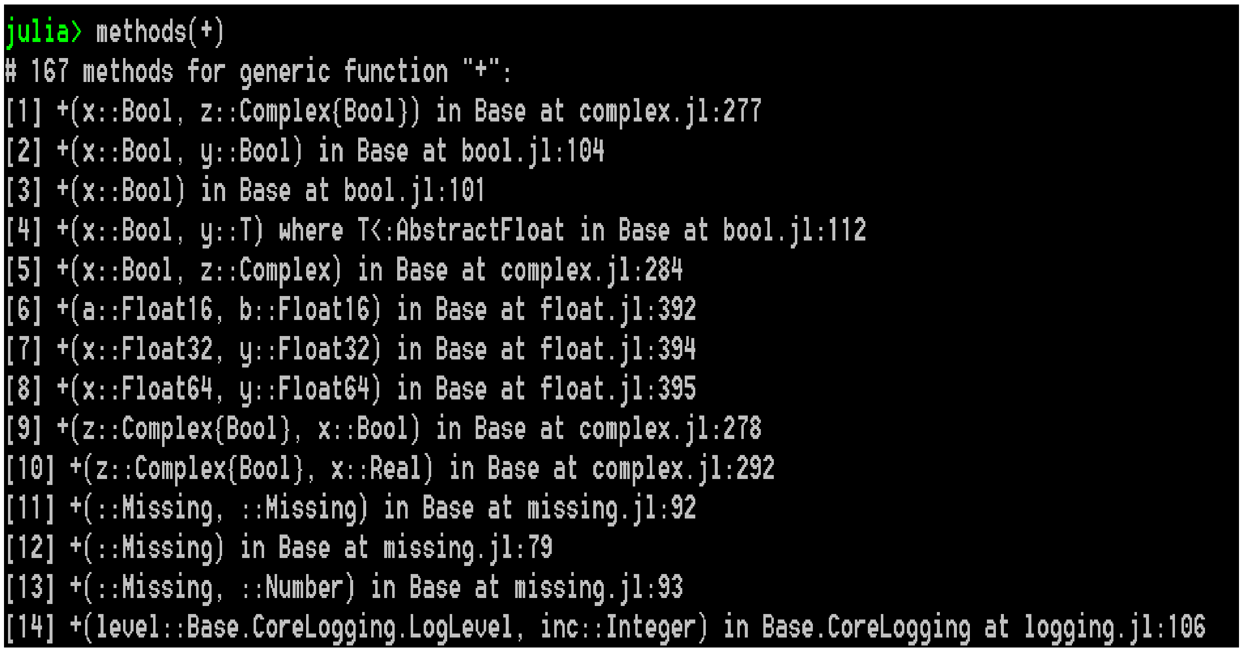
- Digging deeper into the standard library
- Julia's package manager
- Graphics in Julia
- Using Plots on data

Digging deeper into the standard library

The standard library is written in Julia and is comprised of a very broad range of functionalities: from regular expressions, working with dates and times, a package manager, internationalization and Unicode, linear algebra, complex numbers, specialized mathematical functions, statistics, I/O and networking, **Fast Fourier Transformations** (**FFT**), and parallel computing, to macros, and reflection. Julia provides a firm and broad foundation for numerical computing and data science (for example, much of what NumPy has to offer is provided). Despite being targeted at numerical computing and data science, Julia aims to be a general-purpose programming language.

The source code of the standard library can be found in the `share\julia\base` and `share\julia\stdlib` subfolders of Julia's root installation folder. Coding in Julia leads almost naturally to this source code, for example, when viewing all the methods of a particular function with `methods()`, or when using the `@which` macro to find out more about a certain method (refer to the *Generic functions and multiple dispatch* section in `Chapter 3`, *Functions*).

Here, we see the output of the command `methods(+)`, which lists 167+ methods available in Julia version 1.0, together with their source locations:



```
julia> methods(+)
# 167 methods for generic function "+":
[1] +(x::Bool, z::Complex{Bool}) in Base at complex.jl:277
[2] +(x::Bool, y::Bool) in Base at bool.jl:104
[3] +(x::Bool) in Base at bool.jl:101
[4] +(x::Bool, y::T) where T<:AbstractFloat in Base at bool.jl:112
[5] +(x::Bool, z::Complex) in Base at complex.jl:284
[6] +(a::Float16, b::Float16) in Base at float.jl:392
[7] +(x::Float32, y::Float32) in Base at float.jl:394
[8] +(x::Float64, y::Float64) in Base at float.jl:395
[9] +(z::Complex{Bool}, x::Bool) in Base at complex.jl:278
[10] +(z::Complex{Bool}, x::Real) in Base at complex.jl:292
[11] +(::Missing, ::Missing) in Base at missing.jl:92
[12] +(::Missing) in Base at missing.jl:79
[13] +(::Missing, ::Number) in Base at missing.jl:93
[14] +(level::Base.CoreLogging.LogLevel, inc::Integer) in Base.CoreLogging at logging.jl:106
```

The same command in a Jupyter notebook even provides hyperlinks to the source code.

We covered some of the most important types and functions in the previous chapters, and you can refer to the manual for a more exhaustive overview at `https://docs.julialang.org/en/latest/base/base/`.

It is certainly important to know that Julia contains a wealth of functional constructs to work with collections, such as the `reduce`, `fold`, `min`, `max`, `sum`, `any`, `all`, `map`, and `filter` functions. Some examples are as follows:

- `filter(f, coll)` applies the function `f` to all the elements of the collection `coll`:

  ```
  # code in Chapter 10\stdlib.jl:
  filter(x -> iseven(x), 1:10)
  ```

  This returns `5-element Array{Int64,1}`, which consists of `2`, `4`, `6`, `8`, and `10`.

- `mapreduce(f, op, coll)` applies the function `f` to all the elements of `coll` and then reduces this to one resulting value by applying the operation `op`:

  ```
  mapreduce(x -> sqrt(x), +, 1:10) #> 22.4682781862041
  # which is equivalent to:
  sum(map(x -> sqrt(x), 1:10))
  ```

When working in the REPL, it can be handy to store a variable in the operating system's clipboard if you want to clean the REPL's variables memory with `workspace()`. Consider the ensuing example:

```
a = 42
using InteractiveUtils
clipboard(a)
# quit and restart REPL:
a # returns ERROR: a not defined
a = clipboard() # returns "42"
```

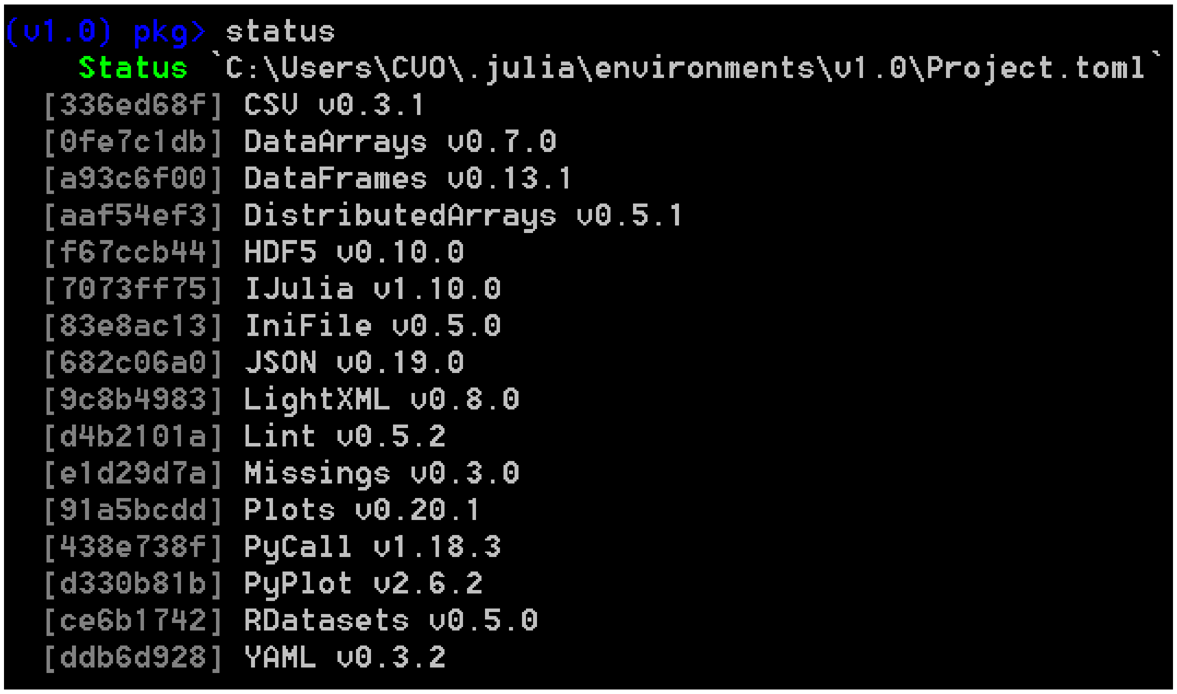
This also works while copying information from another application, for example, a string from a website or from a text editor.

Julia's package manager

The *Packages* section in `Chapter 1`, *Installing the Julia Platform*, introduced us to Julia's package system (some 1,906 packages in September 2018 and counting) and its manager program `Pkg`. Most Julia libraries are written exclusively in Julia; this makes them not only more portable, but also an excellent source for learning and experimenting with Julia in your own modified versions. The packages that are useful for data scientists are `Stats`, `Distributions`, `GLM`, and `Optim`. You can search for applicable packages in the `https://pkg.julialang.org/` repository.

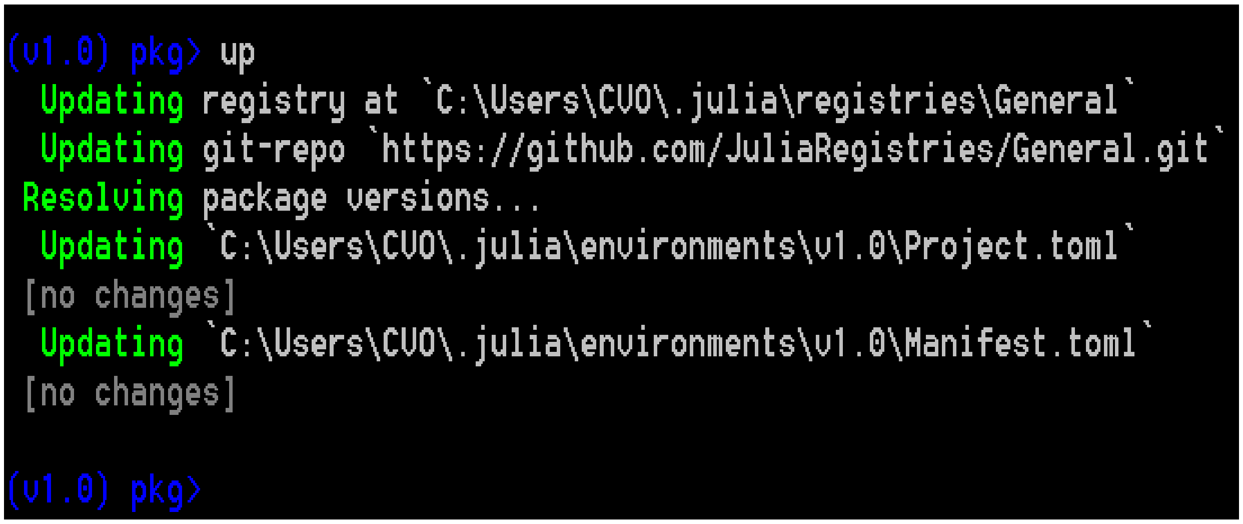
Installing and updating packages

Use the `status` command in the package REPL mode to see which packages have already been installed:



```
(v1.0) pkg> status
    Status `C:\Users\CVO\.julia\environments\v1.0\Project.toml`
  [336ed68f] CSV v0.3.1
  [0fe7c1db] DataArrays v0.7.0
  [a93c6f00] DataFrames v0.13.1
  [aaf54ef3] DistributedArrays v0.5.1
  [f67ccb44] HDF5 v0.10.0
  [7073ff75] IJulia v1.10.0
  [83e8ac13] IniFile v0.5.0
  [682c06a0] JSON v0.19.0
  [9c8b4983] LightXML v0.8.0
  [d4b2101a] Lint v0.5.2
  [e1d29d7a] Missings v0.3.0
  [91a5bcdd] Plots v0.20.1
  [438e738f] PyCall v1.18.3
  [d330b81b] PyPlot v2.6.2
  [ce6b1742] RDatasets v0.5.0
  [ddb6d928] YAML v0.3.2
```

It is advisable to regularly (and certainly before installing a new package) execute the `up` command to ensure that your local package repository is up-to-date and synchronized, as shown in the following screenshot:



```
(v1.0) pkg> up
  Updating registry at `C:\Users\CVO\.julia\registries\General`
  Updating git-repo `https://github.com/JuliaRegistries/General.git`
 Resolving package versions...
  Updating `C:\Users\CVO\.julia\environments\v1.0\Project.toml`
 [no changes]
  Updating `C:\Users\CVO\.julia\environments\v1.0\Manifest.toml`
 [no changes]

(v1.0) pkg>
```

If you only want to update one package, specify the package name after the `up` command.

The `rm` command is used for deleting a package, but it removes only the reference to it. To completely remove the sources, use the `gc` command.

As we saw in `Chapter 1`, *Installing the Julia Platform*, packages are installed via `add PackageName` and brought into scope using `PackageName`. You can also clone a package from a `git` repository as follows:

```
Pkg.clone("git@github.com:ericchiang/ANN.jl.git")
```

If you need to force a certain package to a certain version (perhaps an older version), use `pin`. For example, use `pin HDF5, v"0.4.3"` to force the use of version 0.4.3 of package `HDF5`, even when you already have version 0.4.4 installed.

Graphics in Julia

Several packages exist to plot data and visualize data relations; `Plots` and `PyPlot` are some of the most commonly used:

- `PyPlot`: (refer to the *Installing and working with Jupyter* section in `Chapter 1`, *Installing the Julia Platform*) This package works with no overhead through the `PyCall` package. The following is a summary of the main commands:
  - `plot(y)`, `plot(x,y)` plots y versus 0,1,2,3 or versus `x:loglog(x,y)`
  - `semilogx(x,y)`, `semilogy(x,y)` for log scale plots
  - `title("A title")`, `xlabel("x-axis")`, and `ylabel("foo")` to set labels
  - `legend(["curve 1", "curve 2"], "northwest")` to write a legend at the upper-left
  - `grid()`, `axis( "equal")` adds grid lines, and uses equal `x` and `y` scaling
  - `title(L"the curve $e^\sqrt{x}$")` sets the title with a LaTeX equation
  - `savefig( "fig.png")`, `savefig( "fig.eps")` saves as the PNG or EPS image

- `Plots`: (refer to the *Adding a new package* section in `Chapter 1`, *Installing the Julia Platform*) This is the favorite package in the Julia Computing community. It is a visualization interface and toolset that works with several backends, in particular `GR` (the default backend), `PyPlot`, and `PlotyJS`. To start using a certain backend, type `gr()` or `pyplot()` after you have given the command `using PyPlots`.

  Plot styles can be adapted by so-called **attributes** (documented at `http://docs.juliaplots.org/latest/attributes/`). Some of the most used attributes are:

  - `* xaxis`, `yaxis`, and `zaxis`
  - `line`—to adapt line visualizations
  - `fill`—to fill surfaces with color and transparency
  - The `subplot` category to modify visualization of an entire plot
  - The `plot` category to modify visualization of an entire plot

Lots of other plots can be drawn in `Plots`, such as scatter plots, 2D histograms, and box plots. You can even draw in the REPL if you want to.

Comprehensive documentation and a tutorial can be found here: `http://docs.juliaplots.org/latest/`

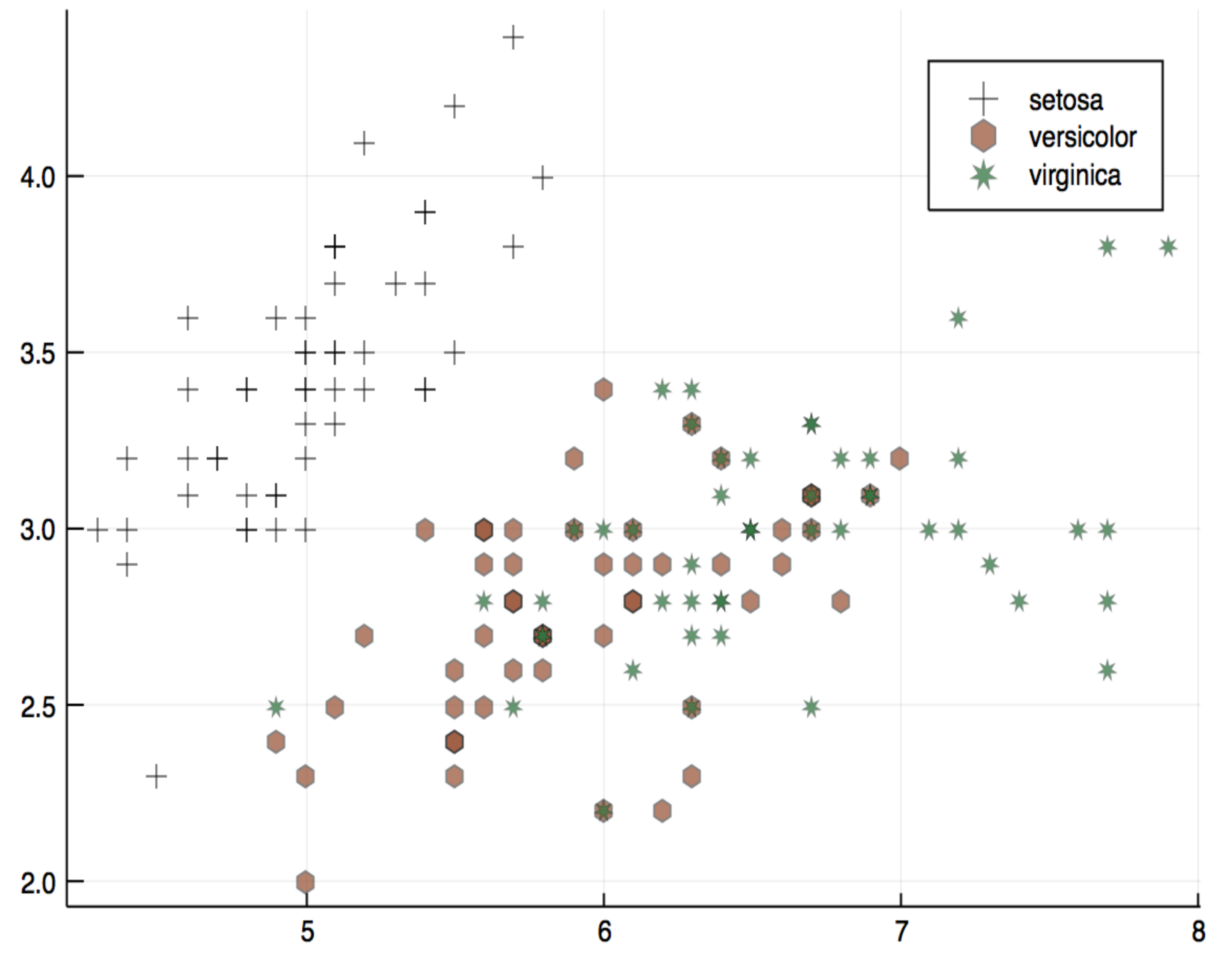
We apply `Plots` to visualize data in the next section.

Using Plots on data

Let's apply `Plots` to show a graph on the famous Iris flower data set, which can be found in the `Rdatasets` package (don't forget to first add this package). We also need the `StatPlots` package when visualizing `DataFrames`. It contains an `@df` macro, which makes this much easier. Here is the code to draw a scatter plot:

```
# code in Chapter 10\plots_iris.jl
  using PyPlots, StatPlots, RDatasets
  iris = dataset("datasets", "iris")
  @df iris scatter(:SepalLength, :SepalWidth, group=:Species,m=(0.5,
  [:+ :h :star7], 4), bg=RGB(1.0,1.0,1.0))
```

We plot the `sepalwidth` property against the `sepallength` of the flowers. In the preceding code, `iris` is the name of our `DataFrame`, which is passed as the first argument to the `@df` macro. We then call the scatter function to obtain the following plot:



4.0
3.5
3.0
2.5
2.0
5
6
7
8
setosa
versicolor
virginica

Summary

In this chapter, we looked at the built-in functionality Julia has to offer in its standard library. We also took a peek at some of the more useful packages to apply in the data sciences.

We hope that this whirlwind overview of Julia has shown you why Julia is a rising star in the world of scientific computing and (big) data applications and this is what, you will take it up in your projects.

Other Books You May Enjoy