



2ND EDITION

# Mastering Julia

Enhance your analytical and programming skills  
for data modeling and processing with Julia



MALCOLM SHERRINGTON

# **Mastering Julia**

Enhance your analytical and programming skills for data modeling and processing with Julia

**Malcolm Sherrington**

**<packt>**

# Mastering Julia

Copyright © 2024 Packt Publishing

*All rights reserved.* No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

**Group Product Manager:** Kunal Sawant

**Senior Editor:** Kinnari Chohan

**Technical Editor:** Rajdeep Chakraborty

**Copy Editor:** Safis Editing

**Project Coordinator:** Manisha Singh

**Indexer:** Rekha Nair

**Production Designer:** Joshua Misquitta

**Marketing DevRel Coordinator:** Sonia Chauhan

First published: July 2015

Second edition: January 2024

Production reference: 2180124

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul's Square

Birmingham

B3 1RB, UK

ISBN 978-1-80512-979-0

[www.packtpub.com](http://www.packtpub.com)

*I would like to dedicate this book to the memory of my late wife, Hazel Sherrington, without whose encouragement and support, my involvement in Julia would not have started but who is no longer here to see the culmination of her vision.*

# Contributors

## About the author

**Malcolm Sherrington** has been working in computing for over 35 years. He holds degrees in mathematics, chemistry, and engineering and has given lectures at two different universities in the UK as well as worked in the aerospace and healthcare industries. Currently, he is running his own company in the finance sector, 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. In addition, he is a co-organizer of the UK High Performance Computing and the financial engineers and Quant London meetup groups.

## About the reviewer

**Mattia Nicolò Careddu** is a software engineer from Milan, Italy. He graduated in Statistics and Big Data. He is Senior Software Engineer at Conio inc, previously Lead AI Engineer at nCore HR. He is passionate about Technology, Bitcoin, Artificial Intelligence and Startups. He is an open source contributor and creator of Wasabi.jl, an ORM for Julia.



# Table of Contents

## Preface

xiii

1

## The Julia Environment

1

Julia 101	2	Julia has genuine runtime macros	11
Overview of Julia	2	<b>Getting started with Julia</b>	11
Philosophy	3	A first Julia script	12
Not only, but also...	4	Editors and IDEs	16
What is data science?	5	A quick look at some (more) Julia	22
Comparison with other languages	5	<b>Package management</b>	29
Why is Julia fast?	7	Listing, adding, and removing packages	30
Why use Julia?	9	Testing a package	32
Julia is easy to learn	9	Choosing and exploring packages	33
Julia is written (mostly) in Julia	9	Machine learning	36
Julia can interface with other languages	10	<b>Final thoughts</b>	36
Julia has a novel type system	10	<b>Summary</b>	36

2

## Developing in Julia

39

Technical requirements	40	Big integers	45
Integers, bits, bytes, and Booleans	40	<b>Arrays</b>	46
Integers	40	Broadcasting and list comprehensions	47
Primitive types	42	<b>Computing recursive functions</b>	49
Logical and arithmetic operators	44	Implicit and explicit typing	51
Booleans	44		

<b>Simple matrix operations</b>	<b>53</b>	<b>Multi-dimensional arrays</b>	<b>70</b>
<b>Characters and strings</b>	<b>55</b>	Sparse matrices	71
Characters	56	Sparse vectors	72
Strings	56	Sparse diagonal matrices	73
Byte array literals	59		
<b>Complex and rational numbers</b>	<b>60</b>	<b>Data arrays and data frames</b>	<b>73</b>
Complex numbers	60	Dictionaries, sets, stacks, and queues	74
Rationals	62	Dictionaries	74
<b>A little light relief</b>	<b>63</b>	Sets	77
The Sieve of Eratosthenes	63	Stacks and queues	78
Bulls and cows	64		
Julia sets	67	<b>Summary</b>	<b>79</b>

## 3

<b>The Julia Type System</b>		<b>81</b>
------------------------------	--	-----------

<b>More about functions</b>	<b>82</b>	<b>Derived and composite types</b>	<b>100</b>
The do syntax	82	A look at the Rational type	100
First-class objects	83	A composite Vehicle data type	104
Closures and currying	87	Modularization	109
<b>Passing arguments</b>	<b>89</b>	<b>typealias and unions</b>	<b>110</b>
Default and optional arguments	89	Enumerations	112
Variable argument list	92		
Keyword arguments	93	<b>Multidimensional vectors and computing pi (revisited)</b>	<b>113</b>
<b>Scope</b>	<b>94</b>	Parameterization	115
<b>The Queens problem</b>	<b>97</b>	<b>Higher dimensional vectors</b>	<b>116</b>
Conversion between numbers and strings	98	<b>Summary</b>	<b>118</b>

## 4

<b>The Three Ms</b>		<b>119</b>
---------------------	--	------------

<b>Multiple dispatch</b>	<b>119</b>	<b>Metaprogramming</b>	<b>129</b>
Code generation	122	Symbols and expressions	129
		Manipulating the code tree	132

---

<b>Macros</b>	<b>134</b>	<b>Functions or macros?</b>	<b>152</b>
Timing macros	136	<b>Modularity</b>	<b>152</b>
Macro hygiene	138	Loading a module	154
Macro expansions	139	Modular integers	155
MacroTools	143	Methods	157
Macro reductions	145	Testing	159
Lazy evaluation	147	Ordered pairs	160
<b>Generated functions</b>	<b>149</b>	<b>Summary</b>	<b>165</b>

## 5

---

<b>Interoperability</b>	<b>167</b>
-------------------------	------------

<b>C and Fortran</b>	<b>168</b>	Text processing and pipes	194
Mapping C types	170	Finding large files	195
Calling Fortran routines	170	<b>Perl one-liners</b>	<b>197</b>
Basel and Horner functions in C	171	A couple of examples	197
<b>C++</b>	<b>174</b>	<b>Using process I/O channels</b>	<b>200</b>
<b>Python, R, and Java</b>	<b>176</b>	<b>Interfacing with other languages</b>	<b>201</b>
Python	177	Perl 6	201
Going the other way	179	Ruby	202
Packages with Python wrappings	180	Python	202
The R (language)	183	Other languages supported by the JuliaInterop group	203
Java	189	<b>Working with the filesystem</b>	<b>204</b>
<b>Working with the OS and pipelines</b>	<b>191</b>	<b>Summary</b>	<b>205</b>
Running commands	191		

## 6

---

<b>Working with Data</b>	<b>207</b>
--------------------------	------------

<b>Basic I/O</b>	<b>207</b>	Text processing	214
Terminal I/O	208	<b>Binary files</b>	<b>217</b>
Terminal output	208	<b>Structured datasets</b>	<b>220</b>
Terminal input	209	CSV and other delimited (DLM) files	220
<b>Text files</b>	<b>212</b>	HDF5 and JLD files	226

Julia data format (JLD)	227	<b>Some simple statistics</b>	<b>244</b>
XML files	228	Kernel densities	246
<b>Time series</b>	<b>232</b>	Testing hypothesis	249
<b>DataFrames and statistics</b>	<b>236</b>	<b>Summary</b>	<b>251</b>
DataFrames	237		

## 7

---

<b>Scientific Programming</b>	<b>253</b>
-------------------------------	------------

---

<b>Linear algebra</b>	<b>254</b>	A touch of chaos	275
Matrix decomposition	256	Stochastic DEs	277
Simultaneous equations	256	<b>Calculus</b>	<b>278</b>
Eigenvalues and eigenvectors	258	Differentiation	278
High-order algebraic equations	260	Automatic differentiation	280
<b>Signal processing</b>	<b>261</b>	Quadratures	284
Frequency analysis	261	<b>Optimization</b>	<b>284</b>
Image convolutions	266	JuMP	285
<b>DEs</b>	<b>268</b>	<b>Stochastic simulations</b>	<b>290</b>
ODEs	268	SimJulia	290
Simulating a (real) pendulum	271	<b>Summary</b>	<b>295</b>
Catastrophic equations	273		

## 8

---

<b>Visualization</b>	<b>297</b>
----------------------	------------

---

<b>Textual visualization</b>	<b>298</b>	<b>The Plots API</b>	<b>319</b>
Simple inline displays	298	Creating multiple plots using layouts	321
Luxor	300	Recipes	322
Turtle graphics	302	Backends	324
PGFPlots	303		
<b>Basic graphic packages</b>	<b>305</b>	<b>Visualization frameworks</b>	<b>328</b>
PyPlot and PythonPlot	306	Plotly/PlotlyJS	328
Winston	309	StatsPlots	331
Gadfly	312	Makie	335

---

Basic image processing	339	<b>Summary</b>	<b>345</b>
The Images(.jl) family	342		

**9**


---

<b>Database Access</b>	<b>347</b>
------------------------	------------

<b>Database preliminaries</b>	<b>347</b>	Document databases	373
Interfacing to databases	348	<b>Interfacing with REST</b>	<b>377</b>
<b>Relational databases</b>	<b>349</b>	JSON/BSON formats	377
Building and loading	349	Web databases	379
Interfacing with a database	353	<b>(The) Queryverse</b>	<b>385</b>
SQLite	353	Querying the stocks dataset	386
MySQL	355	LINQ queries	388
PostgreSQL	362	Vega-Lite	389
JDBC databases	363		
<b>NoSQL databases</b>	<b>366</b>	<b>Summary</b>	<b>391</b>
KV datastores	366		

**10**


---

<b>Networks and Multitasking</b>	<b>393</b>
----------------------------------	------------

<b>Sockets and servers</b>	<b>393</b>	Web crawlers	413
Well-known ports	394	Genie	415
UDP and TCP sockets	395	<b>Tasks and remote procedures</b>	<b>419</b>
A “looking-glass world” echo server	396	Tasks	419
<b>Working with the web</b>	<b>401</b>	Remote procedures	421
HTTP methods	402	Needles and PI(ns)	422
Utility functions	404	Distributed arrays and Map-Reduce	424
TCP servers	407	Running on multiple machines	428
Routing	409	<b>Distributed data sources</b>	<b>428</b>
Mux	411	<b>Summary</b>	<b>432</b>

# 11

<b>Julia's Back Pages</b>		<b>433</b>	
<hr/>			
<b>Configuring Julia and the OS</b>	<b>434</b>	Performance tips	<b>453</b>
Getting Julia sources	434	Debugging	454
The <code>.julia</code> subdirectory	435	Revision	458
Julia environments	436	Profiling	462
Startup configuration(s)	438	<b>Creating packages</b>	<b>465</b>
<b>Standalone Julia</b>	<b>440</b>	A “funky” module	466
Scripting	440	Creating the layout	469
System images	446	Collaborating with Git	471
<b>Development tools</b>	<b>448</b>	<b>Quo Vadis, Julia?</b>	<b>471</b>
Document strings	449	<b>Summary</b>	<b>472</b>
<hr/>			
<b>Index</b>		<b>473</b>	
<hr/>			
<b>Other Books You May Enjoy</b>		<b>484</b>	

# Preface

The previous incarnation of this book was written when Julia was at v0.2, which scurried up to v0.4 by the time it went to press. Now I have been persuaded that a second version is overdue, and similarly, it was at v1.8.2 when I began and is now v1.9.4 but with a v1.10.x out as a release candidate and even v1.11 in development.

Why so long between editions? Julia in the past has not been reluctant to modify the language and seemingly continues to do so at the present time. There are many pitfalls in ignoring backward compatibility when developing a computing language. An example that probably many readers will be aware of is when Python tried to slither up from version 2 to version 3, a sizeable minority of the users had to be dragged screaming to make that leap and it was only when support for version 2 was effectively withdrawn that this happened.

More alarming was when Perl, which once held the position that Python holds now (circa. 2023) moving from v5 to v6 proved so impossible that Perl 6 was renamed (a couple of times) as a new language, one which is rarely used, as is also Perl 5.

So given the gap between the two versions of the book, it is to be hoped that Julia, now well past v1.0, which was announced at JuliaCon 2018 when it was hosted here in London, might at least have now settled down and done without the dreaded depreciation warnings which eventually turn into errors. Moreover, since many Julia packages are written in 100% native Julia code, the effects of changes made in Julia are widespread and packages need to be promptly maintained or the packages should be retired.

Turning to the content of the book, its philosophy remains the same as earlier. So, it will not start by discussing how to print “Hello World” nor how to compute the result of  $1 + 1$ , although there is a version of the former, a metaphorical wolf in sheep’s clothing, so it can be found!

Again, I’m not discussing the usual programming constructs for looping and branching, but rather highlighting some Julia constructs that may not be familiar to all, such as list comprehensions, broadcasting, etc., and of course, dealing with topics such as the Julia type system. The examples are not overly sophisticated but hopefully detailed enough that they will be interesting and importantly runnable for some time in the future.

The chapter count now has risen from 10 to 11, this is due to the some of the material in the *Dispatch and types* chapter being split out into separate chapters, the second one being enigmatically entitled: 3M’s, viz., referring to multiple dispatch, macros and modules, all of which will have been met already in some to the earlier material in the book. This means that now the first five chapters are mainly aimed at Julia’s programming and the next five at themed topics, all promoted by an extra number, are akin to those previously and with the material in these being brought up to date. The final chapter is somewhat different, another with an enigmatic name?!

## What this book covers

*Chapter 1: The Julia Environment* is intended as a gentle introduction to Julia which covers the steps needed to get a working distribution up and running and the steps needed to acquire packages and run Julia in the REPL, code editors (in particular) VS-Code and the Jupyter and Pluto IDEs, continuing then by illustrating these with some simple examples.

*Chapter 2: Developing in Julia* is an overview of some of Julia's basic syntax, discussing how to work with simple numeric and character variables either as scalars or in array aggregates, along with briefly introducing the concept of data frames, which will be met often in the remainder of the book.

*Chapter 3: The Julia Type System* describes first the use of further features such as the specification of parametric and optional parameters as arguments to functions and then continues by introducing the idea of abstract and concrete data types as a precursor to working with and defining more complex composite data structures. It concludes by considering unions and aliases in Julia and higher dimensional arrays.

*Chapter 4: The Three M's* covers the topics of three aspects of Julia which may be a little more unfamiliar to users of a different programming language, namely the concept (*and use of*) multiple dispatch, the macro system, and Julia's approach to modularity, which is somewhat different to conventional object-oriented systems.

*Chapter 5: Interoperability* is a little different as it is concerned with Julia working together with other languages either directly with C, Python, and R (*via the shared libraries or APIs*) or indirectly by invoking the aid of the operating system, setting up pipelines and redirecting basic input and output.

*Chapter 6: Working with Data* is the first of the themed chapters, concerned primarily with handling textual and binary data files, and structured datasets, and meeting again in much more detail Julia's sophisticated structures, i.e., data frames and tables. The chapter concludes by applying some simple statistical analysis to some of the datasets referenced.

*Chapter 7: Scientific Programming* may have originally been viewed as the jewel in Julia's crown, the raison d'être of its purpose. The development of the language encompassed many other aspects but recently its focus on machine learning and system simulations may be heralding a move back in this direction. It covers a wide variety of topics including linear algebra, signal processing, solution of differential equations, optimization, and stochastics, albeit briefly.

*Chapter 8: Visualization* provides a little light relief turning to the production of graphics and other forms of imagery. Originally Julia was criticized, unfairly, as being deficient in this area but now has an embarrassment of riches. The chapter covers simple (raster) video displays and packages for creating vector graphics & hardcopy output such as PDFs and then now to the comprehensive Plots API. It also touches on the powerful image processing packages and concludes with an overview of some Julia mega graphic frameworks.

*Chapter 9: Database Access*, as might be inferred from the name, deals with Julia's methods for interaction with databases. The division between relational (SQL-style) databases and others, often

lumped together and NoSQL is described and various examples of each are explored, both disk-based and in-memory types. It continues by describing the use of REST-style web database APIs and finishes with a discussion of the use of the Queryverse and its interaction with VegaLite graphics.

*Chapter 10: Networks and Multi-tasking* focuses on the methods in Julia for multitasking and working with distributed systems. It introduces the concept of sockets and creates networked services that can be used when working with the web. Then the methods by which parallel processes can be defined and run in Julia are presented, finally, it concludes with a discussion of analyzing distributed datasets.

*Chapter 11: Julia's Back Pages* are a little different dealing with topics that should be understood to progress from causal user to serious developer. It covers running scripts without the REPL (or an equivalent IDE) and then onto the major topic of creating Julia packages, by means of techniques such as code profiling and use of the debugger. Finally poses the question of where next for Julia, however, does not attempt to answer it!

## Who this book is for

The book is targeted at programmers with some familiarity with Julia or else fluency in another programming language, such as C/C++, Python, or R

It assumes some familiarity with the use of code editors, a popular one currently being Visual Studio Code, and also web-based IDEs such as Jupyter.

It covers a wide range of topics in some chapters where knowledge of working with the underlying operating system(s), developing in other programming languages, and installing and running databases will be beneficial.

## To get the most out of this book

The software and OS requirements to get the most out of this book are:

Software/hardware covered in the book	Operating system requirements
Julia 1.X	Windows, macOS, or Linux

## Downloading the code accompanying the book.

Packt Publishing distributes the sources on their GitHub website, and for this book, the sources can be found at <https://github.com/PacktPublishing/Mastering-Julia-Second-Edition>

These can be viewed using a web browser and can be downloaded using the `git clone <git-url>` command which will create a copy of the sources named `Mastering-Julia-Second Edition`.

As this is somewhat unwieldy, I have renamed this as MJ2 on my computer(s), and this is used in the text throughout the book and in the Julia scripts.

To clone the sources may require using the built-in git command, although it is possible to use an editor such as Visual Studio Code or a GIT IDE. Whether `git` is available from the command line depends on which operating system is being used.

- macOS doesn't have a built-in version of `Gi`; however, it's included as part of Xcode or else installed via using homebrew or by choosing an IDE such as GitHub Desktop.
- Most Linux distros have the `git` command already built in.
- For Windows binaries can be downloaded from the main `git` website using <https://git-scm.com/download>; in fact, this page includes sources for all three operating systems plus links to several popular `git` clients.

The source accompanying this book is arranged as a set of code folders, one per chapter denoted from `Code01` through to `Code11`, containing all the Julia source code relating to the chapter. For example, the `Code01` folder contains the relevant script named as: `Chp02.jl`. There may be additional sources and in particular those corresponding to Jupyter and Pluto notebooks. The sources in each folder are described in the markdown file(s) `README.md` or the equivalent `README.txt` file.

Additionally, there is a `Project.TOML` and a `Manifest.TOML` file for each chapter plus a script (`setup.jl`) which can be used to create these from scratch rather than using the ones provided.

These can be utilized by switching to the specific project via the package manager via

```
julia> import Pkg; Pkg.activate("..")
```

Possibly when first cloned it may be sensible to also include: `Pkg.instantiate()`

It will be found to be necessary to work with individual chapter projects rather than adding all Julia modules to the general registry, which otherwise will get extremely clogged, with probable conflicts arising. However, in addition to building up manifests, the reader may find `setup.jl` useful in getting started with some of the earlier code samples in the book, rather than just adding them individually in the REPL

Additionally, there are two other directories: Alice and DataSources

- Alice has a set of files, from the Alice books and others of Lewis Carroll's works, which are used in some text processing routines.
- DataSources is split into subfolders containing datasets, such as those in CSV and TSV formats, and files necessary for the sections working with SQL-based and other databases.

As an example `ENV["HOME"] * "/MJ2/DataSources"` is used as a reference to the location of the data sources directory, since `ENV["HOME"]` is an alias to my user home directory and I have added the GitHub sources as a folder directly under it.

Note that on Windows this alias is now `ENV["HOMEPATH"]`. So the construct `joinpath(ENV["HOMEPATH"], "MJ2", "DataSources")` may be preferred as it does not depend on the directory separator, though in my experience either “`\`” or “`/`” can be used in Julia when on Windows.

## Share your thoughts

Once you've read *Mastering Julia*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below



<https://packt.link/free-ebook/9781805129790>

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

# 1

## The Julia Environment

In this chapter, we will explore all you need to get started on Julia. Julia is a high-level, high-performance, dynamic programming language that focuses mainly on technical computing. Its primary feature is that, although it's a scripting language, the code is compiled using **low-level virtual machine (LLVM)**. This compilation creates code that corresponds to the underlying hardware/operating system and runs at similar speeds to other compiled languages, such as C/C++ and Fortran.

The chapter will cover the following main topics:

- Acquiring and installing Julia
- The philosophy behind Julia and comparison with other programming languages
- Developing code using the REPL, code editors, and IDEs
- Use of the package management system
- Some simple coding in Julia
- A quick look at some graphics

By the end of this chapter, you should have a grasp of the basics of Julia. All the topics introduced are covered in more detail in the later chapters of this book, so a detailed understanding is not necessary, although it should be possible to set up Julia on your chosen hardware platform and run the examples from the code section by either using the **read-evaluate-print loop (REPL)** or installing one (or more) of the editors or **interactive development environments (IDEs)**.

## Julia 101

Julia can be downloaded from the main website (<https://julialang.org>) via the **Download** menu option. It is available for Windows, macOS, Linux, and FreeBSD, and the appropriate download bundles are provided for a current stable version, a **long-term support (LTS)** version, and the next release. At the time of writing, the stable release is 1.8.5, the LTS is 1.6.7, and the new beta release is 1.9.0. I will briefly cover installing and uninstalling later, although the Julia website is the best source of information.

It can be run using the Julia REPL, the VS Code editor, and a couple of web-based IDEs, Jupyter and Pluto, which we will discuss later in this chapter.

The previous edition used Julia v0.4, which was the stable version at the time, so a large amount has changed since then. When we hosted JuliaCon 2018, v1.0 was announced, promising fewer breaking changes, although these are not insignificant, and so far, a number of breaking changes have been introduced on each release of Julia. For those interested, a list of all release notes is available on GitHub as a Markdown file (<https://github.com/JuliaLang/julia/blob/master/HISTORY.md>).

The other main features since the previous edition are the rapid growth in Julia packages, all of which are available on GitHub, including a new focus on machine learning and AI. The latter chapters in this book will target individual topics such as data handling, statistical analysis, graphics, and database access, choosing the appropriate current primary packages, but you are encouraged to survey new ones that have been or are being developed.

There is a wealth of material on YouTube, including past JuliaCon presentations (*the workshops are particularly useful*), the work of the Dabbling Doggo (<https://www.youtube.com/@doggodotjl>), and the admirable MIT course, *Computational Thinking* (<https://computationalthinking.mit.edu/Fall122>).

All the code for individual chapters is available as code scripts, which can be run in the Julia REPL or Visual Studio Code, and in many cases, Jupyter and/or Pluto notebooks are also provided.

The code in this book, and that on GitHub, is normally covered by an MIT open source license and so is free to be used (*or abused*) by you.

## Overview of Julia

Julia was first released to the world in February 2012 after a couple of years of development at the **Massachusetts Institute of Technology (MIT)**. This was followed by a couple of years of development at MIT. Later, in 2015, a commercial arm called Julia Computing was set up to acquire funding and provide consultancy and (some) enterprise packages; this was later renamed to JuliaHub to reflect the inclusion of a cloud computing platform facility, at present, hosted on **Amazon Web Services (AWS)**. The use of the JuliaHub cloud is discussed in the last chapter of this book.

Most of Julia remains freely available and we will be concentrating on that here. As mentioned previously, version 1.0 was released in 2018 and, at present, is approaching v1.9.0, although there seem to be no plans for a **main** release to mark crossing the v2 barrier.

All the original developers – Jeff Bezanson, Stefan Karpinski, and Viral Shah – still maintain roles in the evolution of the language and with JuliaHub but have been joined by some of the major contributors over the last five years. So, uniquely, all the principal authors are still actively employed in Julia's progress.

The language is open source, so is available to preview, and a version is available as a download from <https://julialang.org/downloads> or <https://github.com/JuliaLang/julia>. The latter has a discussion on installing from binaries, which, in the main, is just a matter of getting the appropriate source for a specific operating system and running through the usual application installation procedure.

Using binary sources is the preferred (and simplest) method to install Julia, but the method by which Julia can be built from source is also discussed.

One remarkable fact is that a single set of source files can be used to build on both MS Windows and POSIX systems such as macOS, Linux, and BSD. There may be an instance where a package struggles to execute on Windows, but the developers have worked hard to ensure that this is not the case for the core language.

There is a small amount of C/C++ code plus some Lisp and Scheme, but much of the core is (*very well*) written in Julia itself and may be perused at your leisure. Unzip the source code and look at the `base` and `stdlib` directories to find the bulk of the Julia code. If you wish to write exemplary (although often quite complex) Julia, this is a good place to go in order to seek inspiration.

Julia is often compared with programming languages such as Python, R, and MATLAB, which tend to occupy the same *computational* space. It is important to realize that Python and R have been around since the mid-1990s and MATLAB since 1984, so these are more stable, settled languages in terms of their syntactic sugar. Since MATLAB is proprietary (\**MathWorks*), there are a few clones – in particular, GNU Octave, which again dates from the same era as Python and R.

As registered packages are stored in GitHub, it is useful to have the command version of Git, or a GUI such as GitKraken, installed on your computer, although normal interaction is largely hidden from the user since Julia incorporates a working version of Git, wrapped up in a package manager (Pkg), which can be called from the console. Both the package manager and the use of Git will be discussed later in the book.

## Philosophy

Julia was designed with scientific computing in mind. The developers tell us that they came with a wide array of programming skills – Lisp, Python, Ruby, R, and MATLAB.

All needed a “fast” compiled language in the armory such, as C or Fortran as the current languages listed previously are pitifully slow. Here is a quote from the development team:

*We want a language that’s open source, with a liberal license. We want the speed of C with the dynamism of Ruby. We want a language that’s homoiconic, with true macros like Lisp, but with obvious, familiar mathematical notation like MATLAB.*

*We want something as usable for general programming as Python, as easy for statistics as R, as natural for string processing as Perl, as powerful for linear algebra as MATLAB, as good at gluing programs together as the shell. Something that is dirt simple to learn yet keeps the most serious hackers happy. We want it to be interactive and we want it compiled.*

With the introduction of the LLVM compilation, it became possible to achieve this goal and to design a language from the outset that makes the “two-language” approach largely redundant.

Julia was designed as a language like the other scripting languages and so should be easy to learn for anyone familiar with Python, R, and MATLAB. Julia code looks very similar to MATLAB; however, it is not a MATLAB clone – that is, MATLAB code will not run in Julia nor Julia code in MATLAB. Also, there are many important differences between the syntax of the two languages, as we will see when progressing through this book.

Also, we should not be overly fixated on considering Julia as a challenger to Python and R; in the last few years, Julia has emphasized its advantages in the high-speed technical computational sphere, particularly with respect to machine learning, Bayesian analysis possibly in conjunction to often seamless deployment onto the GPU, all of which can be done within Julia without any reference to creating or utilizing any low-level C/C++ (or similar) code.

In fact, as will be illustrated, there are instances where the languages are used to complement each other. Since Julia was not conceived as a straightforward competitor, there are certain things that Julia does that make it ideal for use in the scientific community.

## Not only, but also...

Julia was initially designed with scientific computing in mind and has made considerable strides in recent years in the fields of machine learning, optimization, solution of differential systems, and so on, all of which will be dealt with topics in the latter portion of this book.

However, Julia was originally seen as a vehicle for the emerging discipline of data science, possibly as an alternative or adjunct to more popular approaches, and it is worth remarking that practitioners in data science would be advised to see what Julia has to offer here as well.

Although the term *data science* was coined as early as the 1970s, it was only given prominence in 2001 by William S. Cleveland in his article, *Data Science: An Action Plan for Expanding the Technical Areas of the Field of Statistics*.

Almost in parallel with the development of Julia has been the growth in data science and the demand for data science practitioners.

## What is data science?

People often say that there are many possible definitions of data science, just as there are many people who call themselves data scientists. However, data science is not technically a science in the same way that physics, chemistry, and biology are sciences, as it does not typically involve the application of the scientific method. A definition might be as follows:

*Data science is the study of the generalizable extraction of knowledge from data. It incorporates varying elements and builds on techniques and theories from many fields, including signal processing, mathematics, probability models, machine learning, statistical learning, computer programming, data engineering, pattern recognition and learning, visualization, uncertainty modeling, data warehousing, and high-performance computing with the goal of extracting meaning from data and creating data products.*

If this sounds familiar, then it should be. These were the precise goals laid out at the onset of the design of Julia. To fill the void, most data scientists have turned to Python and, to a lesser extent, to R. One principal cause of the growth in the popularity of Python and R can be traced directly to the interest in data science.

So, what do we set out to achieve in this book?

To show you, as a budding data scientist, why you should consider using Julia and, if convinced, then how to do it.

Along with data science, the other “new kids on the block” are big data and cloud computing.

Big data was originally the realm of Java, largely because of the uptake of the Hadoop/HDFS framework, which is written in Java, making it convenient to program map-reduce algorithms in it or any language that runs on the JVM.

This leads to an obscene amount of bloated boilerplate coding. However, with the introduction of Yarn and Hadoop stream processing, the paradigm of processing big data is opened up to a wider variety of approaches.

Python, in the beginning, was considered an alternative to Java, but on inspection, Julia made an excellent candidate in this category too.

## Comparison with other languages

The most *well-known* feature of Julia is that it creates code that executes very quickly.

As we continue to look at the language, we will discover why this is and also see many other features incorporated into Julia that impart much more benefit to the programmer and the data analyst alike; however, it is nice to be fast too!

The home page of the website of the main Julia website, as of July 2014, includes references to benchmarks:

	Fortran	Julia	Python	R	Matlab	Octave	Mathematica	Javascript	Go
fib	0.26	0.91	30.37	411.31	992.06	3211.816	4.46	2.18	
mandel	0.86	0.85	14.19	106.97	4.58	316.95	6.07	3.49	
pi_sum	0.80	1.00	16.33	15.42	1.29	237.41	1.32	0.84	
mat_stat	0.64	1.66	13.52	10.84	6.61	14.98	4.52	3.28	
mat_mul	0.96	1.01	3.41	3.98	1.10	3.41	1.16	14.60	

In the preceding table, all the times are scaled by dividing by the corresponding time for the benchmark coded in C. The lower the time, the better, and in some cases, the performance of Fortran and Julia is better than C, probably due to effective code optimization.

The Julia site does its best to lay down the parameters for these tests by providing details of the workstation used – processor type, CPU clock speed, amount of RAM, and so on – and the operating system deployed. For each test, the version of the software is provided plus any external packages or libraries – for example, for the `rand_mat` test, Python is using NumPy, and C, Fortran, and Julia are using OpenBLAS.

**Note**

Julia provides a set of web pages specifically for checking on its performance: Julia Micro-Benchmarks (<https://julialang.org/benchmarks>).

This table is useful in another respect too, as it lists all the major comparative languages to Julia; no real surprises here, except perhaps the actual ranges of execution times. It may be noted that both MATLAB and Python have been/are dabbling with LLVM computation but as this is not an intrinsic feature of either language, the places where it can be used and the method of deployment is cumbersome:

- **Python:** This has become the de facto data science language and the range of modules available is overwhelming. In general, Julia is at least an order of magnitude faster than Python, which is why enterprise Python code needs to be rewritten and compiled in C/C++ or Java.
- **R:** This started life as an open source version of the commercial S+ statistics package (^ Tibco Software Inc.) but has largely superseded it for use in statistics projects and has a large set of contributed packages. It is single-threaded, which accounts for the disappointing execution times, and parallelization is not straightforward. R has very good graphics and data visualization packages.

- **MATLAB/Octave:** MATLAB is a commercial product (\*MathWorks) for matrix operations, hence the reasonable times for the last two benchmarks, but others are very long. GNU-Octave is a free MATLAB clone. It has been designed for compatibility rather than efficiency, which accounts for the execution times being even longer.
- **Mathematica:** This is another commercial product (\*Wolfram Research) for general-purpose mathematical problems. No obvious clone, although the Sage framework is open source and uses Python as its computation engine, so its timings are similar to Python.
- **Javascript** is interpreted using a JIT engine, initially found on client-side web applications, but latterly becoming popular under the guise of Node.js.
- **Go** is compiled and is suited for building high-performance, concurrent systems and projects that require efficient memory management.

Julia would seem to be an ideal language for tackling data science problems.

It is important to recognize that many of the built-in functions in R and Python are not implemented natively but are written in C.

Julia produces code that executes *roughly* as that written in C. One consequence is that Julia won't markedly outperform languages such as R or Python if most of the work done (in R or Python) consists basically of calling built-in functions. With native code, such as that involving any explicit iteration or recursion, Julia comes into its own.

It is the perfect language for users of R or Python who are trying to build advanced tools inside of those languages. The alternative to Julia is typically resorting to C. Although R provides this through Rcpp and Python through Cython, both approaches involve moving outside the native language syntax and, in my experience, are seldom implemented.

There is possibly more cooperation between Julia with R and/or Python than the competition, although this is not yet the common view.

## Why is Julia fast?

Julia's "big" idea was to compile the program right down to the machine code level.

This was by incorporating the LLVM technology developed at Urbana-Champaign in the early 2000s. LLVM was originally termed as the low-level virtual machine but is now seen as a mnemonic. Conceptually, Julia Core is parsed via an internal Lisp (Femtolisp) translator and then compiled into an **intermediate representation (IR)**, and then a machine-dependent LLVM compiler is invoked to produce the actual executable.

Although this represents an overhead, the code is cached (i.e., only compiled once), and much effort has gone into creating system images of the basic Julia system and caching individual packages. This makes execution times of the same "order" as C code, perhaps about (x2). C compilers are often better

optimized, but LLVM is getting there quickly. So, Julia provides the holy grail of compact code as well as fast execution times. The downside is that when a package is installed, a lengthy process of pre-compilation is performed so that the compiled code can be cached.

Julia provides ways to look at the code at the various stages from parsing to final machine code, and we will discuss these later.

*Figure 1.1* shows a comparison of the conciseness of code (viz., bytes/file) against the execution speed for the various tests. Note that the closer to the origin, the better, providing compact code that executes quickly.

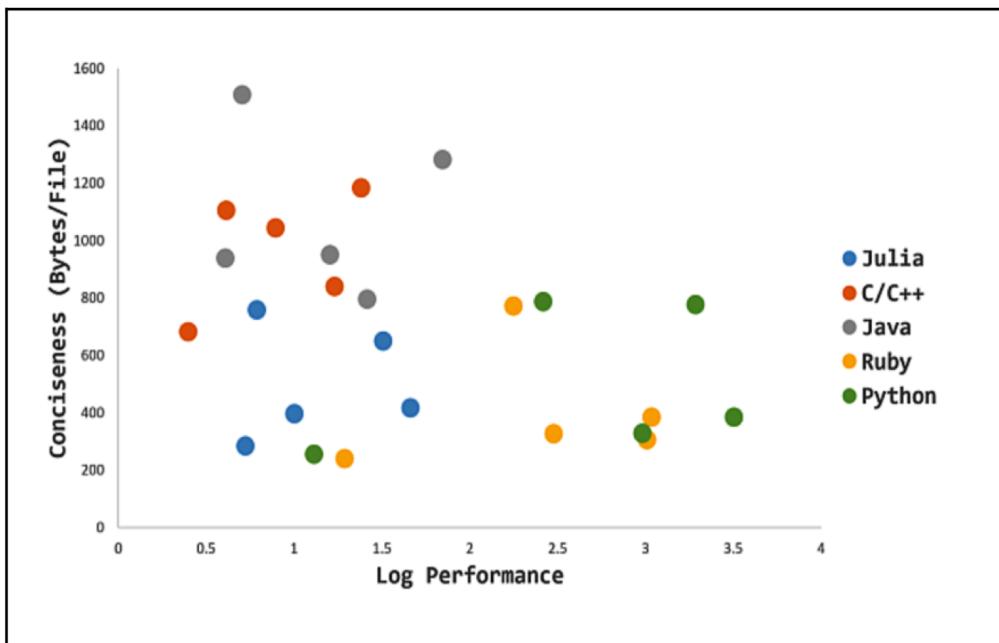


Figure 1.1 – The performance times are all on a logarithmic scale

One of the great features of Julia is that, as a scripted language, it produces compact code that runs quickly, unlike Python, Ruby, or R, which are compact but slow, or C and Java, which are quick but verbose, as *Figure 1.1* shows.

It is possible to write inefficient code and make Julia run (relatively) slowly, although still quicker than Python, R, and so on. In the final chapter, I'll be discussing how to write efficient code and providing some examples of the converse; however, with Julia, this is still fast. In my opinion, the speed of Julia should not be considered the principal reason to learn a language.

Since Julia was only designed in 2010 and has been actively modified all the way up to version 1.0, the advances in computing software and hardware since the 1990s, when Python and R first were

---

developed, have been built into Julia from its design. Retrofitting these to existing language architectures has not always proved to be so easy.

## Why use Julia?

Programming in Julia sometimes seems too good to be true. As it has been implemented in the last few years, many of the recent ideas in computer science design have been incorporated into the language and the developers have not been afraid to modify Julia's structure and syntax on the run-up to version 1.0, even though this has led to deprecations and breaking changes.

We have pointed out previously that Julia creates executable code from scripts without a separate compilation step and this results in runtimes in the same order as those of C, Fortran, Java, and so on; however, in my opinion, that is not the main reason to use Julia. In this section, we will look at the other factors that make it a *must-see* for any programmer, analyst, and data scientist.

### Julia is easy to learn

Writing simple code in Julia will be almost immediate for anyone with a grounding in Python, R, C, and so on, as this book will show.

As mentioned previously, the syntax is based on MATLAB, where code blocks, `for/while` loops, `if` statements, and so on are ALL terminated by `end`.

There is no lining up of code (Python) or matching of `{ }` brackets (R) and no distinction between `if-endif`, `for-endfor`, and `end-endwhile`.

The code is very close to the pseudocode that you might write down to sketch out an algorithmic solution.

#### Remember

Julia is not to be seen as a MATLAB clone, as (say) Octave is.

MATLAB code will NOT run in Julia, nor is the reverse true.

However, porting from MATLAB to native Julia code is usually quite straightforward.

### Julia is written (mostly) in Julia

It is difficult to be precise, but based on lines of text (say), approximately 85% of the code is written in Julia. This includes numerical types such as integers, floats, and complex numbers, as well as strings and more sophisticated data structures.

This code is termed the *base* and can be inspected by the programmer as a reference to get inspiration. The same is true for the installed modules (packages), which also will contain test routines and, in many cases, more detailed examples.

## Julia can interface with other languages

The remaining 15% is termed the *core*. The core is principally written in C and compiled into a shared object library (on Linux and macOS) or a DLL (on Windows). The routines in the *base* interact with the *core* via a well-defined API, which is well-documented and examples of how the API is used can be seen by inspecting their sources.

Calling C routines that have been compiled into libraries, and by implication, Fortran routines, is straightforward and normally just a single function call in Julia; if it were not so, Julia would not function. This makes creating “wrapper” packages very easy (i.e., modules that basically interface with a separate set of routines in a separate library). Indeed, the BLAS and Lapack routines for linear algebra manipulation have been implemented in such a fashion from the early days of Julia (see the source in `linalg/lapack.jl` for details), and the power of the I/O system is derived in part from interfacing with the Joyent/nodejs library: `libuv`.

Additionally, Julia can interface with Python, Java, R, and more. Interfacing with Python is two-way and used in the Jupyter IDE, as well as graphics via PyPlot, which is a wrapper around Python’s matplotlib.

We will discuss interfacing in more detail in *Chapter 5*.

## Julia has a novel type system

Data structures (aka *objects*) are defined in packages in a hierarchical system, but only the lowest type is instantiated and has functions that operate on its data.

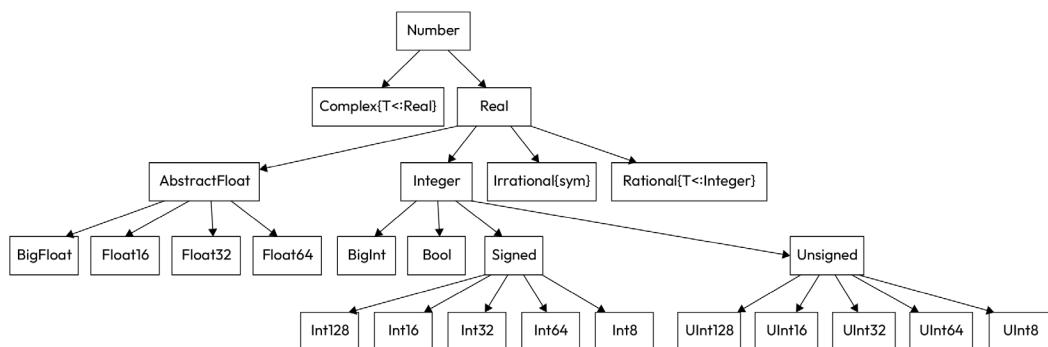


Figure 1.2 – A subset of the type system corresponding to the hierarchy of numbers in Julia

The higher nodes are known as **abstract** types, whereas the bottom ones are termed **concrete** types. There is no inheritance or polymorphism, which may seem like a failure of the traditional object orientation. As we will see in the next chapter, developing Julia’s type system leads to a great simplification in code through aggregation and speed via the very powerful mechanism of multiple dispatches.

In my opinion, *multiple dispatch* is one of the most important features of Julia and is much more significant than merely executing code quickly.

## Julia has genuine runtime macros

Macros are defined via functions that can generate block code in a simple single-line invocation. When a program is run, it evaluates the macro, and the code produced is eventually evaluated as an ordinary expression.

Macros can be distinguished as they are preceded by the @ symbol.

Julia also implements a new hybrid feature called a generated function, via a special macro called @generated. Generated functions have the capability to create specialized code depending on the types of their arguments with more flexibility and less code than what can be achieved with multiple dispatch.

Macros are not to everyone's taste and there will always be ways to code in a more conventional fashion; however, even if not for you, they will have been used by many package developers and you will make use of them extensively.

Indeed, certain common macros such as @time, @assert, @printf, and more will crop up widely throughout this book.

We will look more closely at these in *Chapter 4*.

## Getting started with Julia

Starting to program in Julia is very easy. Naturally, the first task you will need to do is to install Julia on your computer. Thankfully, this has been made very simple. In the early versions of Julia, it was necessary to build from source but was largely made redundant with binaries for the major operating systems.

We differentiate between several different sets of sources:

- Windows
- Apple macOS
- Generic Linux (x86)
- Generic Linux (ARM)
- Free BSD
- Source (necessary for other OSs)

As noted previously, the best place to look is the main Julia website (<http://julialang.org>), and navigate to the **Downloads** tab on the menu.

Windows and macOS are serviced by `exe` and `dmg` binaries, respectively. In these cases, installation is as simple as downloading the binary and clicking on it, and everything else is handled by the installer. As of v1.9, a source on Apple macOS is now available for the new M-series processors, in addition to the more common Intel x86-based systems.

Linux systems were previously distributed for Red Hat/CentOS (`pkg`) and Debian/Ubuntu (`deb`) packages, but now are compiled for generic Linux systems and provided as a zipped archive; however, the overnight development system still provides `pkg` files.

Various binaries for ARM are available.

Also, a source-only archive is available, which can be used to build the binary.

It is worth noting that Julia has comprehensive documentation, which can be found in the **Documentation** tab at <http://julialang.org>, as well as links to the package manager, community and learning resources, and so on.

Once Julia is installed, it is necessary to add some additional modules using the package manager. In this book, we will introduce packages as they are needed.

We will not be discussing build from source, as this is no longer needed to get up and running.

For those interested, the subsection of the Julia GitHub project specifically dealing with Julia itself gives comprehensive documentation via its markup page: <https://github.com/JuliaLang/julia>.

These web pages also deal with uninstalling Julia, which is as simple as deleting the source and the (hidden) directory in the user's home directory, `.julia` (note: dot subdirectories are NOT hidden in Windows).

If you are interested in low-level development in Julia, then this is the place to start.

## A first Julia script

We will be looking at examples of Julia code later in this chapter, but if you want to be a little more adventurous, if you have installed Julia, start the command-line version (REPL) and try typing in the following at the `julia>` prompt:

```
using Printf
sumsq(x,y) = x^2 + y^2;
N = 1000000;
x = 0;
for i = 1:N
    if sumsq(rand(), rand()) < 1.0
        global x += 1
    end
end
@printf "Estimate of PI for %d trials is %8.5f\n" N 4.0*(x / N)
```

This first script computes a simple estimate of  $\pi$  by generating pairs of random numbers distributed uniformly over a unit square [0.0:1.0, 0.0:1.0].

#### Note

In the code I have terminated some statements with a semicolon (;) as this inhibits output when run interactively in the REPL. This is purely a choice, for convenience, and does not alter the overall execution of the script.

If the sum of the squares of the pairs of numbers is less than 1.0, then the point defined by the two numbers lies within the unit circle. The ratio of the sum of all such points to the total number of pairs will be in the region of one-quarter PI.

- The line `sumsq(x, y) = x^2 + y^2` is an example of an inline function definition. Of course, multiline definitions are possible and more common but to be able to do one-liners is very convenient. It is possible to define anonymous functions too, as we will see later.
- Although Julia is strictly typed, a variable's type (when not explicitly defined) is inferred from the assignment, unless explicitly defined.
- Constructs such as `for` loops and `if` statements are terminated with `end`, and there are no curly brackets, { }, or matching `endfor` or `endif`.
- Printing to standard output can be done using the `println` call, which is a function and needs the brackets. `@printf` is an example of a macro that mimics the C-like `printf` function, allowing us to format outputted values
- As of v1.0, the `@printf` macros have been moved out of the `base` and into a separate package, so we need to include using `Printf` at least once in the code.
- In v1.0, there are new scoping rules that disbar top-level variables in the REPL from being visible inside loops, although they are visible in `begin/end` and `if/else/end` statements, hence the presence of the `global` statement inside the loop. These have been modified somewhat now that Julia has reached v1.8, and I will deal with these in the next section.

Note that if you are interested in how quickly this runs, it is possible to prefix the `for` loop with the `@time` macro:

```
@time for i = 1:N
    if sumsq(rand(), rand()) < 1.
        global x += 1;
    end
end
0.175244 seconds (3.78 M allocations: 73.008 MiB, 5.80% gc time)
```

This is possible after the `sumsq` function has been defined and the value of `N` set; also, `sumsq()` should be run at least once to exclude the compilation time from the overall timing.

### ***Scoping rules***

As we said previously, global variables in v1.0 are not visible inside `for/end` and `while/end` loops due to new scoping rules. If you were running v1.0, the error message is less than helpful:

```
k = 0;
for i = 1:10
    k += i
end
ERROR: UndefVarError: k not defined
Stacktrace:
 [1] top-level scope at ./REPL[3] :
```

As noted previously, this can be solved by flagging the `k` variable as `global`. This caused some consternation with the current author, among others, and resulted in some protracted discussions, the outcome being that the `k` variable on line 1 would be treated as `global` but that on line 3 as `local`, and a warning to that extent issued.

It is worth noting that this only applied to scalars (such as `k` previously), array values are visible inside the loop and to members of tuples, which we will meet later. Also, it does not apply inside function definitions.

As of v1.8, this now works in the REPL without the `global` prefix inside the loop. I have provided a script in the code accompanying this chapter called `pi.jl`, without the `global` statement, which can be cut and pasted into the REPL to demonstrate this. So has the behavior gone away? Sadly, this is not the case.

It is possible to run the script from the command line using syntax such as the following:

```
$> julia pi.jl
Warning: Assignment to `K` in soft scope is ambiguous because a global
variable by the same name exists: `K` will be treated as a new local.
Disambiguate by using `local K` to suppress this warning or `global K`
to assign to the existing global variable.
@ ~/MJ2/Chp01/Code/PIEs/pi.jl:10
ERROR: LoadError: UndefVarError: K not defined
```

Note that the `$>` dollar prompt refers to that from the operating system and will vary with individual users. [I will be discussing the use of scripts in more detail in the final chapter of the book.]

So, the warning is back in this situation.

Following the warning, a fatal error occurs because `k` is not defined.

It is also the case that the warning/error occurs if the code is referenced in the REPL using the `include` statement: that is, `include ("pi.jl")`.

Later in this chapter, we will be discussing various editors and IDEs but to anticipate the outcomes, the version without the `global` statement will work in the VS Code editor and IJulia (viz., Jupyter) but not, without a little TLC, in the Pluto IDE.

However, the `global` version works fine:

```
$> julia piG.jl  
Estimate of PI for 100000 trials is 3.14056
```

The function versions of PI estimation work well, but not without the odd quirk, as we will see when discussing functions in *Chapter 2*.

What we have done is create a simple piece of code that works in the REPL, among other places, but does not run from the command line, which, to my mind, seems to be worse than the prior v1.0 behavior.

The full explanation of the current Julia scoping rules can be found in the Julia documentation, Scope of Variables (<https://docs.julialang.org/en/v1/manual/variables-and-scoping/>), which defines three levels of scope – global, local (soft), and local (hard) – and elaborates in detail on the situations where each applies. I will leave it to you to make sense of it.

It is possible to pass parameters to an inner loop using arrays, structs, and **dictionaries (Dicts)**, all of which are data structures passed by reference; it is just the humble scalar that comes undone by the scoping rules.

A set of scripts can be found in the code section of Chp01, in the `PIEs` subfolder, which demonstrates the different methods to tackle the PI estimate code. Rather than discuss these in detail here, if interested, look at the `README.txt` file in the folder for detailed information.

### ***Adding modules with Pkg***

Comprehensive package management uses the Package Manager (alternate) view from the Julia REPL, and we will discuss this later in the chapter. But it is worth noting that there is an alternate built-in package, `Pkg`, which can be used for simpler tasks such as installing (*adding*) and deleting (*removing*) modules.

For example, the `BenchmarkTools` package can be installed as follows:

```
julia> using Pkg  
julia> Pkg.add("BenchmarkTools")
```

This normally stores the package after compilation and adds a reference to it in two files, `Project.toml` and `Manifest.toml`, more of which we will see later.

Deleting the modules is simple using the `Pkg.rm("BenchmarkTools")` statement.

It used to be quite simple using the built-in function `is_Pkg.installed()` but this now throws a method error and the appropriate way is to create a list of dependencies and check whether a package is a “direct” dependency, as follows:

```
julia> isinstalled(pkg::String) =
    any(x -> x.name == pkg &&
        x.is_direct_dep, values(Pkg.dependencies()));  
  
julia> isinstalled("BenchmarkTools")
true
```

The exact syntax of the function should be more apparent after we have discussed functions more thoroughly in *Chapter 2*.

To simplify the setup of packages on a per-chapter basis, I have created a script called `setup.jl`, which can be run from the command line as `julia setup.jl` or included directly in the REPL as `include("setup.jl")`.

Also, there are a pair of TOML files, `Project.toml` and `Manifest.toml`, which can be used by the Package Manager in order to “activate” specific folders (see next).

## Editors and IDEs

Julia has some alternatives, rather than working with the REPL. In chronological order, the main ones are as follows:

- Jupyter
- Juno
- Visual Studio Code
- Pluto

The sources accompanying the chapters of this book are mainly provided in source format (`.jl`), which can run in the REPL, but also conveniently in VS Code, which is now an almost de facto approach to developing Julia code.

In addition, some examples as IJulia (aka Jupyter notebooks (`.ipynb`) and Pluto workbooks (`.pluto.jl`) where relevant and each chapter contains a Markdown file, which lists the various source files and their relevance.

Juno is now virtually defunct, but I will address briefly why.

VS Code is a Visual Studio (free) development framework from Microsoft that can run a variety of extensions, one of which supports the Julia language. It can run code and display graphics, and it incorporates a debugging function; the latter we will get to in the final chapter of this book.

Pluto is a different style IDE (to Jupyter), which is reactive, and changing any values in individual cells will affect a change thorough out the entire workbook, so is popular when giving lectures and workshops, especially as it includes binding of values to widgets such as sliders, spinors, and buttons to make changing of values easy.

A full discussion is outside the scope of this book, but I will briefly introduce each, in chronological order, and provide some relevant references, current at the time of writing, for those of you who are not familiar with each.

There are also a number of “code” extensions for various editors, such as Notepad++, VIM, Sublime, and Emacs, which highlight the Julia coding syntax, but these are not as powerful as VS Code, and apart from this brief discussion here, their existence will not be dealt with again. For further information, refer to the **Julia Editor Support** group (<https://github.com/JuliaEditorSupport>) for download and documentation.

## *Jupyter*

Jupyter was developed by Steven Johnson from MIT in the early days of Julia, under the guise of IJulia, as a spinoff to the (then) IPython IDE. This was part of his work on interfacing with Python (PyCall) and creating a graphics package (PyPlot), both of which I will introduce in this book.

The Julia team then collaborated with the latter to incorporate the R language and hence came up with the portmanteau name JUPYteR, although to be frank, proponents of R have such a powerful IDE in RStudio that it is little used in that language. However, a consequence is that the collaboration clarified how to create other “kernels” for Jupyter, in C, Ruby, Perl (5 and 6), OCaml, Scala, Lisp, and many more; a full list is available via the pages on GitHub (<https://github.com/topics/jupyter-kernels>).

It can be installed by default from the IJulia package using the REPL by the method discussed previously:

```
using Pkg; Pkg.add("IJulia")
```

Adding the IJulia package will also add several other dependent packages, such as PyCall and PyPlot.

Jupyter is started by using the `IJulia` package and then using the `notebook()` function:

```
using IJulia; notebook()
```

This will start up Jupyter in a local browser and normally on a well-known port, usually 8888, but if that is taken, then 8889.

Python needs to be present, and using the Anaconda distribution (<https://www.anaconda.com/download>) is recommended if requiring a full, flexible version of Python. Otherwise, the

first time the `notebook()` command is issued, the user will be prompted to install a Miniconda (<https://docs.conda.io/en/latest/miniconda.html>) version, which will be used to interface with IJulia.

I have found it problematic to install Anaconda after IJulia, in as much as the whereabouts of the IJulia kernel can become confused, and this is possibly the only bug-bear in using IJulia. To address this issue, Steven Johnson provides a troubleshooting guide (<https://julialang.github.io/IJulia.jl/dev/manual/troubleshooting/>) as part of the IJulia manual pages.

Otherwise, IJulia should run in the current default web browser. The first screen is a file directory listing; it may be necessary to traverse the folder tree to find the desired notebook. Any files with `.ipynb` will be displayed, regardless of which kernels they are running on.

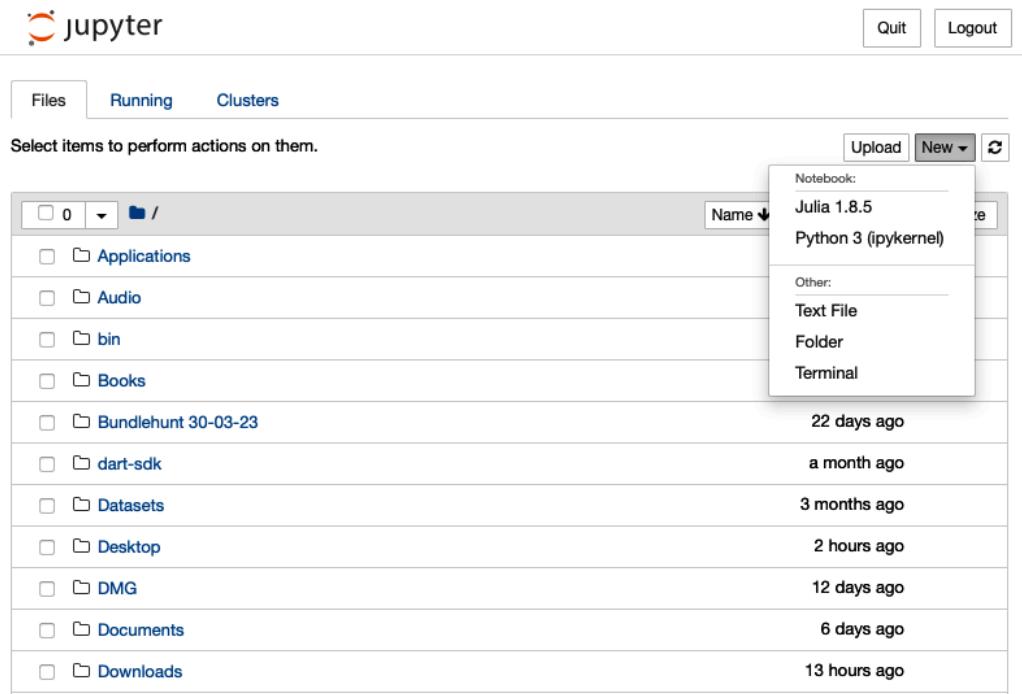


Figure 1.3 – Opening screen when starting Jupyter (aka IJulia)

By default, the notebook opens in your home directory; however, it can be opened in a different location with an argument in the `notebook` statement such as this:

```
notebook(dir=ENV("HOME") * "/MJ2/Code01"
```

There is a full discussion of IJulia provided by the manual pages on [julialang.github.io](https://julialang.github.io) and I recommend you refer to it for more information.

## Estimating Pi

A simple simple estimation of PI by generating pairs of random numbers (x,y).

This are in the unit square and the proportion of pairs with line in the unit circle will be  $\pi/4$ .

```
In [2]: # Define a function to compute the sum of the squares of (x,y)
```

```
sumsq(x,y) = x*x + y*y;
```

```
In [3]: # Set the number of trials and initialise the counter
# Using zero(Integer) will ensure the counter is an integer
```

```
N = 10^8;
K = zero(Integer);
```

```
In [4]: # Sum the number of pairs which lie within the inscribed circle
```

```
for i = 1:N
    if sumsq(rand(), rand()) < 1.0
        K += 1
    end
end
```

```
# ... and output the value for PI
```

```
@printf "Estimate of PI for %d trials is %8.5f\n" N 4.0*(K / N);
```

```
Estimate of PI for 100000000 trials is 3.14151
```

Figure 1.4 – Running the PI estimation script on Jupyter

### Note

After our discussion on scoping rules, notice that the code executes without the global prefix to the `K += 1` statement inside the `for` loop.

## Juno

Juno was developed by the admirable Mike Innes as a plugin to the Atom editor before he turned his attention to other Julia matters relating to machine learning such as Flux and Zygote (which we will meet later). It was then passed on to a group for support, referred to as [junolab.org](http://junolab.org). The plugin was added after installing Atom using the `uber-juno` extension and provided syntax checking, incremental running, variable value display, and debugging.

There was a time when Juno, like its mythological counterpart, was the *Queen of the Heavens*, but alas no more – the age of heroes is gone.

The reason was that doubts have been expressed as to the stability (and safety) of the Atom editor itself, and the group maintaining it, in their terms, sun-settled (<https://github.blog/2022-06-08-sunsetting-atom/>) the product as of December 2022 and advised the users to switch to VS Code. A consequence for Juno Lab was to take a similar path, but luckily, considerable work had been done on a VS Code replacement in Julia, which we will consider next.

## Visual Studio Code

**Visual Studio Code (VS Code)** is provided mostly free by Microsoft, although Microsoft does get certain feedback as to its usage. It is, like Atom, on the JavaScript engine, Electron, which itself is open source.

It should not be confused with Visual Studio (sans Code), which is a commercial product from Microsoft used in creating .NET programs in C#, C++.net, Visual Basic, F#, and so on, and normally available by subscription; the Enterprise Edition is quite expensive.

VS Code, however, is a free download for Windows, macOS, and Linux and is not related to .NET, so the user needs first to download it for the appropriate platform. Installation is relatively straightforward being an EXE (or for Linux, a DEB or RPM) file and, in the case of macOS, the application is zipped up and, after unzipping, is ready to go.

On startup, VS Code displays a **Welcome** page that invites the user to open a file or folder. The latter is quite useful as it switches to a list of files within that folder.

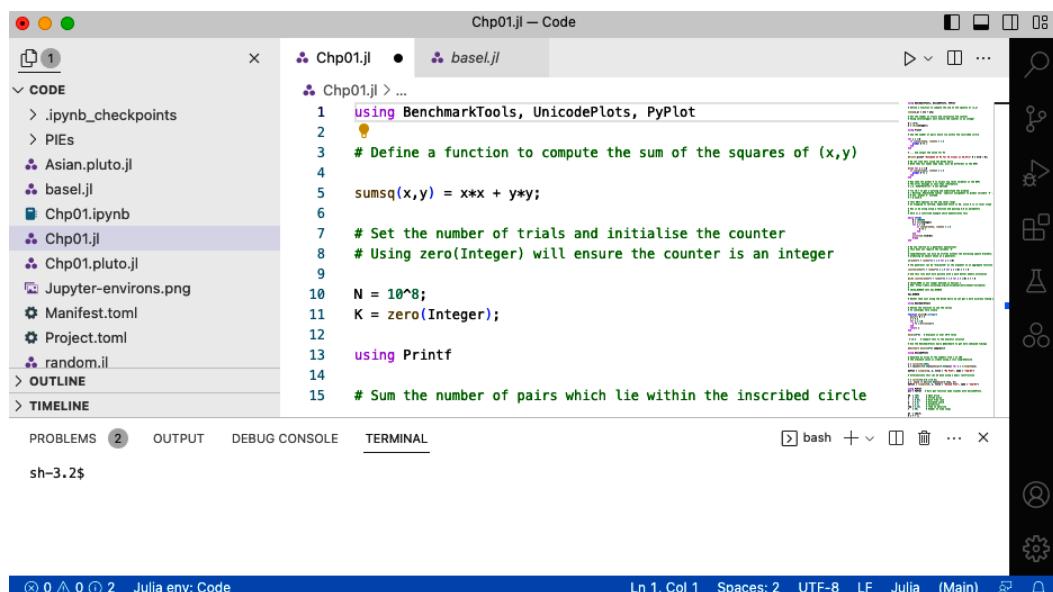


Figure 1.5 – Running the Chp01.jl code on VS Code editor

On first use, it is necessary to add the Julia extension. The set of widgets down the right-hand side is useful here; the fourth one down (*four squares*) will switch to an extension list and has an associated search box in which typing `Julia` will provide the correct one from JuliaLang.<sup>21</sup>

It is usual to tailor VS Code to meet your personal preference, font size, theme, and so on, and the website (<https://code.visualstudio.com/docs>) goes into this in some detail. Also, there is a website specifically targeted at the use of Julia on VS Code: <https://www.julia-vscode.org/docs/stable>. Particularly useful is the **Command Palette**, which can be displayed from the **View** menu or via the keyboard shortcut, *Shift + Command + P*.

It is usually more convenient to use key bindings, such as the preceding one, rather than the VS Code menu. A set for each operating system (Windows, macOS, and Linux) is available as a PDF from the `code.visualstudio.com` website, and I have provided a copy of these in the code section accompanying this book. Also, it is possible to display them using the shortcut *Command + K* followed by *Command + S*.

Finally, to execute code with a file displayed in the editor, highlight the line and use *Shift + Enter*. Any result will be accompanied by a “tick” if it runs OK or with its value if the statement returns one. Any `printf` statements or more extensive output will be displayed in a Terminal window.

Graphic output, using the `Plots.jl` API, can be created in a separate “plot” window via the Common Palette by issuing the `Julia: Show Plot` directive. Refer to *Chapter 8* on visualizations for examples of the use of the `Plots.jl` package.

## ***Pluto***

The final IDE to be discussed is `Pluto.jl`. This varies from Jupyter (say) as it is used to work with reactive notebooks. As well as the GitHub source on `Pluto.jl`, there is a separate website, <https://plutojl.org/>, which has more information on the package.

Being reactive means that changing a value in one cell instantly re-evaluates all other cells that reference that value.

A few consequences of this are as follows:

- Individual assignment statements need to be in separate cells unless enclosed in a `begin...end` block, and Pluto will prompt the user if this is not the case
- Values of a cell are displayed above the cell rather than below it
- Cell contents can be hidden, leaving only the output values visible

Any text statements will NOT be seen in the workbook, rather they are shown in the Terminal window. So, instead of using `printf` statements, it is usual to create a Markdown string, which is then output above the corresponding cell – that is, from the one example we have met so far on evaluating PI, in the `pi.pluto.jl` file:

```
md"The estimate of PI for $N trials is $P"
```

Here, `$N` is substituted by the value of `N` and `$P` by the computed estimate for  $\pi$ .

Note that a Pluto workbook is an ASCII Julia file and can be run in the REPL as well as under the IDE. The files have no specific extension associated with them but, conventionally, I use the `.pluto.jl` suffix to emphasize their function.

To install and run Pluto is particularly simple, just requiring a simple `using` statement followed by an associated `run`:

```
using Pkg; Pkg.add("Pluto")
using Pluto;
Pluto.run()
```

The first statement need only be executed once, of course.

The default browser is opened on port 1234, if available; otherwise, it is possible to specify a different port as an argument to the `run()` statement.

The user can then create a new Pluto workbook or browse for an existing one; also, a list of “recent” workbooks is displayed.

Pluto uses a backend graphics engine. It is recommended to use one that uses the `plotlyjs()` function; we will discuss it in its own right in the chapter on visualization.

As mentioned, Pluto has a supplementary package, `PlutoUI`, which needs to be installed in the usual way. This provides a set of widgets such as sliders and buttons that can be associated with a specific variable via a `@bind` macro.

See some of the examples in this book or at <https://featured.plutojl.org> to illustrate how the `UI` package is used.

## A quick look at some (more) Julia

In the next section of this chapter, we will look at a few examples to get a feel for what Julia’s code looks like and how it works, as you may be a little bored with the estimate of  $\pi$  we have seen so far.

Some of the code included in the scripts may be covered in more detail later in this book. However, it should be possible to follow the listings without too much difficulty.

### *The Basel problem*

First, a simple computation of an infinite series to solve the famous Basel problem. This is relatively easy to compute, and I’ve included listings for Python, R, and Octave, along with Julia in the code section of the accompanying code.

To get an accurate listing, it is necessary to run these sources from the OS; otherwise, interacting with Jupyter will swamp the computation. To this end, I have included a command script in the code

section accompanying this chapter (runnable in macOS and Linux) to perform accurate timings in Julia and in addition, in Python, R, and so on, assuming that these have been previously installed and can be started from the execution path.

The Basel problem is a problem in mathematical analysis with a purpose to number theory. It was first posed by Pietro Mengoli in 1644 and solved by Leonhard Euler in 1734, and presented in December of the following year to the Saint Petersburg Academy of Sciences.

Since the problem had resisted the best efforts of the leading mathematicians of the day, Euler's solution gave him immediate fame at the age of 28. Euler generalized the problem, and his ideas were taken up years later by Bernhard Riemann in his seminal 1859 paper, *On the Number of Primes Less Than a Given Magnitude*, in which he stated his zeta function and proved its basic properties.

The problem is named after Basel, the hometown of Euler as well as the Bernoulli family who unsuccessfully attacked the problem. It asks for the precise summation of the reciprocals of the squares of the natural numbers (i.e., the precise sum of the infinite series):

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{1}{1^2} + \frac{1}{2^2} + \frac{1}{3^2} + \dots$$

The sum of the series is approximately equal to 1.644934.

Euler found the exact sum to be  $\pi^2 / 6$ . He announced this discovery in 1735, but his arguments were based on manipulations that were not justified at the time, although he was later proven correct, and it was not until 1741 that he was able to produce truly rigorous proof.

The following script will compute the sum in Julia. The N parameter is constrained to be an integer; note the use of the `@assert` macro to ensure that this has a positive value:

```
julia> # Define the function to sum the series
function basel(N::Integer)
    @assert N > 0
    s = 0.0
    for i = 1:N
        s += 1.0/float(i)^2
    end
end
#= The function will return 's' as its result as it is the final value to
be computed
=#
julia> basel(10^8) # Evaluate it over 100,000,000 terms
1.644934057834575
```

The bash script provides `basel.sh` runs the Julia code (under macOS and Linux), which compares accurate timings against Python, R, and Octave:

```
$> . $HOME/MJ2/Code01/basel.sh
using Python
Basel estimate is 1.64493396685
Number of terms: 10000000
Time taken was 2.83526992798 sec.
...
using R
[1] "BASEL estimate : " "1.64493396684726"
[1] "Number of terms in series: " "1e+07"
[1] "Time taken: " "5.81213307380676"
...
using Octave
Number of terms is 10000000
Elapsed time is 30.0596 seconds.
Value of BASEL series = 1.644933
using Julia
0.048639 seconds (86 allocations: 6.498 KiB)
Basel estimate 1.64493397 over 10000000 terms
```

Julia takes around 50 msec compared with 2.8 seconds for Python, 5.8 seconds for R, and 30 seconds for Octave (on my MacPro laptop).

To produce a more complete picture, it is useful to use the package called `BenchmarkTools`, which runs a series of tests and outputs the median, mean, maximum, and minimum timings:

```
julia> Pkg.add("BenchmarkTools");
julia> using BenchmarkTools
julia> @benchmark basel(10^8) samples=10
BenchmarkTools.Trial:
memory estimate: 0 bytes
allocs estimate: 0
-----
minimum time: 497.727 ms (0.00% GC)
median time: 506.581 ms (0.00% GC)
mean time: 510.996 ms (0.00% GC)
maximum time: 547.644 ms (0.00% GC)
-----
samples: 10
evals/sample: 1
```

## Displaying some inline graphics

The next example is how to compute some graphic output in the REPL using a package called `UnicodePlots`.

The following code is of a simple function,  $x^*\sin(3x)^*\exp(-0.03x)$ :

```
julia> using Pkg
julia> Pkg.add("UnicodePlots");
julia> using UnicodePlots
# Generate an array of the numbers from 1 to 100
# The ordinate value is created using a list comprehension
x = collect(1:100);
y = [x[i]*sin(0.3*x[i])*exp(-0.03*x[i]) for i = 1:length(x)];
myPlot = lineplot(x, y, title = "My Plot", name = "chp-01")
# Alternatively this can be done using a map() construction such as:
t = collect(0.0:0.1:10.0);
y = map(t -> t*sin(3.0*t)*exp(-0.3*t), x);
```

The code uses a couple of methods to compute the function values, one via a list comprehension and the other using a `map()` construct; a third method will be discussed later when Julia's broadcasting method is discussed.

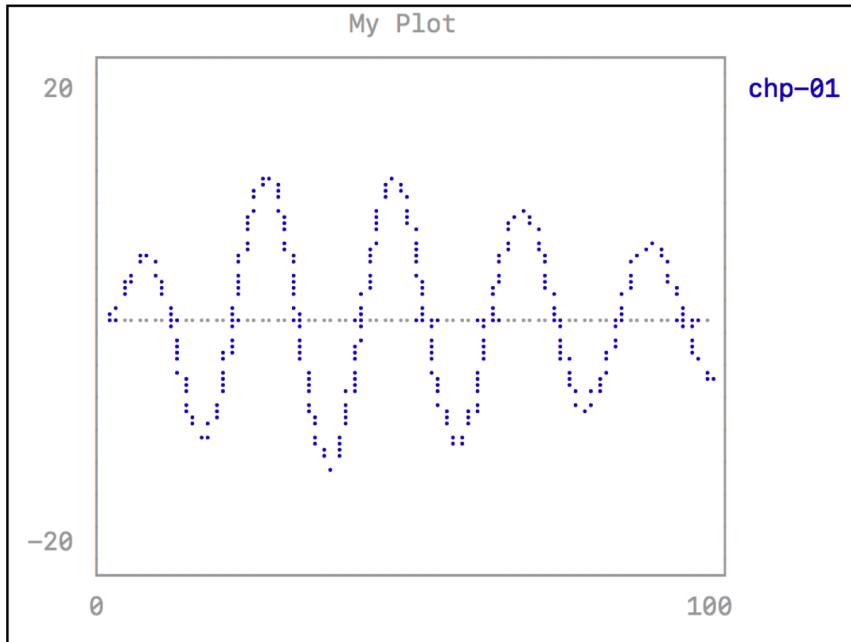


Figure 1.6 – Output of the Unicode single lie plot in the REPL

The `UnicodePlots` package is capable of displaying various other plots than simple line graphs; look at the module source (<https://github.com/JuliaPlots/UnicodePlots.jl>) for more information.

As a second example, let's generate some normally distributed random numbers, display the histogram, and check the elementary statistics of the distribution.

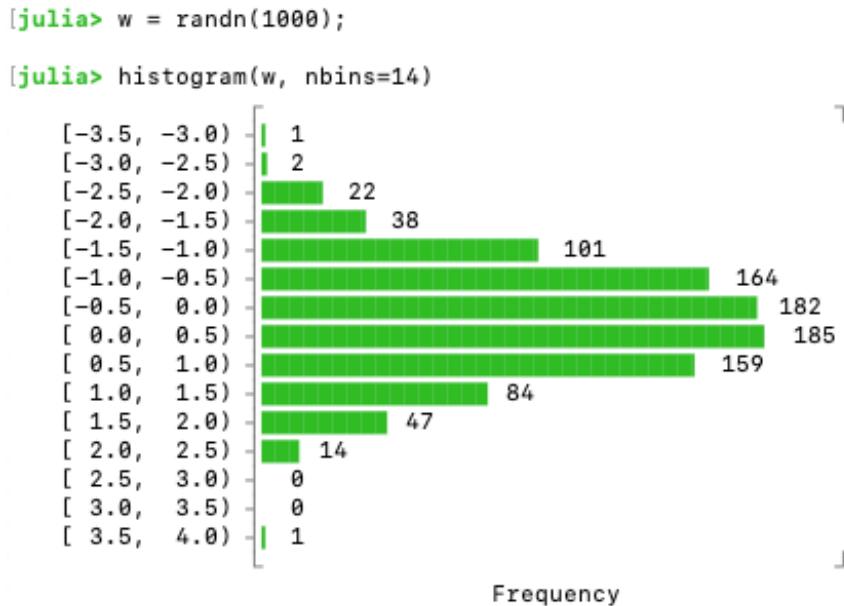


Figure 1.7 – Output of the Unicode histogram plot in the REPL

To calculate the stats for the distribution, we will use both `Statistics` (for mean and standard deviation) and `StatsBase` (for skewness and kurtosis):

```
julia> using Statistics, StatsBase
julia> a = [mean(w) std(w) skewness(w) kurtosis(w)]
1x4 Matrix{Float64}:
 -0.0262424  0.979408  0.00278381  -0.142612
```

For the default normal distribution, we expect a zero mean and unit standard deviation with no skewness. The usual kurtosis value should be around 3, but the `StatsBase` module returns the “excess” value, which is simply the calculated value minus 3, so we expect around a 0 value as well.

For only 1,000 values, the statistics are hardly representative. The following is the same calculation from a million values:

```
julia> using BenchmarkTools
julia> @btime begin
w = randn(1000000);
a = [mean(w) std(w) skewness(w) kurtosis(w)]
end
6.052 ms (7 allocations: 7.63 MiB))
1x4 Matrix{Float64}:
-0.00021976  1.00039  0.0019657  0.00500196
```

In this case, I have used the `@btime` macro from the `BenchmarkTools` package to compute the timings from a series of trials and skip the compilation step. Even for one million values, this is still remarkably rapid.

#### Note

Timing the code initially takes considerably longer in the first instance due to the compilation:  
0.217167 seconds (273.24 k allocations: 25.287 MiB, 10.22% gc time, 92.50% compilation time)

### ***Computing geometric Brownian trajectories***

When we look at Julia's functions in more detail in *Chapter 4*, we will use an example of the computations of stock derivatives known as Asian options in financial markets.

I'll defer a detailed discussion of the stock options until then; here, all we need to note is the following:

- The cost of a *normal* option is determined by the final price of the stock
- The stock is assumed to move with geometric Brownian motion
- The volatility of the stock is assumed constant
- An Asian option differs from a normal option in as much as the mean value is
- used to compute the cost rather than the final price

There is a formula for computing the price of the contract for a normal option; with an Asian one, we need to compute a series of trajectories over many runs and use these to come up with a cost to the broker of purchasing the contract.

This approach is known as Monte Carlo simulation and depends on the generation of random numbers to model the stochastic variation of the stock around a deterministic trend.

Here, I'm just going to look at the code required to produce some of these trajectories; it is relatively short to do this and needs no special features other than simple coding.

We will use PyPlot to display the graphics. This should be installed if you have previously added IJulia; otherwise, add it with the package manager.

The following code computes five trajectories based on a geometric random walk. The first part of the script imports the PyPlot package, sets some parameter values, and adds a title and labels for the plot. The computing is done over the two loops – the outer one to create the five trajectories to be displayed and the inner one to perform the actual computation and store the values in the  $S[0:N]$  array:

```
using PyPlot
plt = PyPlot

S0 = 100; # Spot price
K = 102; # Strike price
r = 0.05; # Risk free rate
q = 0.0; # Dividend yield
v = 0.2; # Volatility
tma = 0.25; # Time to maturity
T = 90; # Number of time steps

dt = tma/T;
N = T + 1;
x = collect(0:T);

plt.title("Asian Option trajectories");
plt.xlabel("Time");
plt.ylabel("Stock price");

for k = 1:5
    S = zeros(Float64,N)
    S[1] = S0;
    dW = randn(N)*sqrt(dt);
    for t = 2:N
        z1 = (r - q - 0.5*v*v)*dt
        z2 = v*dW[t]
        z3 = 0.5*v*v*dW[t]*dW[t]
        S[t] = S[t-1] * (1 + z1 + z2 + z3)
    end
    plt.plot(x,S)
end
```

The display is created using the PyPlot package.

There is a more recent version, `PythonPlot`, which interfaces to the Python `matplotlib` plotting library via the `matplotlib.pyplot` module and will be discussed later.

The first few lines set the parameters for the computation and create an `x` vector using a `collect()` statement. Each trajectory is computed inside a `for` loop, in which the values are stored in a vector array, `S`, via the `zeros()` statement.

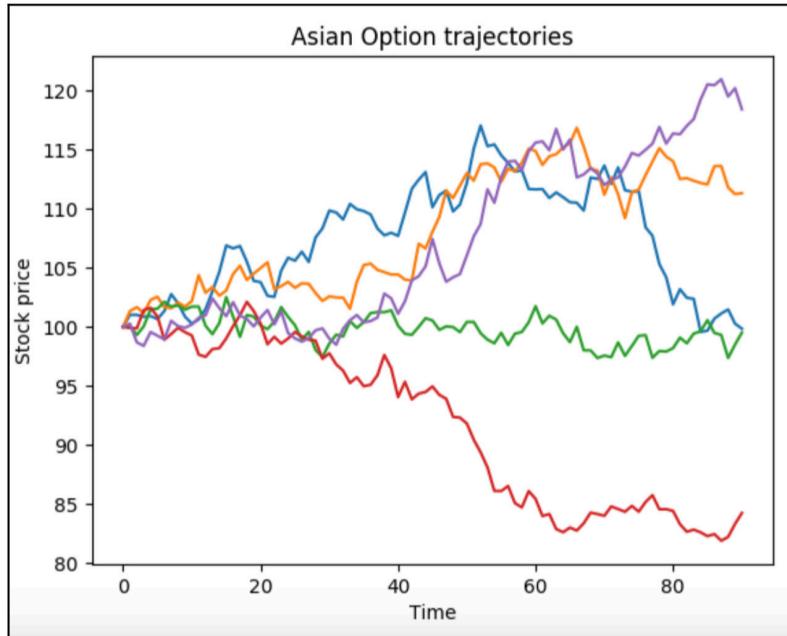


Figure 1.8 – Asian option tracks displayed using the PyPlot package

The Asian option is known as a Weiner process (i.e., Brownian motion in which the stochastic variable is proportional to the square root of the time – in this case, the volatility of the stock).

An inner loop is the one that does the heavy lifting. The formula arises from the Black-Scholes analysis of simple stock options, which we will meet when discussing stochastic differential equations in *Chapter 7*.

Each individual track is displayed “on the fly” in the inner loop.

## Package management

We have noted that Julia uses GitHub as a repository for the language source and a large number of Julia packages are hosted there.

**Note**

Other sources for Julia packages may be found on Bitbucket, GitLab, and so on; see stackshare.io for a full discussion: <https://stackshare.io/stackups/bitbucket-vs-github-vs-gitlab>.

As a full discussion of the package system is given on the Julia website, here, we will cover some of the main commands to use.

In v1.0, a new package manager has been introduced. Given the increase in registered packages, now approaching 2,000, the previous version (which relied on separate invocations of Git) became very slow, especially when using Windows.

Earlier, a method using the Pkg interface for package management was discussed. An alternate approach is to use one of the subset systems from the REPL. This is one of three such subsystems:

**Online Help:** Triggered by typing ?

- **Command Shell:** Triggered by typing ;

**Package Manager:** Triggered by typing ]

All three are exited back to the REPL by hitting *Return/Enter* with no input. The third one will be discussed briefly here.

## Listing, adding, and removing packages

So, to enter the command line of Pkg, type ] at the REPL prompt, which then changes to (v1.0) pkg> ; exiting the package manager can be done either by typing a backspace or *Ctrl + C*.

Pkg uses different means to maintain its package metadata, track dependencies, and assess update requirements, and utilizes a number of folders in \$HOME/.julia.

Repositories are in `environmental/v1.0` and Pkg uses **TOML (Tom's Obvious Minimal Language)** format. The other main folder is `packages`, which keeps a separate user directory/folder to keep the local copies of the packages.

Different from previous package managers, multiple copies of packages are maintained in a set of five alphanumeric hash subfolders; these are one for any top-level package, (i.e., added explicitly), and the other when a package is the dependency of another TLP.

When in Pkg, typing `help` (or just ?) displays a summary of all the commands:

```
help: show this message
st(atus): summarize contents of and changes to environment
add: add packages to project
rm: remove packages from project or manifest.
```

```
up(date): update packages in manifest.  
preview: previews a subsequent command without affecting the current  
state.  
test: run tests a package.  
gc: garbage collect packages not used for a significant time.  
init: initializes an environment in the current, or git base,  
directory.  
build: run the build script for a package.
```

The `add` and `rm` commands are used to install new packages and remove them, respectively; updating installed ones is done by the `up` command (or `update`).

The following is a typical `Pkg` session to install `BenchmarkTools`, which has been installed previously, and then remove it. The `init` command is not strictly required since an implicit initialization will occur on the addition of the first package:

```
pkg> init  
INFO: Initialized environment in /Users/malcolm by creating the file  
Project.toml  
pkg> status  
INFO: Status "~/Project.toml"  
pkg> update  
INFO: Updating registry at /Users/malcolm/.julia/registries/Uncurated  
INFO: Resolving package versions  
INFO: Updating "~/Project.toml"  
[no changes]  
INFO: Updating "~/Manifest.toml"  
[no changes]  
pkg> add BenchmarkTools  
INFO: Resolving package versions  
INFO: Installed Nullables ----- v0.0.3  
INFO: Installed JSON ----- v0.16.4  
INFO: Installed BenchmarkTools - v0.2.4  
INFO: Updating "~/Project.toml"  
[6e4b80f9] + BenchmarkTools v0.2.4  
INFO: Updating "~/Manifest.toml"  
[6e4b80f9] + BenchmarkTools v0.2.4  
[34da2185] + Compat v0.49.0  
[682c06a0] + JSON v0.16.4  
pkg> status  
INFO: Status "~/Project.toml"  
[6e4b80f9] BenchmarkTools v0.2.4  
[4d1e1d77] + Nullables v0.0.3  
pkg> rm BenchmarkTools  
INFO: Updating "~/Project.toml"
```

```
[6e4b80f9] - BenchmarkTools v0.2.4
INFO: Updating "~/Manifest.toml"
[6e4b80f9] - BenchmarkTools v0.2.4
[34da2185] - Compat v0.49.0
[682c06a0] - JSON v0.16.4
[4d1e1d77] - Nullables v0.0.3
pkg> status
INFO: Status "~/Project.toml"
pkg> ^C
julia>
```

It should be noted that removing a package only deletes the TLP version and not any dependencies of foreign packages that were installed at the same time; to clean up any such zombie packages, it is necessary to issue the additional command, `gc`.

It is also possible to use the package manager from within Julia code by using the `Pkg` API, which must first be imported with a `using` statement; for example, we can add the `BenchmarkTools` package by using the following:

```
using Pkg
Pkg.add("BenchmarkTools")
```

Of course, `Pkg` is capable of adding packages not (yet) in the official repository, via the GitHub URL and also any local packages you may have written; I will deal with the latter in the final chapter of this book.

There is quite an extensive discussion of the new package manager in the Julia documentation at <https://docs.julialang.org/en/v1/stdlib/Pkg/>.

## Testing a package

We will be discussing the format of a Julia module later. In addition to having the source code in an `src` folder, a number of tests to exercise the module should be provided in the `test` folder. These tests are “triggered” from the `runtests.jl` script.

There may also be an extra folder, such as `docs`, comprising documentation, although in the simplest cases, this may be covered in `README.md`.

Running the set of tests (i.e., via the `runtests` script) can be done in the Package Manager using the `test` command.

Here is an example of the `PyCall` module on my macOS installation:

```
pkg> test PyCall
Testing Running tests...
```

```
Info: Python version 3.10.10 from /Users/malcolm/.julia/conda/3/x86_64/lib/libpython3.10.dylib, PYTHONHOME=/Users/malcolm/.julia/conda/3/x86_6
```

Test Summary:	Time		
CI setup	None	0.2s	
conversions	470	470	18.7s
pydef	8	8	0.3s
callback	3	3	0.1s
throwing show	4	4	0.2s
PyIterator	14	14	0.3s
atexit	1	1	2.6s
pycall!	16	16	1.1s
PyBuffer	88	88	2.6s

```
Info: No virtualenv command. Skipping the test...
```

```
virtualenv activation | None 0.0s
```

```
Info: Skip venv test with conda.
```

```
venv activation | None 0.0s
```

Test Summary:	Time		
find_libpython	7	7	0.9s
@pyinclude	2	2	0.0s
proper exception raised	4	4	0.0s

## Choosing and exploring packages

For such a young language, Julia has a rich and rapidly developing set of packages covering all aspects of use to the data scientist and mathematical analyst. Registered packages are available on GitHub and the list of such can be referenced via <http://pkg.julialang.org>.

Because the core language is still under review from release to release, some features are deprecated, others changed, and yet others dropped. So it is possible that specific packages may be at variance with the release of Julia you are using, even if it is designated as the current stable one. Also, it may be the case that the package may not work under different operating systems.

How then should we select a package? Even with an old, relatively untouched package, there is nothing to stop you from checking out the code and modifying or building on it. Any enhancements or modifications can be applied and the code returned; that's how open source grows. Also, the principal author is likely to be delighted that someone else is finding the package useful and taking an interest in the work.

Many packages have been adopted by a specific community group (e.g. JuliaStats, JuliaDB, JuliaPlots, etc.), and these are likely to be well maintained and kept up to date, and any issues will be resolved rapidly when flagged up.

### **Statistics and mathematics**

Statistics is rightly seen as the realm of R, and mathematics of MATLAB and Mathematica, while Python impresses in both. The base Julia system provides much of the functionality available in NumPy, while additional packages are adding that of SciPy and pandas.

Statistics are well provided in Julia both on GitHub by the JuliaStats group (<https://github.com/JuliaStats>) and the group's site (<http://juliastats.github.io>), and on Google Groups at <https://groups.google.com/forum/#!forum/julia-stats>.

Much of the basic statistics are provided by `StatsBase.jl` and `DataFrames.jl`. There are means for working with R-style data frames and for loading some of the dataset available to R and even calling R modules using `RCall.jl`.

The `Distributions.jl` package covers probability distributions and associated functions; also, there is support for time series, cluster analysis, hypothesis testing, MCMC methods, and more. JuliaStats is now incorporating machine learning, and I am devoting a new chapter (*Chapter 10*) to looking at the work being done here.

Mathematical operations such as random number generators, exotic functions, and so on are largely in the core (unlike Python), but packages exist for elemental calculus operations, ODE solvers, Monte Carlo methods, mathematical programming, and optimization. There is a GitHub page for the JuliaOpt group (<https://github.com/JuliaOpt/>), which lists the packages under the umbrella of optimization.

### **Graphics**

Graphic support in Julia has sometimes been given less than favorable press in comparison with other languages such as Python, R, and MATLAB. It is a stated aim of the developers to incorporate some degree of graphic support in the core, but at present, this is largely the realm of package developers.

While it was true that early versions of Julia offered very limited and flaky graphics, the situation vastly improved, and now the breadth of graphics available is quite staggering.

We have met two approaches already using `UnicodePlots` for ASCII character terminal graphics and `PyPlot`, which is a wrapper package around the `matplotlib` Python module.

An early module and a favorite of mine is Winston. This is a 2D graphics package that provides methods for curve plotting and creating histograms and scatter diagrams. Axis labels and display titles can be added, and the resulting display can be saved to files as well as shown on the screen.

Another early package is Gadfly, which is a system for plotting and visualization equivalent to the `ggplot2` module in R. It can be used to render graphic output to PNG, PostScript, PDF, and SVG files. Gadfly works best with the `cairo`, `pango`, and `fontconfig` C libraries installed. The PNG, PS, and PDF backends all require Cairo, but without it, it is still possible to create displays in SVG and JavaScript/D3. At the time of writing, Gadfly is not v1.0 compliant but I will include a discussion of it in the later chapter on graphics, and I assume it will be fully functional by the time this book is published.

The JuliaPlots group now supports a general API (`Plots.jl`), which aims to provide a general calling interface to a series of graphic backends. While neither Gadfly nor Winston supports the API, `PyPlot` does, and newer modules: GR, PlotlyJS, and Makie.

We will look at all of these later in *Chapter 8*, which is devoted entirely to graphics.

## ***Web and networking***

Distributed computing is well represented in Julia. TCP/IP sockets are implemented in the core. Additionally, there is support for `curl` and SMTP and WebSocket.

HTTP protocols and parsing are provided with a number of packages such as HTTP, `HttpParser`, `HTTP Server`, `JSON`, and `Mustache`.

Working in the cloud, at present, there are a couple of packages – AWS, which addresses the use of Amazon, **Simple Storage System (S3)**, and **Elastic Compute Cloud (EC2)**. The other, `HDF5`, provides a wrapper over `libhdfs` and a Julia map-reduce functionality.

The JuliaParallel group has provided a number of packages to implement support for parallel, multiprocessor, and distributed processing. We will be discussing this work later in the book.

## ***Database packages***

The database is supported mainly through the use of the ODBC package. On Windows, ODBC is standard, while on Linux and macOS, it requires the installation of UnixODBC or iODBC. A similar approach is to use database connectivity via JDBC and JavaCall.

At the time of writing, there is currently no native support for the main SQL database, such as Oracle and SQL Server/Sybase. Further support for databases such as MariaDB, MySQL, and PostgreSQL is limited, but this may have changed as this book is being read.

The JuliaDatabase group has provided a general **database interface (DBI)** similar to the facility in Perl, where it becomes a simple matter to implement a database driver interface to API. The `SQLite` package provides an interface to DBI.

There is a Mongo package that implements bindings to the NoSQL database, MongoDB.

Other NoSQL databases, such as CouchDB and Neo4j, exposed a RESTful API, so some of the HTTP packages coupled with JSON can be used to interact with these. However, many of the NoSQL packages

have received little attention recently and it may well be necessary to discuss other non-native methods using Python libraries and REST.

## Machine learning

The future role of Julia is thought to lie with machine learning rather than conventional data science. We have remarked that because Julia creates “compiled” code using the LLVM framework, Julia runtimes are comparable to those of C/C++ and Fortran rather than Python and R. The latter languages rely on wrappers around low-level C (usually) APIs, and although Julia can interface with C in a very straightforward fashion, often, it is easier to stay with native Julia code.

In addition, switching from the CPU to the GPU is often very simple, and large-scale *strides* are being made in the use of the GPU. Julia now has a wealth of modules for automatic differentiation, Bayesian estimate, conventional ML training methods, neural networks, and more.

## Final thoughts

All the material covered in this chapter will be looked at in more detail in the rest of the book. The aim was to indicate what a simple, straightforward, yet powerful language Julia is.

Julia has been maturing for nearly 12 years, and with the advent of the v1.0+ releases, the formation of the commercial company, JuliaHub (*previously named Julia Computing*), and the growth of modules (approaching 10,000 in number at the time of writing), it has never been a better time to study and perhaps adopt Julia as a programming language of choice.

The fact that all three of the original developers are still actively involved with the evolution of the language as well as playing major roles within JuliaHub is a testament to the faith that they and many others are putting in it

## Summary

This chapter introduced you to Julia, and how to download it, install it, and build it from the source. We saw that the language is elegant, concise, and powerful, and introduced some simple Julia coding examples.

The next four chapters will discuss the features of Julia in more depth.

We looked at interacting with Julia via the command line (REPL) in order to use a random walk method to evaluate the price of an Asian option. Also, we discussed the use of the VS Code editor and two IDEs, IJulia and Pluto, as an alternative to the REPL.

Additionally, we reviewed the built-in package manager, how to add, update and remove modules, and then demonstrated the use of two graphics packages to display typical trajectories of the Asian option calculation. In later chapters, we will look at various other approaches in order to create display graphics and quality visualizations.

In the next chapter, we will begin our detailed look at coding in Julia by discussing its primitive types and how to write functions to manipulate them.



# 2

## Developing in Julia

Julia is a feature-rich language. It was designed to appeal to novice programmers and purists alike. For those whose interests lie in data science, statistics, and mathematical modeling, Julia is well-equipped to meet all their needs.

Our aim is to furnish you with the necessary knowledge to begin programming in Julia almost immediately. So, rather than begin with an overview of the language's syntax, control structures, and the like, we will introduce Julia's facets gradually over the rest of this book. Over the next four chapters, we will look at some of the basic and advanced features of the Julia core. Many of the features—such as graphics and database access, which are implemented via the package system—will be left until later when discussing more specific aspects of programming Julia.

In this chapter, we will be discussing manipulating Julia's data structures and will cover the following topics:

- Data types such as integers and floating-point and complex numbers
- Vectors, matrices, and multi-dimensional arrays
- List comprehensions and broadcasting
- Recursive functions
- Characters and strings
- Complex and rational numbers
- Data arrays and data frames
- Dictionaries, sets, stacks, and queues

If you are familiar with programming in Python, R, MATLAB, and so on, you will not find the journey terribly arduous; in fact, we believe it will be a particularly pleasant one.

## Technical requirements

All code files are placed on GitHub at <https://github.com/PacktPublishing/Mastering-Julia-Second-Edition>. Refer to the section in the Preface for details on how to download and run them.

## Integers, bits, bytes, and Booleans

While Julia is usually dynamically typed—that is, in common with most interpreted languages, it does not require the type to be specified when a variable is declared; rather, it infers it from the form of the declaration. However, it also can be considered as a strongly typed language and, in this case, allows the programmer to specify a variable's type precisely.

A variable in Julia is any combination of upper- or lowercase letters, digits, and the underscore (\_) and exclamation (!) characters. It must start with a letter or an underscore.

Conventionally, variable names consist of lowercase letters with long names separated by underscores rather than using camel case.

To determine a variable type, we can use the `typeof()` function, as follows:

```
julia> x = 2;    typeof(x) # => gives Int  
julia> x = 2.0;  typeof(x) # => gives Float
```

Notice that the type (see the preceding code) starts with a capital letter and ends with a number, which indicates the number of bit length of the variable. The bit length defaults to the word length of the operating system, and this can be determined by examining the `WORD_SIZE` built-in constant, as follows:

```
julia> WORD_SIZE # => 64 (on my MacPro computer)
```

In this section, we will be dealing first with integer and Boolean types.

### Integers

An integer type can be any of `Int8`, `Int16`, `Int32`, `Int64`, and `Int128`, so the maximum integer can occupy 16 bytes of storage and be anywhere within the range of -2127 to (+2127 - 1).

If we need more precision than this, Julia core implements the `BigInt` type:

```
julia> x = BigInt(2^32)  
6277101735386680763835789423207666416102355444464034512896
```

As well as the integer type, Julia provides the unsigned integer type, `UInt`; again, `UInt` ranges from 8 to 128 bytes, so the maximum `UInt` value is  $(2^{128} - 1)$ .

We can use the `typemin()` and `typemax()` functions to output the ranges of the `Int` and `UInt` types, like so:

```
julia> for T =
    Any[Int8, Int16, Int32, Int64, Int128, UInt8, UInt16, UInt32, UInt64, UInt128]
    println("$(@pad(T, 7)) : [$(typemin(T)), $(typemax(T))] ")
end
Int8: [-128, 127]
Int16: [-32768, 32767]
Int32: [-2147483648, 2147483647]
Int64: [-9223372036854775808, 9223372036854775807]
Int128: [-170141183460469231731687303715884105728,
          170141183460469231731687303715884105727]
UInt8: [0, 255]
UInt16: [0, 65535]
UInt32: [0, 4294967295]
UInt64: [0, 18446744073709551615]
UInt128: [0, 340282366920938463463374607431768211455]
```

Particularly, notice the use of the form of the `for` statement, which we will discuss when we deal with arrays and matrices later in this chapter.

Suppose we type the following:

```
julia> x = 2^32; x*x # => the answer 0
```

The reason for the answer being 0 is that the integer “wraps” around, so squaring  $2^{32}$  gives 0, not  $2^{64}$ , since my `WORD_SIZE` value is 64:

```
julia> x = int128(2^32); x*x
# => the answer we would expect 18446744073709551616
```

We can use the `typeof()` function on a type such as `Int64` in order to see what its parent type is:

```
# So typeof(Int64) gives DataType and typeof(UInt128) also gives
DataType.
```

A definition of `DataType` is hinted at in the `boot.jl` core file; I say *hinted at* because the actual definition is implemented in C, and the Julia equivalent is commented out.

Definitions of the integer types can also be found in `boot.jl`, this time not commented out.

In the next chapter, we will discuss the Julia type system in some detail. Here, it is worth noting that we distinguish between two kinds of data types: **abstract** and **primitive** (concrete).

The general syntax for declaring an abstract type is shown here:

```
abstract type <<name>> end  
abstract type <<name>> <: <<supertype>> end
```

Typically, this is how it would look:

```
abstract type Number end  
abstract type Real <: Number end  
abstract type AbstractFloat <: Real end  
abstract type Integer <: Real end  
abstract type Signed <: Integer end  
abstract type Unsigned <: Integer end
```

Here, the `< :` operator corresponds to a subclass of the parent.

Let's suppose we type the following:

```
julia> x = 7; y = 5; x/y # => this gives 1.4
```

Here, the division of two integers produces a real result. In interactive mode, we can use the `ans` symbol to correspond to the last answer—that is, `typeof(ans)` gives `Float`.

To get the integer divisor, we use the `div(x, y)` function, which gives 1, as expected, and `typeof(ans)` is `Int64`. The remainder is obtained either by `rem(x, y)` or by using the `%` operator.

Julia has one curious operator—the backslash. Syntactically, `x\y` is equivalent to `y/x`. So, with `x` and `y`, as before, `x\y` gives 0.71428 (to 5 decimal places).

## Primitive types

A primitive type is a concrete type whose data consists of a series of bits. Examples of primitive types are the (well-known) integers and floating-point values that we have met previously.

The general syntax for declaring a primitive type is like that of an abstract type but with the addition of the number of bits to be allocated:

```
primitive type <<name>> <<bits>> end  
primitive type <<name>> <: <<supertype>> <<bits>> end
```

Since Julia is written (mostly) in Julia, a corollary is that Julia lets you declare your own primitive types, rather than providing only a fixed set of built-in ones.

That is, all the standard primitive types are defined in `Base` itself, as follows:

```
primitive type Float16 <: AbstractFloat 16 end  
primitive type Float32 <: AbstractFloat 32 end
```

```

primitive type Float64 <: AbstractFloat 64 end
primitive type Bool <: Integer 8 end
primitive type Char 32 end
primitive type Int8 <: Signed 8 end
primitive type UInt8 <: Unsigned 8 end
primitive type Int16 <: Signed 16 end
primitive type UInt16 <: Unsigned 16 end
primitive type Int32 <: Signed 32 end
primitive type UInt32 <: Unsigned 32 end
primitive type Int64 <: Signed 64 end
primitive type UInt64 <: Unsigned 64 end
primitive type Int128 <: Signed 128 end
primitive type UInt128 <: Unsigned 128 end

```

Note that only sizes that are multiples of 8 bits are supported, so Boolean values, although they really need just a single bit, cannot be declared to be any smaller than 8 bits. *Figure 2.1* demonstrates a portion of the Julia hierarchical structure as it applies to simple numerical types:

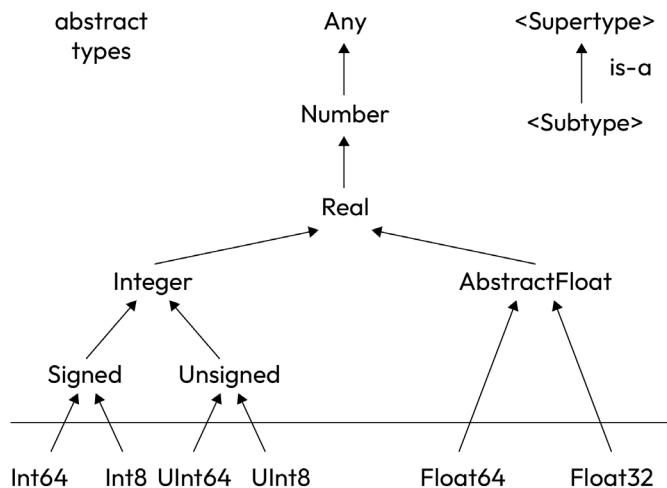


Figure 2.1 – Tree structure for numerical types

Those above the line are abstract types beginning with Any and cascading down through Number and Real before splitting into Integer and AbstractFloat types, eventually reaching the primitive types defined in Julia Base, which are shown below the line.

Primitives can't be subclassed further, hence terminating the various branches of the tree.

## Logical and arithmetic operators

As well as decimal arguments it is possible to assign binary, octal, and hexadecimal ones using the `0b`, `0o`, and `0x` prefixes.

So, `x = 0b110101` creates the hexadecimal number `0x35` (that is, decimal 53), and `typeof(ans)` is `UInt8` since 53 will “fit” into a single byte.

For larger values, the type is correspondingly higher—that is, `x = 0b1000010110101` gives `x = 0x10b5`, and `typeof(ans)` is `UInt`.

When operating on bits, Julia provides `~` (not), `|` (or), `&` (and), and `$` (xor):

```
julia> x = 0xbb31; y = 0xaa5f;
julia> x$y
0x116e
```

Also, we can perform arithmetic shifts using the `(LEFT)` and `(RIGHT)` operators.

### Note

Because `x` is of the `UInt16` type, the shift operator retains that size, so `x = 0xbb31; x<<8`. This gives `0x3100` (the top two nibbles being discarded), and `typeof(ans)` is `UInt`.

## Booleans

Julia has the `Bool` logical type. Dynamically, a variable is assigned a `Bool` type by equating it to the `true` or `false` constant (both lowercase), or alternatively, to a logical expression such as the following:

```
julia> p = (2 < 3) # => true
julia> typeof(p)    # => Bool
```

Many languages treat 0, empty strings, and `NULL` instances as representing `false` and anything else as `true`. This is *NOT* the case in Julia, however; there are cases where a `Bool` value may be promoted to an integer, in which case `true` corresponds to unity.

That is, an expression such as `x + p` (where `x` is of the `Int` type and `p` of the `Bool` type) will output the following:

```
julia> x = 0xbb31; p = (2 < 3);
julia> x + p
0xbb32
julia> typeof(ans) # => UInt16
```

## Big integers

Let's consider the factorial function defined by the usual recursive relation:

```
# n! = n*(n-1)! for integer values of n (> 0)
function fac(n::Integer)
    @assert n > 0
    (n == 1) ? 1 : n*fac(n-1)
end
```

Note that since normally, integers in Julia *overflow* (a feature of **Low-Level Virtual Machine (LLVM)**), the preceding definition can lead to problems with large values of n, as illustrated here:

```
julia> using Printf
      for i = 20:30
          @printf "%3d : %d\n" i fac(i)
      end
20 : 2432902008176640000
21 : -4249290049419214848
22 : -1250660718674968576
23 : 8128291617894825984
24 : -7835185981329244160
25 : 7034535277573963776
26 : -1569523520172457984
27 : -5483646897237262336
28 : -5968160532966932480
29 : -7055958792655077376
30 : -8764578968847253504

# Since a BigInt <: Integer,
# if we pass a BigInt the routine returns the correct value
julia> fac(big(30))
265252859812191058636308480000000
# See can check this since integer values: Γ(n+1) === n!
julia> gamma(31)
2.6525285981219107e32
```

The `big()` function uses string arithmetic, so it does not have a limit imposed by the `WORD_SIZE` constant but is clearly much slower than using conventional arithmetic. The `big()` function is not only restricted to integers but can be applied to reals (floats) or even complex numbers.

We can introduce a new function, `|>`, which applies a function to its preceding argument, providing a chaining functional style:

```
julia> 30 |> big |> fac  
265252859812191058636308480000000
```

Here, the 30 argument is piped to the factorial function but after first being converted into a `BigInt` type.

Also, note that the syntax is equivalent to `fac(big(30))`.

For now, we are going to leave our discussion on functions and begin to study in depth how arrays are constructed and used in Julia.

## Arrays

An array is an indexable collection of (normally) heterogeneous values such as integers, floats, and Booleans. In Julia, unlike many programming languages, the index starts at 1, not 0.

One-dimensional arrays are also termed vectors and two-dimensional arrays as matrices.

Let's define the following vector:

```
julia> A = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,  
610];  
julia> typeof(A)  
Vector{Int64} (alias for Array{Int64, 1})
```

This represents a column array, whereas not using the comma as an element separator creates a row matrix:

```
julia> A = [1 1 2 3 5 8 13 21 34 55 89 144 233 377 610];  
julia> typeof(A)  
Matrix{Int64} (alias for Array{Int64, 2})
```

We observed these are the first 15 numbers of the well-known Fibonacci sequence.

In conjunction with loops in the Asian option example in the previous chapter, we meet the definition of a range as `start : [step] : end`:

```
julia> A = 1:10; typeof(A)  
UnitRange{Int64}  
julia> B = 1:3:15; typeof(B)  
StepRange{Int64, Int64}  
julia> C = 0.0:0.2:1.0; typeof(C)  
StepRangeLen{Float64, Base.TwicePrecision{Float64},  
Base.TwicePrecision{Float64}, Int64}
```

In Julia, the preceding definition returns a `range` type.

To convert a range to an array, we have seen previously that it is possible to use the `collect()` function, as follows:

```
julia> C = 0.0:0.2:1.0; collect(C)
6-element Vector{Float64}:
 0.0
 0.2
 0.4
 0.6
 0.8
 1.0
```

Julia also provides functions such as `zeros()`, `ones()`, and `rand()`, which provide array results.

Normally, these functions return floating-point values, so a little bit of TLC is needed to provide integer results, as seen in the following code snippet:

```
A = convert.(Int64,zeros(15));
B = convert.(Int64,ones(15));
C = convert.(Int64,rand(1:100,15));
```

The preceding code is an example of broadcasting in Julia, and we will discuss it a little further in the next section.

## Broadcasting and list comprehensions

Originally, the application of operators to members of an array was implemented using a `broadcast()` function. This led to some pretty unwieldy expressions, so this was simplified by the preceding “dot” notation.

Let's define a 2x3 matrix of rational numbers and convert them to floats, outputting the result to 4 significant places:

```
julia> X = convert.(Float64, [11/17 2//9 3//7; 4//13 5//11 6//23])
2x3 Matrix{Float64}:
 0.647059  0.222222  0.428571
 0.307692  0.454545  0.26087

julia> round.(X, digits=4)
2x3 Matrix{Float64}:
 0.6471  0.2222  0.4286
 0.3077  0.4545  0.2609
```

Note that the second statement does not alter the actual precision of the values in `X` unless we reassign—that is, `X = round.(X, digits=4)`.

Consider the function we plotted in *Chapter 1* to demonstrate the use of the `UnicodePlots` package:

```
julia> f(x) = x*sin(3.0x)*exp(-0.03x)
```

This does not work when applied to the matrix, as we can see here:

```
julia> Y = f(X)
ERROR: Dimension Mismatch: matrix is not square: dimensions are (2, 3)
```

But it can be evaluated by broadcasting; in this case, the broadcasting dot follows the function names, and also note that broadcasting can be applied to a function defined by *ourselves*, not just to built-in functions and operators:

```
julia> Y = f.(X)
2x3 Matrix{Float64}:
 0.591586  0.136502  0.40602
 0.243118  0.438802  0.182513
```

This can also be done without the `f()` temporary function:

```
julia> Y = X .* sin.(3.0 .* X) .* exp.(- 0.03 .* X)
2x3 Matrix{Float64}:
 0.591586  0.136502  0.40602
 0.243118  0.438802  0.182513
```

Finally, in the following example, we are using the `|>` operator we met previously and an anonymous function:

```
julia> X |> (x -> x .* sin.(3.0 .* x) .* exp.(- 0.03 .* x))
2x3 Matrix{Float64}:
 0.591586  0.136502  0.40602
 0.243118  0.438802  0.182513
```

This introduces the alternate style (`x -> f(x)`) as a mapping function, equivalent to the syntax to map (`f,X`).

Another method of creating and populating an array is by using a list comprehension:

```
# Using a list comprehension is a bit more cumbersome
julia> Y = zeros(2,3);
julia> [Y[i,j] =
           X[i,j]*sin(3.0*X[i,j])*exp(-0.03*X[i,j]) for i=1:2 for j=1:3];

julia> Y
2x3 Matrix{Float64}:
 0.591586  0.136502  0.40602
 0.243118  0.438802  0.182513
```

There are cases where a list comprehension is useful—for example, to list only odd values of the Fibonacci series, we can use the following statement:

```
julia> [fac(k) for k in 1:9 if k%2 != 0]
5-element Vector{BigInt}:
 1
 6
 120
 5040
 362880
```

For the moment, armed with the use of arrays, we will look at recursion and how this is implemented in Julia.

## Computing recursive functions

We considered previously the factorial function, which was an example of a function that used recursion—that is, it called itself. Recursive definitions need to provide a way to exit from the function. Intermediate values are pushed on the stack, and on exiting, the function unwinds, which has the side effect that a function can run out of memory, and so is not always the best (or quickest) method of implementation.

An example in the previous section where this is the case is computing values in the Fibonacci sequence, and we explicitly enumerate the first 15 values. Let's look at this in a bit more detail:

- The series has been identified as early 200 BCE by Indian mathematician Pingala.
- More recently, in Europe around 1200, Leonardo of Pisa (aka Fibonacci) posed the problem of an idealized rabbit population, where a newly born breeding pair of rabbits are put out together and each breeding pair mates at the age of 1 month. At the end of the second month, they produce another pair of rabbits, and the rabbits never die. Fibonacci considered the following question: *How many pairs will there be in 1 year?*
- In nature, the nautilus shell chambers adhere to the Fibonacci sequence's logarithmic spiral, and this famous pattern also shows up in many areas, such as flower petals, pinecones, hurricanes, and spiral galaxies.

We noted that the sequence can be defined by the recurrence relation, as follows:

```
julia> A = Array{Int64}(undef,15);
julia> A[1]=1; A[2]=1;
julia> [A[i] = A[i-1] + A[i-2] for i = 3:length(A)];
```

This presents a similar problem to the factorial in as much as eventually, the value of the Fibonacci sequence will overflow.

To code this in Julia is straightforward:

```
function fib(n::Integer)
    @assert n >= 1
    return (n == 1 || n == 2 ? 1 : (fib(n-1) + fib(n-2)));
end
```

So, the answer to Fibonacci's problem is `fib(12)`, which is 144.

A more immediate problem is with the recurrence relation itself, which involves *two* previous terms, and the execution speed will get rapidly (as  $2^n$ ) longer.

My Mac Pro (Intel i7 processor with 16 GB RAM) runs out of steam around the value 50:

```
julia> @time fib(50);
75.447579 seconds
```

To avoid the recursion relation, a better version is to store all the intermediate values (up to n) in an array, like so:

```
function fib1(n::Integer)
    @assert n > 0
    a = Array{typeof(n),1}(undef,n)
    a[1] = 1
    a[2] = 1
    for i = 3:n
        a[i] = a[i-1] + a[i-2]
    end
    return a[n]
end
```

Using the `big()` function avoids overflow problems and long runtimes, so let's try a larger number:

```
julia> @time(fib1(big(101)))
0.053447 seconds (115.25 k allocations: 2.241 MiB)
573147844013817084101
```

A still better version is to scrap the array itself, which reduces the storage requirements a little, although there is little difference in execution times:

```
function fib2(n::Integer)
    @assert n > 0
    (a, b) = (big(0), big(1))
```

```

while n > 0
    (a, b) = (b, a+b)
    n -= 1
end
return a
end
julia> @time(fib2(big(101)))
0.011516 seconds (31.83 k allocations: 760.443 KiB)
573147844013817084101

```

Observe that we need to be careful about our function definition when using list comprehensions or applying the `|>` operator.

Consider the following two definitions of the Fibonacci sequence we gave previously:

```

julia> [fib1(k) for k in 1:2:9 if k%2 != 0]
ERROR: BoundsError:attempt to access 1-element Vector{Int64} at index
[2]

```

```

julia> [fib2(k) for k in 1:2:9 if k%2 != 0]
5-element Vector{BigInt}:
 1
 2
 5
13
34

```

The first version, which uses an array, raises a bounds error when trying to compute the first term, `fib1(1)`, whereas the second executes successfully.

## Implicit and explicit typing

In the definitions and the factorial function and Fibonacci sequence, the type of the input parameter was explicitly given (as an integer), which allowed Julia to raise that an error is real, complex, and so on, and was passed. This allowed us to check for positivity using the `@asset` macro.

The question arises: *Can the return type of a function be specified as well?* The answer is yes.

Consider the following code, which computes the square of an integer. The return value is a real number (viz. `Float64`) where normally, we would have expected an integer; we term this process as *promotion*, which we will discuss in more detail later in the book:

```

julia> sqi(k::Integer)::Float64 = k*k
sqi (generic function with 1 method)

```

```
julia> sqi(3)
9.0
```

In the next example, the input value is taken as a real number but the return is an integer:

```
julia> sqf(x::Float64)::Int64 = x*x
sqf (generic function with 1 method)
```

This works when the input can be converted *exactly* to an integer but raises an `InexactError` error otherwise:

```
julia> sqf(2.0)
4
julia> sqf(2.3)
ERROR: InexactError: Int64(5.289999999999999)
```

Alternatively, let's consider explicitly specifying the type of a variable.

When using implicit typing, the variable can be reassigned and its type changes appropriately:

```
julia> x = 2; typeof(x)
Int64
julia> x = 2.3; typeof(x)
Float64
julia> x = "Hi"; typeof(x)
String
```

Now, if we try to explicitly define the type of the existing variable, it raises an error:

```
julia> x::Int64 = 2; typeof(x)
ERROR: cannot set type for global x. It already has a value or is
already set to a different type.
```

So, let's start with a new as yet undefined variable:

```
julia> y::Int64 = 2; # => 4
julia> typeof(y)
Int64
```

In this case, assigning the input to a non-integer results in an `InexactError` error, as before:

```
julia> y = 2.3
ERROR: InexactError: Int64(2.3)
```

Also, we cannot redefine the type of the variable now it has been defined:

```
julia> y::Float64 = 2.3; typeof(y)
ERROR: cannot set type for global y. It already has a value or is
already set to a different type.
```

Finally, suppose that we prefix the assignment with the `local` epithet; this seems to be OK except that the variable type is unchanged and its value rounded down rather than an error being raised:

```
julia> local y::Float64 = 2.3;
julia> typeof(y)
Int64
julia> y
2
```

The value of the `y` global is not changed since we are not introducing a new block, and so the scope remains the same.

So far, we have been discussing arrays consisting of a single index (aka one-dimensional), which are equivalent to vectors. In fact, only column-wise arrays are considered to be vectors—that is, consisting of a single column and multiple rows. Here's an example:

```
julia> [1; 2; 3]
3-element Vector{Int64}:
 1
 2
 3
```

Alternatively, an array comprising a single row and multiple columns is viewed as a two-dimensional array, which is commonly referred to as a matrix. We will turn to operating on matrices next:

```
julia> [1 2 3]
1x3 Matrix{Int64}:
 1  2  3
```

Note that a vector is created by separating individual items using a semicolon, whereas the `1x3` matrix is constructed only as space(s). This convention is used in creating multirow and column arrays.

## Simple matrix operations

We will be meeting matrices and matrix operations throughout this book, but let us look now at the simplest of operations.

Let's take A and B, as defined in the following code snippet:

```
julia> A = [1 2 3; 4 5 6];
julia> B = [1 5; 4 3; 2 6];
```

The normal matrix rules apply, which is a feature of multiple dispatch; we will cover this in *Chapter 4*.

The transpose of B can be computed as follows:

```
julia> C = transpose(B)
2x3 transpose(::Matrix{Int64}) with eltype Int64
1 4 2
5 3 6
```

This can also be written more compactly as  $C = B'$ :

```
julia> A + C
2x3 Matrix{Int64}:
2 6 5
9 8 12
julia> A*B
2x2 Matrix{Int64}:
15 29
36 71
```

Matrix division makes more sense with square matrices, but it is possible to define the operations for non-square matrices too. Note here that the / and \ operations produce results of different sizes:

```
julia> A / C
2x2 Matrix{Float64}
0.332273 0.27663
0.732909 0.710652
julia> A \ C
3x3 Matrix{Float64}:
1.27778 -2.44444 0.777778
0.444444 -0.111111 0.444444
-0.388889 2.22222 0.111111
```

The type of the array was previously defined as `Array{Int64, 2}` rather than the now more compact form of `Matrix{Int64}`, and ditto `Array{Float64, 2}` has been replaced with `Matrix{Float64}`.

We will discuss matrix decomposition in more detail later when looking at linear algebra.

Although  $A * C$  is not allowed because the number of columns of  $A$  is not equal to the number of rows of  $C$ , the following broadcasts are all valid:

```
julia> A .* C 2x3 Matrix{Int64}: 1  8  6 20 15 36  
  
julia> A ./ C 2x3 Matrix{Float64}: 1.0 0.5      1.5 0.8 1.66667 1.0  
  
julia> A .== C 2x3 BitMatrix 1  0  0 0  0  1
```

So far, we have only been looking at manipulating variables representing arithmetic values. Julia has a variety of string types, which we will look at next.

## Characters and strings

The simplest character-based variables consist of ASCII and Unicode characters.

A single character is delimited by single quotes, whereas a string uses double quotes or, in some cases, triple-double quotes ("'"), which is discussed in this section.

A string can be viewed as a one-dimensional array of characters and can be indexed and manipulated in a similar fashion as an array of numeric values:

```
julia> s = "Hi there, Blue Eyes!"  
"Hi there, Blue Eyes!"  
julia> length(s)  
20  
  
julia> s[11]  
'B': ASCII/Unicode U+0042 (category Lu: Letter, uppercase)  
julia> s[end]  
'!': ASCII/Unicode U+0021 (category Po: Punctuation, other)
```

*Hint*—Try evaluating the following list comprehension: `[s[i] for i = length(s):-1:1]`.

## Characters

Observe that Julia has a built-in `Char` type to represent a character.

A character occupies 32 bits, *not* 8, which is why it can hold a Unicode character. Have a look at the following example:

```
# All the following represent the ASCII character capital-A
julia> c = 'A';
julia> c = Char(65);
julia> c = '\U0041'
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
```

Julia supports Unicode code, as we see here:

```
julia> c = '\Uc041'
': Unicode U+c041 (category Lo: Letter, other)
```

As such, we can output characters from a variety of different alphabets—for example, Chinese:

```
julia> '\U7537'
'男': Unicode U+7537 (category Lo: Letter, other)
```

It is possible to specify a character code of '`\Uffff`' but `char` conversion does not check that every value is valid. However, Julia provides an `isValid()` function that can be applied to characters:

```
julia> c = '\Udff3'; isValid(c)
false
```

Julia uses the special C-like syntax for certain ASCII control characters such as '`\b`', '`\t`', '`\n`', '`\r`', and '`\f`' for backspace, tab, newline, carriage-return, and form-feed, respectively.

The backslash acts as an escape character, so `Int ('\s')` => 115, whereas `Int ('\t')` => 9.

If more than one character is supplied between the single quotes, this raises an error:

```
julia> 'Hello'
ERROR: syntax: character literal contains multiple characters
```

## Strings

The type of string we are most familiar with comprises a list of ASCII characters that, as we have observed, are normally delimited with double quotes, as in the following example:

```
julia> s = "Hello there, Blue Eyes";
julia> typeof(s)
String
```

The following points are worth noting:

- The built-in concrete type used for strings (and string literals) is `String`
- This supports the full range of Unicode characters via UTF-8 encoding
- All string types are subtypes of the `AbstractString` abstract type, so when defining a function expecting a string argument, you should declare the type as `AbstractString` in order to accept any string type

A `transcode()` function can be used to convert to/from other Unicode encodings:

```
julia> s = "αβγ";
julia> transcode(UInt16, s)
3-element Vector{UInt16}:
 0x03b1
 0x03b2
 0x03b3
```

In Julia (as in Java), strings are immutable—that is, the value of a `String` object cannot be changed. To construct a different string value, you construct a new string from parts of other strings. Let's look at this in more detail:

- ASCII strings are indexable, so from `s` as defined previously: `s[14:17] # => "Blue".`
- The values in the range are inclusive, and if we wish, we can change the increment to `s[14:2:17] => "Bu"` or reverse the slice to `s[17:-1:14] => "eulB".`
- Omitting the end of the range is equivalent to running to the end of the string: `s[14:] => "Blue Eyes".`
- However, `s[:14]` is somewhat unexpected and gives the character '`B`', not the string up to and including `B`. This is because `:` defines a “symbol”, and for a literal, `:14` is equivalent to `14`, so `s[:14]` is the same as `s[14]` and not `s[1:14]`.
- The final character in a string can be indexed using the notation `end`, so in this case, `s[end]` is equal to the '`s`' character.

Strings allow for special characters such as `\n`, `\t`, and so on.

If we wish to include the double quotes, we can escape them, but Julia provides a `"""` delimiter.

So, `s = "This is the double quote \" character"` and `s = """This is the double quote " character""` are equivalent:

```
julia> s = "This is a double quote \" character."; println(s);
This is a double quote " character.
```

Strings also provide the “\$” convention when displaying the value of a variable:

```
julia> age = 21; s = "I've been $age for many years now!"  
I've been 21 for many years now!
```

Concatenation of strings can be done using the \$ convention, but Julia also uses the ‘\*’ operator (rather than ‘+’ or some other symbol):

```
julia> s = "Who are you?";  
julia> t = " said the Caterpillar."  
julia> s*t or "$s$t" # => "Who are you? said the Caterpillar."
```

#### Note

Here’s how a Unicode string can be formed by concatenating a series of characters:

```
julia> '\U7537'*'\U4EBA'  
“男人”
```

## Regular expressions

**Regular expressions (regexes)** came to prominence with their inclusion in Perl programming.

There is an old Perl programmer’s adage: *“I had a problem and decided to solve it using regular expressions; now, I have two problems.”*

Regexes are used for pattern matching; numerous books have been written on them, and support is available in a variety of programming languages post-Perl, notably Java and Python. Julia supports regexes via a special form of string prefixed with r.

Suppose we define an empat pattern as follows:

```
julia> empat = r"^\S+@\S+\.\S+$"  
julia> typeof(empat)  
Regex
```

The following example will give a clue to what the pattern is associated with:

```
julia> occursin(empat, "fred.flintstone@bedrock.net")  
true  
julia> occursin(empat, "Fredrick Flintstone@bedrock.net")  
false
```

The pattern is for a valid (simple) email address, and in the second case, the space in Fredrick Flintstone is not valid (because it contains a space!), so the match fails.

Since we may wish to know not only whether a string matches a certain pattern but also how it is matched, Julia has a `match()` function:

```
julia> m = match(r"@bedrock", "barney, rubble@bedrock.net")
RegexMatch(„@bedrock”)
```

If this matches, the function returns a `RegexMatch` object; otherwise, it returns `Nothing`:

```
julia> m.match
"@"
julia> m.offset
14
julia> m.captures
0-element Array{Union{Nothing, SubString{String}}}, 1}
```

A detailed discussion of regexes is beyond the scope of this book.

The following link provides a good online source for all things regex, including an excellent cheat sheet via the Quick Reference page: <https://www.rexegg.com>.

In addition, there are a number of books on the subject, and a free PDF can be downloaded from the following link:

[https://www.academia.edu/22080976/Regular\\_expressions\\_cookbook\\_2nd\\_edition](https://www.academia.edu/22080976/Regular_expressions_cookbook_2nd_edition).

### ***Version strings***

Version numbers can be expressed with non-standard string literals as `v“...”`.

These literals create `VersionNumber` objects that follow the specifications of “semantic versioning” and therefore are composed of major, minor, and patch numeric values, followed by pre-release and build alpha-numeric annotations.

So, a full specification typically would be “v1.9.1-rc1”, where the major version is “1”, minor version “9”, patch level “1”, and release candidate “1”.

Currently, only the major version needs to be provided, and the others will assume default values; for example, “v1” is equivalent to “v1.0.0”.

(The release candidate has no default, so needs to be explicitly defined.)

### **Byte array literals**

Another special form is the `b“...”` byte array literal, which permits string notation to express arrays of `UInt8` values.

These are the rules for byte array literals:

- ASCII characters and ASCII escape sequences produce a single byte
- `\x` and octal escape sequences produce a byte corresponding to the escape value
- Unicode escape sequences produce a sequence of bytes encoding that code points in UTF-8

Consider the following two examples:

```
julia> A = b"HEX:\xefcc"
7-element Base.CodeUnits{UInt8,String}:
[0x48, 0x45, 0x58, 0x3a, 0xef, 0x63, 0x63]

julia> B = b"\u2200 x \u2203 y"
11-element Base.CodeUnits{UInt8,String}:
0xe2
0x88
0x80
0x20
0x78
0x20
0xe2
0x88
0x83
0x20
0x79
```

Here, the first three elements represent the `\u2200` code, then `0x20,0x78,0x20` correspond to `<space>x<space>`, followed by three more elements for the `\u2203` code, and finally, `0x20, 0x79`, which represents `<space>y`.

## Complex and rational numbers

We have met the syntax for rational numbers in the previous chapter, and we will review operations on them here. Also, we will introduce another arithmetic type: complex numbers.

### Complex numbers

There are two ways to define a complex number in Julia—first, using the `Complex` type definition as its associated `Complex()` constructor:

```
# Note the difference in these two definitions
julia> c = Complex(1, 2); typeof(c)
Complex{Int64}
```

```
julia> c = Complex(1, 2.0); typeof(c) Complex{Float64}

julia> c = ComplexF32(1,2.0); typeof(c) Complex{Float32}
```

Because in the second example, the complex number consists of an ordered pair of two reals, its size is 128 bits, whereas `ComplexF32` has 2x `Float32` arguments and `ComplexF16` will have 2x `Float16` arguments.

The `Complex(0.0, 1.0)` number corresponds to the imaginary number '`I`'—that is, `sqrt(-1.0)`—but Julia uses the '`im`' symbol rather than '`I`' to avoid confusion with an `I` variable, frequently used as an index iterator.

Hence, `Complex(1, 2)` is exactly equivalent to `1 + 2*im`, but normally the '`*`' operator is omitted, and this would be expressed as `1 + 2im`.

The complex number supports all normal arithmetic operations, as illustrated here:

```
julia> c = 1 + 2im;
julia> d = 3 + 4im;
julia> c*d
-5 + 10im
julia> c/d
0.44 + 0.08im
julia> c\d
2.2 - 0.4im
```

The `c/d` and `c\d` divisions produce real arguments even when the components are integers.

This is like Julia's behavior with a simple division of integers. Also, it defines `real()`, `imag()`, `conj()`, `abs()`, and `angle()` complex functions.

`abs` and `angle` can be used to convert complex arguments to polar form:

```
julia> c = 1.0 + 2im; abs(c)
2.23606797749979
julia> angle(c)
1.1071487177940904 # (in radians)
```

Complex versions of many mathematical functions can be applied:

```
julia> c = 1 + 2im;
julia> sin(c)
3.1657 + 1.9596im
```

```
julia> log(c)
0.8047 + 1.10715im
julia> sqrt(c)
1.272 + 0.78615im
```

## Rationals

Julia has a rational number type to represent exact ratios of integers. A rational is defined by the use of the `//` operator—for example, `5//7`. If the numerator and denominator have a common factor, then the number is reduced to its simplest form; for example, `21//35` reduces to `3//5`.

Operations on rationals or on mixed rationals and integers return a rational result:

```
julia> x = 3; y = 5//7;
julia> x*y
15//7
julia> y^2
25//49
julia> y/x
5//21
```

The `numerator()` and `denominator()` functions return the numerator and denominator of a rational, and `float()` can be used to convert a rational to a float:

```
julia> x = 17//100;
julia> numerator(x)
17
julia> denominator(x)
100
julia> float(x) => 0.17
```

Constructing infinite rational values, both positive and negative, is acceptable:

```
julia> 5//0
1//0
julia> -5//0
-1//0
```

Notice that both computations reduce the numerator to 1. It is possible to construct rationals of complex numbers, as in this example:

```
julia> c = (1 + 2im)//(4 + 3im)
2//5 + 1//5*im
```

This output is a little confusing as the actual value is  $(2 + 1\text{im})//5$ , which arises by multiplying the top and bottom values by the complex conjugate of the denominator ( $4 - 3\text{im}$ ).

The `typeof(c)` value is `Complex{Rational{Int64}}`, and as of now, the `numerator()` and `denominator()` functions fail, even though these should return  $(2 + 1\text{im})$  and 5 respectively:

```
julia> numerator(c)
ERROR: MethodError: no method matching
        numerator(::Complex{Rational{Int64}})
```

Closest candidates are:

```
numerator(::Integer) at rational.jl:236
numerator(::Rational) at rational.jl:237
```

## A little light relief

To add a bit of flesh to some of the discussions so far, here are three very different examples, all of which make use of various Julia data structures.

### The Sieve of Eratosthenes

The Sieve of Eratosthenes is an ancient algorithm for finding all prime numbers up to a given limit. As the name suggests, this goes back to the ancient Greeks, around 200 BCE.

The algorithm is quite simple and consists of marking composites (viz. not primes), the multiples of each prime, starting with the first prime number, 2.

First, we need to define a function to determine whether a number is a composite from a list of primes:

```
julia> cop(x, i) = any(j -> i % j == 0, x)
```

Now, let's test it:

```
julia> A = [2 3 5 7 11 13 17 19];
julia> cop(A, 53)
false

julia> cop(A, 54)
true
```

Now, we can construct the function to implement the Sieve of Eratosthenes:

```
function erato(N::Integer)
    @assert N > 0
```

```
P = Int[]
for i in 2:N
    if !cop(P, i)
        push!(P, i)
    end
end
return P
end
```

This function uses an empty integer array and pushes values that are not composite onto it using `push!()` because `push` alters the array:

```
julia> erato(10)
4-element Vector{Int64}:
 2
 3
 5
 7
```

This seems to work, so let us see how long Julia takes to compute the primes up to 1 million and how many primes there are:

```
julia> tm = @elapsed A = erato(1_000_000);
julia> print("Computed $(length(A)) primes in $(round(tm, digits=4))
sec.")
Computed 78498 primes in 12.7348 sec.
```

`@elapsed` macro is like `@time` but returns the elapsed time as a real number in seconds. This has been rounded to produce more compact output. The implementation is hardly the most efficient one that can be constructed. One problem is with the `cop()` routine, as a number in the range 1:N needs only be checked up to a limit of `sqrt(N)` since if one factor is greater than this limit, the other factor must be less.

I'll leave it to you to construct a more efficient algorithm with or without help from ChatGPT.

Alternatively, there is a `Primes.jl` module, which was introduced back in Julia v0.5 and has been largely untouched since then. The approach is much more sophisticated, and the source is well worth a look, even though not all the nuances will be familiar as yet. You can find details at <https://juliamath.github.io/Primes.jl>.

## Bulls and cows

Let us look at some code to play the game *Bulls and Cows*. A computer program `moo`, written in 1970 at MIT in the **Programming Language One (PL/I)** language, was among the first Bulls and Cows

---

computer implementations. It is proven that any number could be solved for up to 7 turns, and the minimal average game length is 5.21 turns.

The computer enumerates a 4-digit random number from the digits 0 to 9, without duplication. The player inputs their guess, and the program should validate the player's guess, reject guesses that are malformed, then print the "score" in terms of the number of bulls and cows according to the following rules:

- One bull is accumulated for each digit in the guess that equals the corresponding digit in the randomly chosen initial number
- One cow is accumulated for each digit in the guess that also appears in the randomly chosen number but in the wrong position
- The player wins if the guess is the same as the randomly chosen number, and the program ends

The program accepts a new guess, incrementing the number of tries:

```
# Coding this up in Julia
function bacs()
    bulls = cows = turns = 0
    a = Any[]
    while length(unique(a)) < 4
        push!(a,rand('0':'9'))
    end
    my_guess = unique(a)
    println("Bulls and Cows")
    while (bulls != 4)
        print("Guess? > ")
        s = chomp(readline(stdin))
        if (s == "q")
            print("My guess was "); [print(my_guess[i]) for i=1:4]
            return
        end
        guess = collect(s)
        k = length(guess)
        if !(k == 4 && all(isdigit,guess) &&
            length(unique(guess)) == k)
            print("\nEnter four distinct digits or q to quit: ")
            continue
        end
        bulls = sum(map(==, guess, my_guess))
        cows = length(intersect(guess,my_guess)) - bulls
        println("$bulls bulls and $cows cows!")
        turns += 1
    end
end
```

```
    end
    println("\nYou guessed my number in $turns turns.")
end
```

One way to run this game is by *including* the `bacs.jl` file and then issuing the `bacs()` command:

```
julia> include("bacs.jl");
julia> bacs()
```

Here is a game played recently:

```
BULLS and COWS
=====
Enter four distinct digits or <return> to quit
Guess> 1234
0 bulls and 1 cows!
Guess> 5678
0 bulls and 1 cows!
Guess> 1590
2 bulls and 0 cows!
Guess> 2690
2 bulls and 0 cows!
Guess> 3790
2 bulls and 0 cows!
Guess> 4890
2 bulls and 2 cows!
Guess> 8490
4 bulls and 0 cows!
You guessed my number in 7 turns.
```

We define an `A` array as `Any []`. This is because although arrays were described as homogeneous collections, Julia provides an `Any` type that can, as the name suggests, store any form of variable—this is similar to the Microsoft variant data type.

The principal features of the code are set out here:

- Integers are created as characters using the `rand()` function and pushed onto `A` with `push!()`
- The `A` array may consist of more than four entries, so a `unique()` function is applied, which reduces it to four by eliminating duplicates, and this is stored in `my_guess`
- User input is via `readline()`, and this will be a string including the trailing `return (\n)`, so a `chomp()` function is applied to remove it, and the input is compared with '`q`' to allow an escape before the number is guessed

- A `collect()` function is applied to return a four-element array of type `Char`, and it checks that there are four elements and that these are all digits.

The number of bulls is determined by comparing each entry in `guess` and `my_guess`; this is achieved by using a `map()` function to apply '`==`' , 4 bulls, and we are done. Otherwise, it's possible to construct a new array as the intersection of `guess` and `bacs_number`, which will contain all elements that match. So, subtracting the number of bulls leaves the number of cows.

## Julia sets

The Julia documentation provides an example of generating a Mandelbrot set; instead, we will provide code to create a Julia set.

This is named after Gaston Julia and is a generalization of the Mandelbrot set. Computing a Julia set requires the use of complex numbers.

Both the Mandelbrot set and the Julia set (for a given constant  $z_0$ ) are the sets of all instances of  $z$  (complex number) for which the  $z = z^*z + z_0$  iteration does not diverge to infinity. The Mandelbrot set is those  $z_0$  constants to which the Julia set is connected.

We create a `jset.jl` file, and its contents define the function to generate a Julia set:

```
function juliaset(z, z0, nmax::Int64)
    for n = 1:nmax
        if abs(z) > 2 (return n-1) end
        z = z^2 + z0
    end
    return nmax
end
```

Here,  $z$  and  $z_0$  are complex values, and  $nmax$  is the number of trials to make before returning. If the modulus of the complex number  $z$  gets above 2, then it can be shown that it will increase without limit.

The function returns the number of iterations until the modulus test succeeds, or else  $nmax$ .

Also, we will write a second file, `pgmfile.jl`, to handle displaying the Julia set:

```
function create_pgmfile(img, outf::String)
    s = open(outf, "w")
    write(s, "P5\n")
    n, m = size(img)
    write(s, "$m $n 255\n")
    for i=1:n, j=1:m
        p = img[i,j]
        write(s, uint8(p))
    end
```

```
    close(s)
end
```

Although we will not be looking in any depth at graphics later in the book, it is quite easy to create a simple disk file using the portable bitmap (`netpbm`) format. This consists of “magic” numbers P1 - P6, followed on the next line by the image height, width, and a maximum color value, which must be greater than 0 and less than 65536; all of these are ASCII values, not binary values.

Then follows the image values (height x width), which may be ASCII for P1, P2, and P3 or binary for P4, P5, and P6. There are three different types of portable bitmap; B/W (P1/P4), grayscale (P2/P5), and color (P3/P6).

The `create_pgm()` function creates a binary grayscale file (magic number = P5) from an image matrix where the values are written as `UInt8`. Notice that the `for` loop defines the `i, j` indices in a single statement with correspondingly only one `end` statement. The image matrix is output in column order, which matches the way it is stored in Julia.

The main program, `jmain.jl`, looks like this:

```
include("jset.jl")
include("pgmfile.jl")
h = 400; w = 800;
m = Array{Union{Nothing, Int}}(nothing, h, w);
c0 = -0.8+0.16im;

pgm_name = "julia.pgm";
t0 = time();
for y=1:h, x=1:w
    c = complex((x-w/2)/(w/2), (y-h/2)/(w/2))
    m[y,x] = juliaset(c, c0, 256)
end
t1 = time();
create_pgmfile(m, pgm_name);
print("Written $pgm_name\nFinished in $(round((t1-t0), digits = 4))\nseconds.\n")
```

This assumes the two `include` files are in the same directory as the program file listed previously, and then the PGM file will be written in the same place:

```
$> julia print("Written $pgm_name\nFinished in $eps seconds.\n")
Written julia.pgm
Finished in 0.3894 seconds.
```

The following points are worthy of note:

- We define a matrix `N` of type `UInt64` to hold the return values from the `juliaset` function
- The `c0` constant is arbitrary; different values of `c0` will produce different Julia sets, and the starting value for `c0 = 0.0+0.0im` corresponds to the standard Mandelbrot set
- The starting complex number is constructed from the `(x,y)` coordinates and scaled to the half-width and height
- The *magic* number for this type of PGM file is `P5`, which is hardcoded in the `create_pgmfile()` routine
- We have “cheated” a little by defining the maximum number of iterations as `256`

Because we are writing byte values (`UInt8`) and the values that remain bounded will be `256`, we subtract 1 from this value to ensure values are in the range `[0,255]`, so do not overflow.

After running the `jmain.jl` file from the **read-eval-print loop (REPL)** (or in VS Code), the output to disk should look like that shown in *Figure 2.2*:

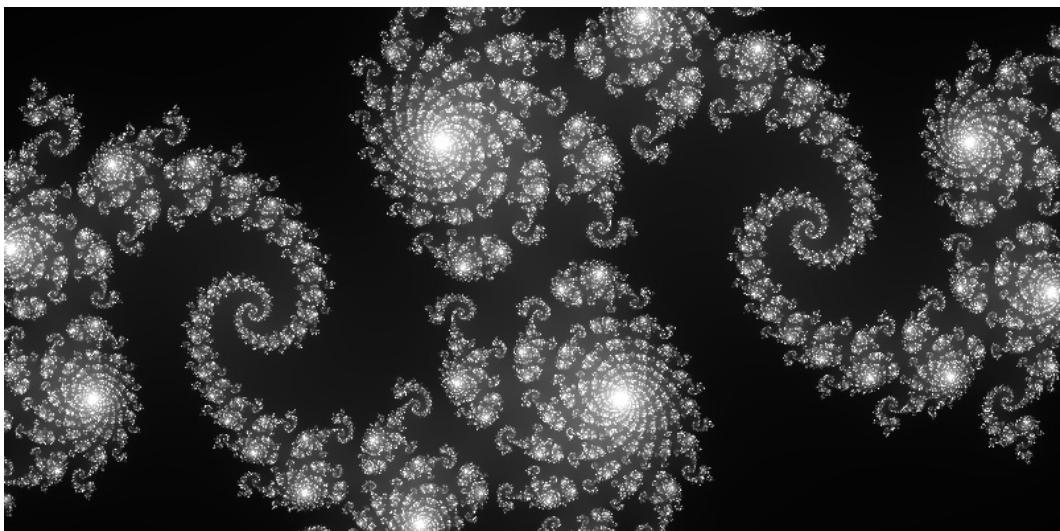


Figure 2.2 – The Julia set generated by the preceding code

After that light relief, it is time to conclude this chapter by introducing a few additional data structures, and we will begin by considering arrays of more than two dimensions—that is, neither vectors nor matrices.

## Multi-dimensional arrays

In fact, Julia views *all* arrays as a single stream of values and applies size and reshape parameters to compute the appropriate indexing.

Arrays with the number of dimensions greater than 2 (that is, `array > 2`) can be defined in a straightforward method:

```
julia> A = rand(4,4,4)

4x4x4 Array{Float64,3}:
[:, :, 1] =
0.522564 0.852847 0.452363 0.444234
0.992522 0.450827 0.885484 0.0693068
0.378972 0.365945 0.757072 0.807745
0.383636 0.383711 0.304271 0.389717
[:, :, 2] =
0.570806 0.912306 0.358262 0.494621
0.810382 0.235757 0.926146 0.915814
0.634989 0.196174 0.773742 0.158593
0.700649 0.843975 0.321075 0.306428

[:, :, 3] =
0.638391 0.606747 0.15706 0.241825
0.492206 0.798426 0.86354 0.715799
0.971428 0.200663 0.00568161 0.0868379
0.936388 0.183021 0.0476718 0.917008

[:, :, 4] =
0.252962 0.432026 0.817504 0.274034
0.164883 0.209135 0.925754 0.876917
0.125772 0.998318 0.593097 0.614772
0.865795 0.204839 0.315774 0.520044
```

Note the use of slice '`:`' notation to display the 3D matrix; values are ordered by the third index, then the second, and finally the first.

It is possible to convert this 3D array into a standard matrix containing the same number of values, as follows:

```
julia> B = reshape(A, 8, 8)

8x8 Array{Float64,2}:
0.522564 0.452363 0.570806 ... 0.15706 0.252962 0.817504
0.992522 0.885484 0.810382 ... 0.86354 0.164883 0.925754
```

```
0.378972 0.757072 0.634989 ... 0.005681 0.125772 0.593097
0.383636 0.304271 0.700649 ... 0.0476718 0.865795 0.31577
0.852847 0.444234 0.912306 ... 0.241825 0.432026 0.274034
0.450827 0.0693068 0.235757 ... 0.715799 0.209135 0.876917
0.365945 0.807745 0.196174 ... 0.086838 0.998318 0.614772
0.383711 0.389717 0.843975 ... 0.917008 0.204839 0.520044
```

Or, it could appear as a simple vector, like this:

```
julia> C = reshape(A,64); typeof(C); # => Array{Float64,1}

julia> transpose(C)
1x64 LinearAlgebra.Transpose{Float64,Array{Float64,1}}:
0.522564 0.992522 0.378972 0.383636 ... 0.876917 0.614772 0.520044
```

## Sparse matrices

Normal matrices are sometimes referred to as “dense,” which means that there is an entry for cell [i,j]. In cases where most cell values are, say, 0, this is inefficient, and it is better to implement a scheme of tuples (i,j,x), where x is the value referenced by i and J.

These are termed sparse matrices, and we can create a sparse matrix by executing the following code:

```
using SparseArrays
S1 = SparseArrays.sparse(I, J, X[, m, n, combine])
```

S1 will have dimensions m by n and S[I[k], J[k]] = X[k].

If m and n are not given, they default to `max(I)` and `max(J)` respectively. The `combine()` function is used to combine duplicates, and if not provided, duplicates are added by default.

Sparse matrices support much of the same set of operations as dense matrices, but there are a few special functions that can be applied. For example, `spzeros()` is a counterpart of `zeros()`, and random number arrays can be generated by `sprand()` and `sprandn()`:

```
# The 0.3 means only 30% for the numbers generated will be non-zero
# This will produce different arrays each time it is run

julia> A = sprand(5,5,0.3)
0.21055 0.544431 0.16395
0.76612 0.785714 0.993288
0.740757 0.209118
```

```
# So squaring the matrix produces another sparse matrix

julia> A * A
5×5 SparseMatrixCSC{Float64,Int64} with 10 stored entries:
      0.121447  0.034285
      .           .
      .           0.0345197
      .
      .
      0.601951  0.735785  0.207715  0.617346  0.906046
      0.155966  .         0.403291
```

Using `Matrix()` converts the sparse matrix to a dense one, as follows:

```
julia> convert(Matrix,A);
5×5 Matrix{Float64}:
 0.0      0.0      0.0      0.0      0.16395
 0.21055  0.0      0.544431  0.0      0.0
 0.0      0.0      0.0      0.0      0.0
 0.76612  0.0      0.0      0.785714  0.993288
 0.0      0.740757  0.209118  0.0      0.0
```

## Sparse vectors

Alternatively, we can convert a vector into a sparse array using the `sparsevec()` function:

```
julia> sparsevec([1 7 0 3 0])
5-element SparseVector{Int64, Int64} with 3 stored entries:
 [1] = 1
 [2] = 2
 [4] = 4
```

Another method of construction can make use of a dictionary, as follows:

```
julia> sparsevec(Dict(1 => "Malcolm", 3 => "malcolm@myemail.org"))
3-element SparseVector{String, Int64} with 2 stored entries:
 [1] = "Malcolm"
 [3] = "malcolm@myemail.org"

julia> sparsevec(Dict("name" => "Malcolm", "email" => "malcolm@myemail.org"))
ERROR: MethodError: no method matching sparsevec(::Dict{String, String})
```

*Note: The key must be an integer; otherwise, an error is raised.*

## Sparse diagonal matrices

The `eyes()` function to produce an identity matrix has been deprecated.

Instead, we can use `spdiagm()` to create a sparse diagonal matrix, and then `convert()` is required to convert it to a real matrix:

```
julia> A = spdiagm(ones(Int64,3)) # or spdiagm([1,1,1])
3x3 SparseMatrixCSC{Int64, Int64} with 3 stored entries:
 1 . .
   . 1 .
   . . 1

julia> convert(Matrix{Float64},A)
3x3 Matrix{Float64}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0
```

Arrays consist of a collection of homogeneous elements. Later, in *Chapters 6 and 7*, we will examine more sophisticated structures where the columns can be addressed by name.

These are termed `DataFrames` and can be thought of as equivalent to data held in a spreadsheet, but we will briefly introduce them here.

## Data arrays and data frames

Users of R will be aware of the success of data frames when employed in analyzing datasets, a success that has been mirrored by Python with the `pandas` package.

Julia too adds data frame support through the use of a `DataFrames` package.

The package extends Julia's base by introducing three basic types, as follows:

- `Missing.missing`: An indicator that a data value is missing
- `DataArray`: An extension to the `Array` type that can contain missing values
- `DataFrame`: A data structure for representing tabular datasets

It is such a large topic that we will be looking at data frames in some depth when we consider statistical computing.

However, here's some code to get a flavor of processing data with these packages:

```
julia> using DataFrames
julia> df1 = DataFrame(ID = 1:4,
                      Cost = [10.1,7.9,missing,4.5])
```

4 × 2 DataFrame		
Row	ID	Cost
1	1	10.1
2	2	7.9
3	3	missing
4	4	4.5

Common operations include computing `mean(d)` or `var(d)` of the Cost because of the missing value in row 3:

```
julia> using Statistics  
julia> mean(!, df1[:Cost])  
missing
```

We can create a new data frame by dropping *ALL* rows with missing values, and now statistical functions can be applied as normal:

```
julia> df2 = dropmissing(df1). << This might have changed ??? >>>  
3 × 2 DataFrames.DataFrame  
| Row | ID | Cost |  
|---|---|---|  
| 1 | 1 | 10.1 |  
| 2 | 2 | 7.9 |  
| 3 | 4 | 4.5 |  
  
julia> (μ,σ) = (mean(df2[!, :Cost]), std(df2[!, :Cost]))  
(7.5, 2.8213471959331766)
```

We will cover data frames in much greater detail when we consider data I/O in *Chapter 6*.

At this time, we will look at the `Tables` API, implemented in the `Tables.jl` file, which is used by a large number of packages.

## Dictionaries, sets, stacks, and queues

In addition to arrays, Julia supports associative arrays, sets, and many other data structures. In this section, we will introduce dictionaries, sets, and a couple of others.

### Dictionaries

Associative arrays consist of collections of `key-values` pairs. In Julia, associative arrays are called dictionaries (dicts).

Let us look at a simple data type to hold user credentials: ID, password, email, and so on. We will not include a username as this will be the key to a credential data type. In practice, this would not be a great idea as users often forget their username as well as their password!

This includes a type (`struct`) and some functions that operate on that type, as follows:

```
using Base64
struct UserCreds
    uid::Int
    password::String
    fullname::String
    email::String
    admin::Bool
end

function matchPwd(uc::Dict{String,UserCreds}, uname::String, pwd::String)
    return (uc[uname].password == base64encode(pwd) ? true : false)
end
isAdmin(uc::Dict{String,UserCreds}, fname::String) = uc[fname].admin;
```

We can use this to create an empty authentication array (AA) and add an entry for myself.

For now, we will just use the `base64()` function to scramble the password, although, in practice, a better coding scheme would be used:

```
julia> AA = Dict{String,UserCreds}();
julia> AA["malcolm"] = UserCreds(101,
        base64encode("Pa55word"),
        "Malcolm Sherrington",
        "malcolm@myemail.org", true)

julia> println(matchPwd(AA, "malcolm", "Pa55word") ? "OK" : "No,
sorry")
OK
```

Adding the user requires the scrambling of the password by the user; otherwise, `matchPwd()` will fail.

To overcome this, we can override the `UserCreds()` default constructor by adding an internal constructor inside the type definition—this is an exception to the rule that type definitions can't contain functions, since clearly it does not conflict with the requirement for multiple dispatch.

An alternative way to define the dictionary is by adding some initial values.

The values can be referenced via the key, as follows:

```
julia> me = AA["malcolm"]
UserCreds(101, "UGE1NXdvcmQ=", "Malcolm Sherrington",
          "malcolm@myemail.org", true)
```

The ' .' notation is used to reference the fields:

```
julia> me.fullname
"Malcolm Sherrington"
```

Alternatively, it is possible to iterate over all the keys:

```
julia> for who in keys(AA)
           println(AA[who].fullname)
       end
"Malcolm Sherrington"
```

Attempting to retrieve a value with a key that does not exist, such as AA [ "james" ], will produce an error.

We need to trap this in the module routines such as `matchPwds()` and `isAdmin()` using `try/catch/finally` syntax, like so:

```
# isAdmin function could be rewritten as:
function isAdmin2(uc::Dict{String,UserCreds},uname::String)
    check_admin::Bool = false;
    try
        check_admin = uc[uname].admin
    catch
        check_admin = false
    finally
        return check_admin
    end
end

julia> isAdmin(AA,"james")
ERROR: KeyError: key "james" not found
julia> isAdmin2(AA,"james")
false
```

## Sets

A set is a collection of distinct unordered objects.

The basic constructor creates a set with elements of type Any; supplying arguments will determine (*restrict*) the set type:

```
julia> S0 = Set()  
Set{Any}()
```

Alternatively, we can create a set of specific types of elements by supplying a list, like so:

```
julia> S1 = Set([1,2,3,1])  
Set([2, 3, 1])  
julia> typeof(S1)  
Set{Int64}  
julia> S2 = Set([2,4,6])  
Set([4, 2, 6])
```

The “usual” functions of union and intersection can be applied to S1 and S2, as follows:

```
julia> S3 = union(S1, S2)  
Set([4, 2, 3, 6, 1])  
julia> S4 = intersect(S1, S2)  
Set([2])
```

We can check whether one set is a subset of a second by executing the following code:

```
julia> issubset(S3, S4)  
false  
julia> issubset(S4, S3)  
true
```

Elements can be added to a set using the push! () function.

Recall that ! implies that the data structure is altered, even though it is constructed as immutable:

```
# This works  
julia> push!(S0, "Malcolm")  
Set{Any}(["Malcolm"])
```

```
# But this does NOT
julia> push!(S1, "Malcolm")
ERROR: MethodError: Cannot `convert` an object of type String to an
object of type Int64
```

It is possible to push mixed data types onto the S0 set, as this was defined as the Any type:

```
julia> push!(S0, 21)
Set{Any} with 2 elements:
"Malcolm"
21
```

Because the set has no duplicate items, repeated ones are removed, and notice the order in the set is not the same as that in the list:

```
julia> S4 = Set([1, 1, 2, 3, 3, 5, 8])
Set{Int64} with 5 elements:
5
2
8
3
1

julia> pop!(S4)
5
```

The `pop()!` function works on a `Set` but the order in which items are returned is random, corresponding to the arbitrary order created when the set was created.

## Stacks and queues

The `DataStructures` package implements a rich bag of data structures, including deques, queues, stacks, heaps, ordered sets, linked lists, digital trees, and so on.

For a full discussion of *ALL* of these, see the following URL: <https://github.com/JuliaCollections>.

As an illustration, let's look at the stack and deque data structures.

This is a double-ended queues that allows the insertion and removal of elements at both ends of a sequence.

The `Stack` and `Queue` types are based on the `Deque` type and provide interfaces for **first in, last out (FILO)** and **first in, first out (FIFO)** access respectively. Deques expose `push!()`, `pop!()`, `shift!()`, and `unshift!()` functions.

---

Consider the following simple example to illustrate using stacks and queues:

```
julia> using DataStructures
julia> S = Stack{Char}(100); typeof(S)
Stack{Char}
julia> Q = Queue{Char}(100); typeof(Q)
Queue{Char}
```

A stack will use `push!()` and `pop!()` to add and retrieve data, while a queue will use `shift!()` and `unshift!()`.

Queues also encapsulate the latter two processes as `enqueue!()` and `dequeue!()`.

Stacks are FILOs, while queues are FIFOs, as the following code snippet demonstrates:

```
julia> greet = "Here's looking at you kid!";
julia> for i = 1:lastindex(greet)
        push!(S,greet[i])
        enqueue!(Q,greet[i])
    end

julia> for i = 1:lastindex(greet) print(pop!(S)) end
!dik uoy ta gnikool s'ereH

julia> for i = 1:lastindex(greet) print(dequeue!(Q)) end
Here's looking at you kid!
```

## Summary

In this chapter, we started having a more in-depth look at Julia, with a more detailed discussion of various scalar, vector, and matrix data types comprising integer, real numbers, characters, and strings, as well as the operations acting on them.

We then moved on to data types such as rational numbers, big integers, floats, and complex numbers.

We also looked at arithmetic functions, comparing the use of recursive and non-recursive definitions.

Finally, we looked at some complex data structures such as data arrays and data frames, dictionaries and sets, and stacks and queues.

The next chapter follows on by expanding our survey of Julia functions to accommodate passing variable arguments and then considering the type system in greater detail, defining composite data structures, implicit and explicit variable assignment, and the use of parametrization.



# 3

## The Julia Type System

In this chapter and the next two, we will discuss the features that make Julia appealing to data scientists and scientific programmers.

Julia was conceived to meet the frustrations of the principal developers with existing programming languages. It is well designed and beautifully written. Moreover, much of the code is written in Julia so is available to be inspected and changed. Although we do not advocate modifying much of the base code (also known as the standard library), it is there to look at and learn from.

Much of this book is aimed at the analyst with some programming skills and the jobbing programmer, so we will postpone the guts of the Julia system until the last chapter, when we consider package development and contributing to the Julia community.

We will begin by continuing our discussion on functions, followed by a more in-depth discussion about the Julia type system.

This chapter will cover the following:

- Using optional parameters in functions
- Variable-length argument lists
- Currying and closures
- The built-in Julia type for rational numbers
- Creating a composite type
- Type aliases and unions
- Defining high-dimensional vectors

## More about functions

We have met functions in previous chapters defined as a `function () ... end` block and shown that there is a convenient one-line syntax for the simplest of cases:

```
# rsq(x) = 1/(x*x) is exactly equivalent to:
function rsq(x)
    y = 1/(x*x)
    return y
end
```

The `y` variable is not needed (of course). It is local to the `rsq()` function and has no existence outside the function call, and the last statement could be written as `return 1/(x*x)` or even just as `1/(x*x)`, since functions in Julia return their last value.

## The do syntax

In the previous chapter, we looked at ways of performing operations using broadcasting as an alternative to conventional `for ... end` loops and/or list comprehensions. In order to work more compactly, it is often useful to use a different construct using a `do ... end` block, which we will introduce here:

```
julia> map([1,2,3,4]) do x
           rsq(x)
       end
4-element Vector{Float64}:
 1.0
 0.25
 0.1111111111111111
 0.0625
```

### Note

You can think of this as equivalent to `map(rsq, [1,2,3,4])`.

Also, we can use the `do` syntax combined with a function that operates on a set of array elements:

```
julia> sum(collect(1:10^6)) do x
           rsq(x)
       end
1.6449330668487272
```

This result we have seen before as Euler's solution to the sum of the series of  $1/x^2$ , which converges to  $\pi^2/6$ .

## First-class objects

Functions are first-class objects in Julia. This allows them to be assigned to other identifiers, passed as arguments to other functions, returned as the value from other functions, stored as collections, and applied (*mapped*) to a set of values at runtime.

The argument list consists of a set of dummy variables, and the data structure using the () notation is called a tuple. By default, the arguments are of type {Any}, but explicit argument types can be specified, which aids the compiler in assigning memory and optimizing the generated code.

So, `sq(x)` would work with any data structures where the \* operator is defined, whereas a definition of the form `*sq(x::Integer) = x*x` would only work for integers.

Surprisingly, perhaps, `sq()` does work for strings since the \* operator is used for string concatenation rather than +:

```
sq(x) = x*x; sq("Hello") ; # => HelloHello
```

This is an example of Julia's overloading of functions using **multiple dispatch**, which is discussed in more detail in the next chapter.

It is possible to overload the + operator for strings, but since it is part of `Base`, it is necessary to import it first:

```
julia> "Hello"+" World"
ERROR: MethodError: no method matching +(::String, ::String)
String concatenation is performed with * (See also: https://docs.julialang.org/en/v1/manual/strings/#man-concatenation).
Closest candidates are:
+(::Any, ::Any, ::Any, ::Any...)
@ Base operators.jl:578
import Base:+
+(s1::String,s2::String) = s1*s2; # or else string(s1,s2)
julia> "Hello"+" World"
"Hello World"
```

To apply a function to a list of values, we can use the `map()` construct.

We are going to modify `sq()` slightly so that it can broadcast over a more general type of data structure:

```
julia> sq(x) = x.*x
julia> map(sq, Any[1, 2.0, [1,2,3], 7//5, "Hi"])
4-element Array{Any,1}:
1
4.0
[1, 4, 9]
```

```
49//25
"HiHi"
```

This definition of `sq()` will work with scalars too, and we can use the `split()` function to turn strings into character arrays.

Notice the difference in the following constructs:

```
julia> map(sq,split("HI"));
julia> map(sq,split("H I"))
2-element Array{String,1}:
"HH"
"II"
julia> a = split("H E L L O")
5-element Array{SubString{String},1}:
"H"
"E"
"L"
"L"
"O"
julia> b = split("W O R L D")
5-element Vector{SubString{String}}:
"W"
"O"
"R"
"L"
"D"
julia> import Base.+
julia> +(s1,s2) = string(s1,s2)
+ (generic function with 176 methods)
julia> a.+b
5-element Array{String,1}:
"HW"
"EO"
"LR"
"LL"
"OD"
```

We can list the methods of a function by using `methods()`, which takes as its argument a function name. In Julia, there is no difference between built-in and user-defined functions (other than the requirement to import from `Base`), so our overloaded method for adding strings is tacked on the end of `list`.<sup>Ω</sup>:

```
julia> methods(+)
# 209 methods for generic function "+" from Base:
```

```
[1] +(x::T, y::T) where T<:Union{Int128, Int16, Int32, Int64, Int8,  
Uint128, Uint16, Uint32, Uint64, Uint8}  
    @ int.jl:87  
[2] +(x::T, y::T) where T<:Union{Float16, Float32, Float64}  
    @ float.jl:408  
[3] +(c::Union{Uint16, Uint32, Uint64, Uint8}, x::BigInt)  
    @ Base.GMP gmp.jl:539  
[4] +(c::Union{Int16, Int32, Int64, Int8}, x::BigInt)  
    @ Base.GMP gmp.jl:545  
....  
....  
....  
[208] +(s1, s2)  
    @ Main REPL[11]:1  
[209] +(a, b, c, xs...)  
    @ operators.jl:578
```

The `string()` function is quite useful because it can be used to convert and concatenate an Any data type, although we need to be careful, as an arithmetic expression will be evaluated before the string is created:

```
julia> +(s::String,a::Any) = string(s,a)  
+ (generic function with 177 methods)  
julia> +(a::Any, s::String) = string(a,s)  
+ (generic function with 178 methods)  
julia> "Hello " + 17//11 + " World"  
"Hello 17//11 World"  
julia> "Hello " + 17/11 + " World"  
"Hello 1.5454545454545454 World"
```

Let's finish this section with an example other than squaring data structures, by defining a function that computes the Hailstone sequence of numbers.

These can be generated from a starting positive integer, n, by the following rules:

- If n is 1, then the sequence ends
- If n is even, then the next n of the sequence =  $n/2$
- If n is odd, then the next n of the sequence =  $(3 * n) + 1$

There is a conjecture by Lothar Collatz that states that a hailstone sequence for *any* starting number always terminates.

Here is the code that evaluates this, and some sample output:

```
function hailstone(n::Integer)
    @assert n > 0
    k = 1
    a = [n]
    while n > 1
        n = (n % 2 == 0) ? n >> 1 : 3n + 1
        push!(a,n)
        k += 1
    end
    return (k,a)
end
```

This function will loop forever unless the hailstone conjecture is correct, but such is our belief in the conjecture no break limit is implemented; otherwise, the computer will eventually run out of heap space.

Here are some runs with different starting values:

```
julia> hailstone(17)
(13, [17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1])

julia> (m,s) = hailstone(1000)
(112, [1000, 500, 250, 125, 376, 188, 94, 47, 142... 40, 20, 10, 5, 16, 8, 4, 2, 1] )

julia> (m,s) = hailstone(1000000)
(153, [1000000, 500000, 250000, 125000, 62500, 31250... 10, 5, 16, 8, 4, 2, 1] )
```

There is no obvious pattern to the number of iterations needed in order to converge but all integer values seem to eventually do so. Note that we restrict the parameter type to be an integer using the modifier `::Integer` and checking that it is positive with the `@assert` macro:

```
julia> for i = 1000:1000:
    (mx,sx) = hailstone(i)
    println("hailstone($i) => $mx iterations")
end
hailstone(1000) => 112 iterations
hailstone(2000) => 113 iterations
hailstone(3000) => 49 iterations
hailstone(4000) => 114 iterations
hailstone(5000) => 29 iterations
hailstone(6000) => 50 iterations
```

The function starts by creating an array with the single entry n and sets the counter (k)

- The while - end block will loop until the value of n reaches 1 and each new value is pushed onto the array. Since this effectively modifies the array, by increasing its length, the convention of using a ! is used.
- The statement `(n%2 == 0) ? n>>1 : 3n + 1` encapsulates the algorithm's logic.
- `(condition) ? statement-1:statement-2` is a shorthand for if else end, initially seen in C but borrowed by many languages including Julia.
- `n >> 1` is a bit shift left so effectively halves n when n is even.

The sequence keeps going, halving the number until an odd number appears. When that happens, we triple the number and add one to get a new (even) number, and then the process continues. While it is easy to see that the conjecture is true, the jury is still out on whether it has been proven or not.

It is worth noting that Julia orders its logical statements from left to right, so the || operator is equivalent to orelse and the && operator is equivalent to andthen.

This leads to another couple of constructs, termed short circuit evaluations, that are becoming popular with Julia developers:

```
(condition) || (statement)
# => if condition then true else perform the statement

(condition) && (statement)
# => if condition then perform the statement else return false
```

Notice that because the constructs return a value, this will be true for || if the condition is met and false for && if it is not.

Finally, the function returns two values, the number of iterations and the generated array and this must be assigned to a tuple. These constructs can be used to provide simple guards in a function via multiple return paths.

Since functions are first-class objects, this means that function references can be passed around in the same fashion as scalars, arrays, and structures; this permits us to define closures in Julia.

## Closures and currying

A closure is a way of storing a function while retaining its environment. The environment is a mapping that associates each free variable of the function, viz. variables that are used locally, but defined in an enclosing scope with its value or a reference to which the name was bound when the closure was created.

A closure, unlike a normal function, provides it with the ability to those captured variables through the closure's copies of their values or references, even when the function is invoked outside their scope.

As an example, consider the following code snippet:

```
julia> function counter()
    n = 0
    () -> n += 1, () -> n = 0
end
counter (generic function with 1 method)
```

This is a very simple function that increases the `n` variable; it returns *two* references, the first to do the incrementing and the second to reset the counter.

It is called (i.e., instanced) as follows:

```
julia> (addOne, reset) = counter()
(getfield(Main, Symbol("##3#5"))(Core.Box(0)),
 getfield(Main, Symbol("##4#6"))(Core.Box(0)))
```

So, we can call it a few times, to reset the counter and redo it, starting with zero:

```
julia> addOne(); addOne(); addOne()      #=> 3
julia> reset()                          #=> 0
julia> addOne(); addOne()              #=> 2
```

Another consequence of functions returning references is that it is possible to instantiate some of the parameters and create new (simpler) functions that can be evaluated by specifying the remainder of the parameters. This is a procedure well known to protagonists of functional programming.

The following is a simple example of currying in Julia:

```
function add(x)
    return function f(y)
        return x + y
    end
end
add (generic function with 1 method)
```

This is a relatively simple curried function, and a simpler definition is as follows:

```
add(x) = y -> x + y
```

However, the way it is written here may make the definition a bit unclear.

We can demonstrate the usage as follows:

```
# a3() creates a function to increment by values by 3.
julia> a3() = add(3)
```

```
#8 (generic function with 1 method)
# add() can be called in the following fashion
julia> add(3)(4)
7
# ... but also, more generally, as
julia> a3() = add(3);
u = 4;
julia> a3()(u)
7
```

## Passing arguments

Most function calls in Julia can involve a set of one or more arguments and, in addition, it is possible to designate an argument as being optional and provide a default value.

It is useful if the number of arguments are of varying length and we may also wish to specify an argument by name rather than by its position in the list.

How this is done will be discussed now.

### Default and optional arguments

In the examples so far, all arguments to the function were required, and the function call will produce unless all are provided. If the argument type is not given, a type of `Any` is passed. It is up to the body or the function to treat an `Any` argument for all the cases that might occur, or trap the error and raise an exception.

For example, multiplying two integers results in an integer and two reals results in a real. If we multiply an integer by a real, we get a real number. The integer is said to be promoted to a real. Similarly, when a real is multiplied by a complex number, the real is promoted to a complex number and the result is complex.

When a real and an array are multiplied, the result will be a real array, unless, of course, it is an array of complex numbers. However, when two arrays are multiplied, we get an exception raised, similar to the following:

```
julia> sq(x) = x*x
sq (generic function with 1 method)
julia> a = [1.0,2,3];
julia> sq(a)
ERROR: DimensionMismatch("Cannot multiply two vectors")
Stacktrace:
[1] sq(::Array{Float64,1}) at ./REPL[10]:
```

However, we saw previously that we can definitely use the square function using the `.*` construct, and this will now work and the elements will all be promoted to reals:

```
julia> sqq(x) = x.*x;
julia> a = [1.0, 2, 3];
julia> sqq(a)
3-element Array{Float64,1}:
 1.0
 4.0
 9.0
```

The typing of arguments is a good idea not only because it restricts function behavior but also because it aids the compiler. Just how this is done in Julia without overloading a function for every possible combination of argument types we will see later in this chapter.

Sometimes we wish for some (or all) of a function's argument to take default values if they are not provided. This is done by using an `arg = value` syntax:

```
f(x, p = 0.0) = exp(p*x)*sin(x);
t = LinRange(0.0, 8*pi, 80);
w = map(t) do x
    f(x, 0.1)
end;

using PythonPlot
plot (t, w)
```

With this code, the value for `p` is replaced by that of 0.1, whereas a call of `f(x)` would result in using the default values of 0.0.

Figure 3.1 shows a plot of this function ( $p = 0.0$  and  $0.1$ ) using PythonPlot to display the result.

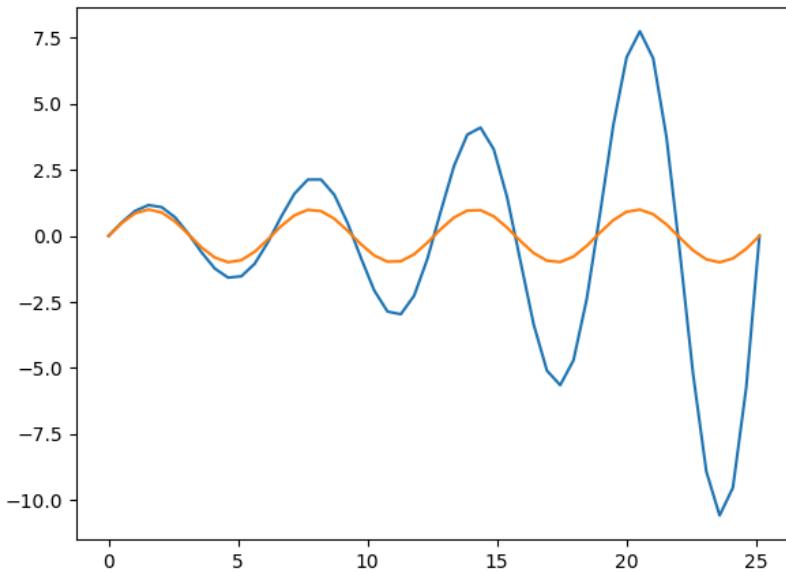


Figure 3.1 – Plot of  $e^{px} * \sin(x)$  for  $p = 0$  and  $0.1$

#### Note

In the call,  $p$  is given the value  $p=0.1$ ; however, we still could pass as (say) an integer value such as  $p = 3$  as this would be promoted in the function body to a real.

Looking at the methods for  $f()$ , we see the following:

```
julia> methods(f)
# 2 methods for generic function "f":
[1] f(x)
    @ REPL [86]:1
[2] f(x, p)
    @ REPL [86]:1
```

In fact, we could pass a rational or even a complex number:

```
julia> f(2.0, 3//4)
4.075188339491183
julia> f(2.0, 2 + 3im)
47.66857308497605 - 13.871849902204445im
```

Because of the complex argument, the result in the second case is complex too.

Optional arguments must come after required ones as otherwise, the meaning would be ambiguous. Also, when there are two optional parameters, values for all the preceding ones must be provided in order to specify those further down the list.

Defining a linear function looks like this:

```
julia> f(x, y, a=2.5, b=4.0, c=1.0) = a*x + y*b + c  
  
julia> f(1,1);           # => 7.5 : 'a' 'b' 'c' are all defaulted  
julia> f(1,1,2);         # => 7.0 : sets 'a' equal to 2  
julia> f(1,1,2.5,4.0,3.0); # => 9.5
```

The final example sets  $c = 3.0$ , but both  $a$  and  $b$  must also be specified even though they are passing their default values.

For long argument lists, this is not practicable, and it is better to use named parameters rather than simple optional ones.

## Variable argument list

First, we can look at the case where we wish to define a function that can take a variable number of arguments. We know that these types of functions exist as '`+`' is an example.

The definition takes the form `g(a, b, c...)`, where `a` and `b` are required arguments but `g` can also take zero or more arguments represented by `c`.

In this case, `c` will be returned as a tuple of values as the following illustrates:

```
function g(a ,b, c...)
    n = length(c)
    if n > 0
        x = zeros(n)
        for i = 1:n
            x[i] = a + b*c[i]
        end
        return x
    else
        return nothing
    end
end

julia> g(1.,2.); # => return 'nothing'
julia> g(1.,2.,3.,4.)
2-element Array{Float64,1}: #=> [ 7.0, 9.0 ]
```

The function needs to be sensible in terms of its arguments, but a call using rationals will work with this definition, as they are promoted to reals:

```
julia> g(1.0, 2.0, 3//5, 5//7)
2-element Array{Float64,1}:
 2.2
 2.428571428571429
```

Since functions are first-class objects, they may be passed as arguments, so *modifying* the definition of `g` slightly gives a (very poor) map function:

```
function g(a, b...)
    n = length(b)
    if n == 0
        return nothing
    else
        x = zeros(n)
        for i = 1:n
            x[i] = a(b[i])
        end
        return x
    end
end

julia> g(x -> x*x, 1., 2., 3., 4.)
4-element Array{Float64,1}:
 1.0
 4.0
 9.0
 16.0
```

Note that in the cases where there were no variable arguments, I chose to return `nothing`. This is a special variable defined by Julia of type `Nothing`.

We will meet another special type – `missing` – when discussing Julia’s implementation of data frames.

## Keyword arguments

Previously, we defined a linear function in two variables (`x,y`) with three default parameters (`a,b,c`) but met the problem that to set the `c` parameter, we need to supply values for `a` and `b`.

To do this, we can use the following syntax:

```
julia> f(x, y; a=2.5, b=4.0, c=1.0) = a*x + b*y + c;
julia> f(1.0, 1.0, c=1.0); # => 7.5
```

The only difference is that the final three arguments are separated from the first two by a semicolon rather than a comma. Now `a`, `b`, and `c` are named parameters and we can pass the value of `c` without knowing those of `a` and `b`. We can combine variable arguments and named parameters in a meaningful way:

```
function f(x...; mu=0.0, sigma=1.0)
    n = length(x)
    (n == 0) ? (return nothing) : begin
        a = zeros(n);
        [a[i] = (mu + sigma*rand())*x[i] for i = 1:n]
        a
    end
end

julia> f(1.0, 2.0, 3.0, sigma=0.5)
3-element Array{Float64,1}:
 0.36376664371632095
 0.31476941816070303
 0.777177296024868
```

So `f()` returns a Gaussian variable with mean `mu` and standard deviation `sigma`.

Because Julia supports the Unicode character set it is possible to define the function using appropriate symbols  $\mu$  and  $\sigma$ : i.e., as `f(x...; μ=0.0, σ=1.0)`.

## Scope

In the previous example, we used `condition ? statement-1 : statement-2` notation as a shorthand for `if-else-end`.

It was necessary to wrap the code following the colon in `begin-end`.

This is a form of a block, as are `if` statements, and `for` and `while` loops.

Julia always signals the termination of the most recent block by way of the `end` statement.

Other examples of blocks we have met so far are those introduced by `module`, `function`, and `struct (type)` definitions and by `try` and `catch` statements.

The question we need to consider is, if a variable is declared inside a block, is it visible outside it? This is controlled by Julia's scoping rules.

Since `if-else` or `begin-end` blocks do not affect a variable's visibility, it is better to refer to the current scope rather than the current block.

There are new scoping rules applying to the visibility of variables declared at the top level.

These were discussed in *Chapter 1*, and you are asked to read them there, if necessary, as they will not be repeated here.

Certain constructs will introduce new variables into the current innermost scope. When a variable is introduced into a scope, it is also inherited by any inner scopes unless one of the scopes explicitly overrides it.

The rules are reasonably clear:

- A declaration local introduces a new local variable
- A `const` is now only allowed at the top level
- A declaration global makes a variable in the current scope (and inner) scopes refer to the global variable of that name
- A function's arguments are introduced as new local variables into the scope of the function's body
- An assignment `x = 1` (say) introduces a new local variable `x` only if `x` is neither declared global nor introduced as local by any enclosing scope before or after the current line of code

Clarify the last statement in function `f()`:

```
function f()
    x = y = 0;
    while (x < 5)
        y = x += 1;
    end
    println(y)
end
f() ; # returns (y) => 5

function f()
    x = y = 0;
    while (x < 5)
        local y = x += 1;
    end
    return y
end
f() ; # returns (y) => 0
```

Notice that the `y` variable in the `while` loop is local to it and so the result returned by the function is 0, not 5.

There is a further construct that Julia provides in passing anonymous function definitions as an argument, which is `do - end`, and we will find it convenient when working with file IO in the next chapter.

Consider mapping an array to its squares when the value is 0.3 or more:

```
julia> a = rand(5);
julia> map(x -> begin
    if (x < 0.3)
```

```
        return(0)
    else
        return(x*x)
    end
end, a)
5-element Vector{Real}:
0
0.2362776937033681
0.9073241663584904
0.18092271236162052
0.9866796667924211

julia> map(a) do x
if (x < 0.3)
    return(0)
else
    return(x*x)
end
end
5-element Vector{Real}:
0
0.2362776937033681
0.9073241663584904
0.18092271236162052
0.9866796667924211
```

Both produce the same result but the second is cleaner and more compact. The use of the `do x` syntax creates an anonymous function with the `x` argument and passes it as the first argument to `map`.

Similarly, `do a, b` would create a two-argument anonymous function and a plain `do` would declare that what follows is an anonymous function of the form `() -> ...`.

Note that Julia does not (*as yet*) have a switch statement (*as in C*), which would be equivalent to successive `if-elseif-else-end` statements. There are packages that introduce a macro that will generate multiple `if-else` statements; one such package is `Match.jl`.

To illustrate, let us consider the mathematical proof that all odd numbers are prime! See the discussion at <http://rationalwiki.org>.

We can code this concisely using pattern matching:

```
# First add the package: i.e. Pkg.add("Match")
using Match
allodds(x) = @match x begin
    _, if !isinteger(x) || iseven(x) || (x < 3) end => "Not a valid
choice"
```

```

3 || 5 || 7 => "$x is prime"
_ => "By induction all numbers are prime"
end
# and running it on a select few gives:
julia> using Printf
julia> for i in 1:2:9
    @printf "%d : %s\n" i alloff(i)
end
1 : Not a valid choice
3 : 3 is prime
5 : 5 is prime
7 : 7 is prime
9 : By induction all odd numbers are prime

```

Finally, I will introduce a function that we will use later for timing macros. This is to solve the Queens problem, which was first introduced by Max Bezzel in 1848, and the first solutions were published by Franz Nauck in 1850.

## The Queens problem

In 1972, Edsger Dijkstra used this problem to illustrate the power of what he called structured programming and published a highly detailed description of a depth-first backtracking algorithm.

The problem was originally to place eight queens on a chessboard so that no queen could take any other, although this was later generalized to  $N$  queens on an  $N$  by  $N$  board. An analysis of the problem is given in Wikipedia. The solution to the case  $N=1$  is trivial and there are no solutions for  $N = 2$  or  $3$ . For a standard chess board, there are 92 solutions out of a possible 4.4 billion combinations of placing the queens randomly on the board, so an exhaustive solution is out of the question.

The Julia implementation of the solution uses quite a few of the constructs we have discussed:

```

struct Queen
    x::Integer
    y::Integer
end
qhorz(qa, qb) = qa.x == qb.x;
qvert(qa, qb) = qa.y == qb.y;
qdiag(qa, qb) = abs(qa.x - qb.x) == abs(qa.y - qb.y);
qhvd(qa, qb) = qhorz(qa, qb) || qvert(qa, qb) || qdiag(qa, qb);
qany(testq, qs) = any(q -> qhvd(testq, q), qs);
function qsolve(nsqsx, nsqsy, nqs, presqs = ())
    nqs == 0 && return presqs
    for xsq in 1:nsqsx
        for ysq in 1:nsqsy

```

```

        testq = Queen(xsq, ysq)
        if !qany(testq, presqs)
            tryqs = (presqs..., testq)
            maybe = qsolve(nsqsx, nsqsy, nqs - 1, tryqs)
            maybe != nothing && return maybe
        end
    end
end
return nothing
end
# Usual case is a square board with the same number of queens
qsolve(nqs) = qsolve(nqs, nqs, nqs)

julia> qsolve(8)
Queen(1, 1), Queen(2, 5), Queen(3, 8), Queen(4, 6),
Queen(5, 3), Queen(6, 7), Queen(7, 2), Queen(8, 4))

```

The code has a matrix [ `nsqsx` by `nsqsy` ] representing the board and so can be applied to non-square boards.

`qhoriz()`, `qvert()`, and `qdiag()` return true if a horizontal, vertical, or diagonal line contains more than a single queen.

`qsolve()` is the main function, which calls itself recursively and uses tree pruning to reduce the amount of computation involved.

This computation slows down markedly with increasing `n`, and I'll use this function at the end of the chapter to give some benchmarks.

## Conversion between numbers and strings

Before moving on to composite data types, I want to look at converting between strings and numeric types by asking the question, what is so special about the number 277,777,788,888,899?

This was a question posed on Numberphile, and a solution in Python was given in the YouTube video at <https://youtube.com/watch?v=Wim9WJeDTHQ>, so I thought it would be nice to do it in Julia.

It involves choosing a positive integer and multiplying all its digits together, which produces a new (lower) digit. If the result is greater than a single digit, then we repeat the process; otherwise, we stop and output the number of times it took to do this.

The conjecture is that using the value 277,777,788,888,899 – *which takes 11 steps* – is the best we can do, and here is the Julia code to do it:

```

function persistence(n::Integer)
    @assert n > 0

```

```
s = string(n)
k = 0
while length(s) > 1
    k += 1
    s = string(prod(parse.(Int64, split(s, ""))))
    @show s
end
return k
end
```

The input integer uses the `string()` function to convert to text, and this is changed into an array of characters (i.e., digits) using `split()` with no target character.

This is converted back to an array of integers using `parse()`, broadcasting over the array, then using `prod()` to compute the product and converting this back to a string, all in a single line:

```
julia> m = persistence(277777788888899);
s = "4996238671872"
s = "438939648"
s = "4478976"
s = "338688"
s = "27648"
s = "2688"
s = "768"
s = "336"
s = "54"
s = "20"
s = "0"

julia> println("Persists for $m steps")
Persists for 11 steps
```

### Note

If we only want the results, then just comment out the `@show` statement.

All the work is done in the statement that recomputes the `s` string, which then continues to loop via the `while` statement until the string length is 1.

When a 5 and an even number occur, such as `s = "54"` in the preceding code segment, then the sequence will terminate in two steps, which is what makes choosing an input value so difficult.

Note that the `parse()` function can be used on integer, real, and Boolean types and also on complex values:

```
julia> parse(Complex{Float64}, "2.2 + 3.1im")
2.2 + 3.1im
```

It also applies to numbers applying a different base (say, hex) by using an optional `base` argument:

```
julia> parse(Int64, "ffe8", base=16)
65512
```

We will look at converting strings to rational types in the next section.

## Derived and composite types

Julia implements a composite-aggregation object model rather than the most common inheritance ones, which are all used for sub-typing and polymorphism.

While this might seem restrictive, it allows the use of a multiple dispatch call mechanism rather than the single dispatch one employed in the usual object-orientated ones.

Coupled with Julia's system of types, multiple dispatch is extremely powerful. Moreover, it is a more logical approach for data scientists and scientific programmers, if for no other reason than exposing this to you, the analyst/programmer, is a reason to use Julia. In fact, there are lots of other reasons as well, as we will see later in this chapter.

## A look at the Rational type

The `Rational` number type was introduced in the previous chapter and, like most of Julia, it is implemented in the language itself. The source is in `base/rational.jl` and is available for inspection.

To see the source code for Julia, go to <https://github.com/JuliaLang/julia>.

Because `Rational` is a base type, it does not need to be included explicitly, so we can explore it immediately:

```
julia> fieldnames(Rational)
2-element Array{Symbol,1}:
:num
:den
```

The `fieldnames()` function lists what, in object-orientated parlance, would be termed properties but what Julia lists as an array of symbols. Julia uses the `:` character as a prefix to denote a symbol. There will be much more to say on symbols when we consider macros.

:num corresponds to the numerator of the rational and :den to its denominator.

**Note**

Be careful to distinguish between : and ::.

The first denotes symbols while the latter indicates a variable's type.

To see how we can construct a Rational type, we can use the methods () function:

```
julia> methods(Rational)
# 12 methods for generic function "(::Type)":
[1] (>::Type{T})(z)::Complex where T<:Real in Base at complex.jl:
[2] (>::Type{Rational})(n)::Integer in Base at rational.jl:
[3] (>::Type{Rational})(n::T, d::T) where T<:Integer in Base at
rational.jl:
[4] (>::Type{Rational})(n::Integer, d::Integer) in Base at rational.jl:
. . .
```

Obviously, we wish to construct rationals from integers, not floating-point numbers, but in Julia, there are different-sized integers, ranging from Int8 to Int128, and we also have signed and unsigned flavors of integers.

Rather than providing a recipe for making a rationale for every combination of these, Julia provides various ways that are generic and methods () helpfully lists the line in the source file in which they occur.

The [2] and [4] constructors in the preceding code segment are the simplest to understand.

Rational(n::Integer) is used for converting an integer to a rational, that is, when the denominator is 1 (and not passed).

Rational(n::Integer, d::Integer) is used when both a numerator and denominator are provided, and both are integers.

Case [3] is similar to [4] but is defined in terms of a parametric type T, which is then constrained to be a subtype of the abstract type Integer, so includes all concrete integers, both signed and unsigned.

Case [1] is an extension to the original definition (in the earlier version of Julia) to cover complex rationals *where the complex coefficients are integers*:

```
julia> z1 = 5 + 1im;
julia> z2 = 3 + 2im;
julia> z1//z
17//13 - 7//13*im
```

Parametric definitions are very useful for establishing the rules for manipulating types, as we will see later.

The source for the `Rational` type is quite long but the first few lines are informative and reproduced here (recall that the full source is at <https://github.com/JuliaLang/julia>):

```
struct Rational{T<:Integer} <: Real
    num::T
    den::T
    function Rational{T}(num::Integer, den::Integer)
        where T<:Integer
        num == den == zero(T) &&
        throw(ArgumentError("invalid rational: zero(\$T)//zero(\$T) "))
        n2,d2 = (sign(den)<0) ? divgcd(-num,-den) : divgcd(num, den)
        new(n2, d2)
    end
end
Rational(n::T, d::T) where {T<:Integer} = Rational{T}(n,d)
Rational(n::Integer, d::Integer) = Rational(promote(n,d)... )
Rational(n::Integer) = Rational(n,one(n))
function divgcd(x::Integer,y::Integer)
    g = gcd(x,y)
    div(x,g), div(y,g)
end
```

Rationals are defined using `struct`. This creates a number whose numerator and denominator are immutable; that is, they cannot be altered once the number is defined. If we need to modify a type's field, then it needs to be defined with a mutable `struct`:

```
julia> r = 5//7; r.num = 3; println(r)
ERROR: type Rational is immutable
Stacktrace:
 [1] setproperty!(::Rational{Int64}, ::Symbol, ::Int64) at ./sysimg.jl:1
 [2] top-level scope
```

The function that follows the type definition is the following:

```
function Rational{T}(num::Integer, den::Integer) where T<:Integer
    ...
    ...
end
```

It is termed the constructor and provides a recipe for creating the value. In cases where a constructor is not defined, the default is just to fill in the type's fields with the values passed.

For rationals, this will not suffice. We need to check that the denominator is not zero (or else we will generate an error) and also to reduce the rational to its least possible form by dividing by common factors between the numerator and the denominator. To this end, a `helper` function is defined using `divgcd()`, which itself uses the `Base` functions `gcd()` and `div()`. The `gnew()` special function is called to return a value for the data type.

The remaining `Rational()` constructors are special cases that will use the first definition to create the value.

We must provide rules for combining rationals with integers and what we might mean when passing real or complex numbers:

```
function // (x::Rational, y::Integer)
    xn,yn = divgcd(x.num,y)
    xn//checked_mul(x.den,yn)
end
function // (x::Integer, y::Rational)
    xn,yn = divgcd(x,y.num)
    checked_mul(xn,y.den)//yn
end
function // (x::Rational, y::Rational)
    xn,yn = divgcd(x.num,y.num)
    xd,yd = divgcd(x.den,y.den)
    checked_mul(xn,yd)//checked_mul(xd,yn)
end
//(x::Complex, y::Real) = complex(real(x)//y,imag(x)//y)
//(x::Number, y::Complex) = x*y'//abs2(y)
function // (x::Complex, y::Complex)
    xy = x*y'
    yy = real(y*y')
    complex(real(xy)//yy, imag(xy)//yy)
end
```

We see that rational numbers can be defined as complex numbers if *all* of the real and imaginary parts of *both* complex numbers are integers. This is easy to achieve by multiplying by the complex conjugate of the denominator, which splits the number into a real (rational) and imaginary (rational) part.

Note the use of `Real` in the definition refers to non-imaginary rather than floating-point numbers; integers are a type of `Real`, as are `FLOATS`:

```
julia> (1+2im) / (3 + 4im)
0.44 + 0.08im
julia> (1+2im) // (3 + 4im)
11/25 + 2/25*im
```

Just defining a type does not mean that we can do anything with it.

For a numeric type, we need to define the usual arithmetic functions (+, -, \*, /) and also comparison operations (==, <, <=, >, >=)

The rest of the code in `rational.jl` deals with definitions, and we will look at how this is done when discussing parametric types and multiple dispatch.

This module contains a method for parsing strings, converting to rationals, which is a little more involved, splitting on the //:

```
function parse(::Type{Rational{T}}, s::AbstractString) where
    T<:Integer
        ss = split(s, '/'; limit = 2)
        if isone(length(ss))
            return Rational{T}(parse(T, s))
        end
        @inbounds ns, ds = ss[1], ss[2]
        if startswith(ds, '/')
            ds = chop(ds; head = 1, tail = 0)
        end
        n = parse(T, ns)
        d = parse(T, ds)
        return n//d
    end
julia> parse(Rational{Int64}, "23//71")
23//71
```

In the next sections, we will explore how to define our own data types and create functions to manipulate them.

## A composite Vehicle data type

Let us build a data type from scratch and, as an example, let us look at data structures that describe vehicles and their ownership. This is in a file provided with the code accompanying this chapter, `vehicles.jl`, which needs to be added to the Julia environment using the `include` statement or, alternatively, encapsulated in a module.

First, we define the types and their associated fields as `struct` and the type hierarchy using the `abstract type ... end` syntax:

```
# Contact details for the vehicle's owner:
struct Contact
    name::String
    email::String
    phone::String
end
# Vehicle is the top of the abstract type chain
```

```
# Define Car, Bike and Boat as subtypes of Vehicle
abstract type Car <: Vehicle end
abstract type Bike <: Vehicle end
abstract type Boat <: Vehicle end
# Add Powerboat as subtype of Boat, and so of Vehicle
abstract type Powerboat <: Boat end
```

Now we will define some *concrete* types of cars, bikes, and boats. These will have fields defined within them, so no further subtypes can be created.

The field cannot be changed once a variable has been constructed and the type is defined using `struct`.

To be able to change the value of a field, use the `mutable struct` syntax:

```
struct Ford <: Car
    owner::Contact
    model::String
    fuel::String
    color::String
    engine_cc::Int64
    speed_mph::Float64
    function Ford(owner, make, engine_cc, speed_mph)
        new(owner,make,"Petrol","Black",engine_cc,speed_mph)
    end
end

mutable struct BMW <: Car
    owner::Contact
    model::String
    fuel::String
    color::String
    engine_cc::Int64
    speed_mph::Float64
    function BMW(owner,make,engine_cc,speed_mph)
        new(owner,make,"Petrol","Blue",engine_cc,speed_mph)
    end
end

struct VW <: Car
    owner::Contact
    model::String
    fuel::String
    color::String
    engine_cc::Int64
    speed_mph::Float64
end
```

```

struct MotorBike <: Bike
    owner::Contact
    model::String
    engine_cc::Int64
    speed_mph::Float64
end

struct Scooter <: Bike
    owner::Contact
    model::String
    engine_cc::Int64
    speed_mph::Float64
end

mutable struct Yacht <: Boat
    owner::Contact
    make::String
    length_m::Float64
end

mutable struct Speedboat <: Powerboat
    owner::Contact
    model::String
    fuel::String
    engine_cc::Int64
    speed_knots::Float64
    length_m::Float64
end

```

This defines three kinds of cars: Ford, BMW, and VW.

Ford and BMW have constructors that set the fuel type and color. In the case of a Ford, this cannot be changed (as Henry Ford said, “You can have any color as long as it is black”) but BMW is defined as a `mutable struct` instance, so this can be changed. The VW type, as with Ford, is also immutable.

Further, we define two concrete types of bikes: `Motorbike` and `Scooter`, and two types of boats: `Yacht` and `Speedboat`.

`Speedboat` is a subtype of `Powerboat`, so we could define other concrete types, such as `DutchBarge`, `NarrowBoat`, and so on, and assign different fields and create their own constructors.

With these definitions, we can start to instantiate some variables corresponding to vehicles and their owners:

```

malcolm = Contact("Malcolm", "mal@abc.net", "+44 7777 555999");
myCar = Ford(malcolm, "Model T", 1000, 50.0);

```

```

myBike = Scooter(malcolm, "Vespa", 125, 35.0);
james = Contact("James", "jim@abc.net", "+44 7777666888");
jmCar = BMW(james, "Series 500", 3200, 125.0);
jmCar.color = "Black";
jmBoat = Yacht(james, "Oceanis 44", 14.6);
jmBike = MotorBike(james, "Harley", 850, 120.0);
david = Contact("David", "dave@abc.net", "+30 7777 222444");
dvCar = VW(david, "Golf", "diesel", "red", 1800, 85.0);
dvBoat = Speedboat(david, "Sealine 28", "petrol", 600, 45.0, 8.2);

```

Note that James' BMW is black, so after creating the variable, I changed the color from blue to black by assigning `jmCar.color = "Black"`.

Given the type structure, we can already do something with these:

```

julia> Car = [myCar, jmCar, dvCar]
3-element Array{Car,1}:
Ford(Contact("Malcolm", "malcolm@abc.net", "07777555999",
    "Model-T", "Petrol", "Black", 1000, 50.0)
BMW(Contact("James", "james@abc.net", "07777666888"),
    "Series 500", "Petrol", "Black", 3200, 125.0)
VW(Contact("David", "dave@abc.net", "07777222444"),
    "Golf", "diesel", "red", 1800, 85.0)

```

This defines the `cs` array since the Ford, BMW, and VW are all subtypes of `Car`; Julia creates the appropriate array and we can iterate over it:

```

julia> for c in cs
    who = c.owner.name
    model = c.model
    make = typeof(c);
    println("$who has a $make $model")
end
Malcolm has a Ford Model-T
James has a BMW series 500
David has a VW Golf

```

Similarly, we could define the vehicles that James owns:

```
vs = [jmCar, jmBike, jmBoat]
```

This will create an Any array of type `Vehicle`, since that is the nearest common super type, and we can use this array to list the vehicles:

```

julia> println("James owns the following:")
julia> for v in vs

```

```

model = v.model
make = typeof(v);
mtype = super(make);
print("$mtype:$make:$model\n")
end
Car:BMW:Series 500
Bike:MotorBike:Harley
Boat:Yacht:Oceanis 44

```

Next, we can define a set of functions to be used with the `Vehicle` class:

```

function is_quicker(a::VW, b::BMW)
    if (a.speed_mph == b.speed_mph)
        return nothing
    else
        return(a.speed_mph > b.speed_mph ? a : b)
    end
end

```

We can compare different sorts of vehicles, such as a boat and a bike, taking into account the fact that a boat's speed is expressed in knots and a bike or car's speed is in mph or kph:

```

Function is_quicker(a::Speedboat, b::Scooter)
# Bike speeds are defined in terms of MPH ...
# whereas with Powerboats they are expressed as Knots.
# We need to convert these using a local variable 'a_mph'
KNOTS_TO_MPH = 1.151
a_mph = KNOTS_TO_MPH * a.speed_knots
if (a_mph == b.speed_mph)
    return nothing
else
    return(a_mph > b.speed_mph ? a : b)
end
end
function is_longer(a::Yacht, b::Speedboat)
    if (a.length_m == b.length_m)
        return nothing
    else
        return(a.length > b.length_m ? a : b)
    end
end
is_quicker(a::BMW, b::VW) = is_quicker(b,a)
is_quicker(a::Scooter, b::Speedboat) = is_quicker(b,a)
is_longer(a::Speedboat, b::Yacht) = is_longer(b,a)

```

```
julia> using Printf
@printf "%s %s\n" is_quicker(dvCar,jmCar) "has the faster car"
James has the faster car
@printf "The faster vehicle is %s\n" is_quicker(msBike,dsBoat)
The faster vehicle is Sealine 28
```

*We are not able to compare two BMWs or two VWs, nor any car with a Ford!*

While it would be possible to cover all the possibilities with only three types of car, it is hardly practicable to do this for all makes of car, let alone including bikes and boats. Julia solves this by using parametric types, which we will discuss later in this chapter.

Secondly, we require all vehicles to have a `speed` field and, if defining rules by use of parametric types, the `speed` field will need to be the same symbol (i.e., have the same name) in each case.

Any defined function can check for the existence of a field before trying to use it or alternatively use a `try-catch` block to trap the error, but the problem largely goes away if concrete types can inherit fields from their supertype(s).

The vehicle type, constructors, and function definitions are defined in a `vehicles.jl` file, which is provided in the source code accompanying this chapter.

## Modularization

There is some merit in treating this as a module and for this we would add the lines to the beginning of the file and, since `module` defines a block, terminate the whole with an `end` statement:

```
module Vehicles
export Contact, Vehicle, Car, MotorBike, Yacht, Powerboat, Boat
export Ford, BMW, VW, Scooter, Speedboat, isquicker, islenger
```

This is now accessed by means of the `using .Vehicles` statement, and it will be picked up from the current directory or in a special array called `LOAD_PATH`.

We can add our own directory (or *directories*) to this array by using a `push!()` call:

```
# .myjulia is a hidden directory in my home directory on a Mac
julia> push!(LOAD_PATH, "/Users/malcolm/.myjulia");
4-element Vector{String}:
 "@"
 "@v#.#"
 "@stdlib"
 "/Users/malcolm/.myjulia"
```

Making this a `.juliarc` file will ensure that it happens each time that Julia is started; again, this is a file located in my home directory.

In the next chapter, I will look more closely at setting up a `.juliarc` file and some of the other things you may wish to include each time Julia starts up.

Note that all the types and functions that we wish to be visible must be included in an `export` statement. This can come at the beginning of the module (as is usual) or the end. It does not matter.

Any function that is not exported can still be called by has to be explicitly referenced as `Vehicles.islonger(jmBoat, dvBoat)`.

Modules can also have `using` statements, so using `myModule` means that `myModule` will be available for resolving names as needed without the need to fully reference the name, with the proviso that a function name does not clash with any existing names.

Since modules are first-class objects in Julia, it is possible to shorten the syntax somewhat by using a variable as an alias:

```
using Vehicles;
vh = Vehicles;
vh.islonger(jmBoat, dvBoat)
```

When including a set of functions rather than the entire set, as in `using myModule.foo1, myModule.foo2, myModule.foo3`, this can be shortened to `using myModule: foo1, foo2, foo3`.

As well as `using` statements, modules can also use reference functions from other modules but cannot add them to the main namespace. The `importall` statement will import all functions exported from a module rather than individual ones.

Modules can also have `import` statements. These support all the same syntax as `using` but only operate on a single name at a time, although the `importall` statement will import all functions exported from a module rather than individual ones.

Importing does not add modules to be searched the way `using` does and differs in that functions must be imported with the `import` statement to be extended with new methods.

One last feature of incorporating our type system in a module is more because of convenience rather than necessity. When developing using the REPL console, any types defined are fixed, and once defined, we are unable to change them without restarting Julia and redefining them.

Modules can be reused, and this will effectively redefine all they contain, including any defined types.

*We will discuss the role that modularization plays in Julia in greater detail in the next chapter.*

## typealias and unions

It is often convenient to introduce a new name for an already expressible type, and for this, Julia provides a `typealias` mechanism.

In version 1.0 onward, the syntax has changed and now uses the following form:

```
julia> const Vct{T} = Array{T,1}
Array{T,1} where T
julia> const Mtx{T} = Array{T,2}
Array{T,2} where T
```

Type aliases are useful when defining an umbrella type as a union of simpler ones.

Union types are extensively used in Base, and there are many examples in the code listing here:

```
julia> const Signed64 = Union{Int8, Int16, Int32, Int64}
Union{Int16, Int32, Int64, Int8}
julia> const Unsigned64 = Union{UInt8, UInt16, UInt32, UInt64}
Union{UInt16, UInt32, UInt64, UInt8}
julia> const Integer64 = Union{Signed64, Unsigned64}
Union{Signed64, Unsigned64}
```

Recall that in our vehicle type, we provided contact details such as name, email, and phone type. However, alternatively, it might be more appropriate to use a postal address. To accommodate both, we can write the following:

```
mutable struct Address
    name::String
    street::String
    city::String
    country::String
    postcode::String
end
julia> postal = Address("Malcolm Sherrington", "1 Main Street",
"London", "UK", "WC2N 9ZZ")
Address("Malcolm Sherrington", "1 Main Street", "London", "UK", "WC2N
9ZZ")
julia> const Owner = Union{Contact,Address}
Owner (alias for Union{Address, Contact})
```

The alias allows us to supply the owner field either as contact or postal details:

```
struct Yacht <: Boat
    owner::Owner
    make::String
    length_m::Float64
end
julia> y1 = Yacht(me, "Moody 36", 11.02)
Yacht(Contact("malcolm", "malcolm@abc.com", "07777555999"),
"Moody 36", 11.02)
```

```
julia> y2 = Yacht(postal,"Dufour 44", 13.47)
Yacht(Address("Malcolm Sherrington", "1 Main Street", "London",
"UK", "EC1A 9ZZ"), "Dufour 44". 13.47)

julia> c1.owner.name;
```

## Enumerations

One problem with our vehicle type is that the fuel is defined as a string, whereas it would be better to restrict the choice to a set of values. Julia provides one approach to enumerations in terms of enumerations.

First, we are going to use a vector of type {Any} to hold the enumerated values. These could be `const` instances using integers or strings, but I'll restrict them to a list of symbols and create a `vnum.jl` file to hold the following code:

```
julia> const VecAny = Array{Any,1}
Vector{Any} (alias for Array{Any, 1})
function vnum(sym::Symbol...)
    aa = Any[]
    for v in sym
        push!(aa,v)
    end
    aa
end
function vidx(aa::VecAny, s::Symbol)
    for (i, v) in enumerate(aa)
        if v == s
            return (i - 1)
        end
        nothing
    end
end
vin(aa::VecAny, s::Symbol) = (vidx(aa,s) >= 0 ? true : false)
```

`vnum` is created by pushing a variable list of symbols onto an empty `Any` vector.

Additionally, there is a `vidx()` function, which returns the position in the enumeration, holding with the convention that this is zero-based, and a one-line `vin()` checks that a symbol is in `vnum`.

We can use this to define our fuel types:

```
fs = vnum(:NONE,:PETROL,:DIESEL,:LPG);
vidx(fs,:DIESEL); # => 2
vin(fs,:NONE); # => true
julia> c2.owner.name; # => "Malcolm Sherrington"
julia> c1.owner.email; # => malcolm@abc.com
```

```
julia> c2.owner.email  
ERROR: type Address has no field email  
julia> typeof(c1.owner); # => Contact  
julia> typeof(c2.owner); # => Address  
  
julia> isa(c1.owner,Contact); # => true  
julia> isa(c1.owner,Address); # => false
```

**Note**

A recent package by Fredrik Ekre has a more extensive implementation of enumerations, currently to be found on his GitHub account at <https://github.com/fredrikekre/EnumX.jl>.

Structs in Julia can be used to create composite numerical types, and in the next two chapters, we will be looking at this in more detail.

However, as a taster, the following section discusses the use of multidimensional vectors, extending the usual arithmetic functions and defining functions to compute vector norm, dot product, distance metrics, and so on.

## Multidimensional vectors and computing pi (revisited)

We will start by defining a vector in three dimensions; this is a vector in the mechanical sense and not the Julia vector sense, that is, a one-dimensional array.

We will define a V3D module using the following code:

```
#=  
# This module uses Float64 components but could use a  
# parameterised type {T} as will be seen later  
#=  
module V3D  
  
# import operators from Base and Linear Algebra  
import Base: +, *, /, ==, <, >, zero, one, iszero  
import LinearAlgebra: norm, dot  
# and export the type Vec3 and norm, dist functions (for Vec3)  
export Vec3, norm, dist  
# define a simple structure to hold the coordinates of the vector  
struct Vec3  
    x::Float64  
    y::Float64  
    z::Float64
```

```

end
# and the 'usual' runs for manipulating vectors

(+) (a::Vec3, b::Vec3) = Vec3(a.x+b.x, a.y+b.y, a.z+b.z)
(*) (p::Vec3, s::Real) = Vec3(p.x*s, p.y*s, p.z*s)
(*) (s::Real, p::Vec3) = p*s
(/) (p::Vec3, s::Real) = (1.0/s)*p
(==) (a::Vec3, b::Vec3) = (a.x == b.x)&&(a.y == b.y)&&(a.z == b.z) ? true : false;
# here are the scalar (dot) product and the norm of then vector
dot(a::Vec3, b::Vec3) = a.x*b.x + a.y*b.y + a.z*b.z;
norm(a::Vec3) = sqrt(dot(a,a));
zero(Vec3) = Vec3(0.0,0.0,0.0);
one(Vec3) = Vec3(1.0,1.0,1.0);
iszero(a::Vec3) = (v == zero(Vec3))
(<) (a::Vec3, b::Vec3) = norm(a) < norm(b) ? true : false;
(>) (a::Vec3, b::Vec3) = norm(a) > norm(b) ? true : false;
# also define the distance between two vectors
dist(a::Vec3, b::Vec3) = sqrt((a.x - b.x)*(a.x - b.x) +
                               (a.y - b.y)*(a.y - b.y) + (a.z - b.z)*(a.z - b.z))
end

```

We can now use this module to define a couple of 3D vectors and output the distance between them:

```

julia> using Main.V3D
v1 = Vec3(1.2,3.4,5.6);
v2 = Vec3(2.1,4.3,6.5);
julia> @printf "Distance between vectors is %.3f\n" dist(v1,v2)")
Distance between vectors is 1.559

```

It is also possible to create a matrix of 3D vectors (quickly):

```

julia> @elapsed begin
    vv = [Vec3(rand(),rand(),rand()) for i = 1:1000000];
    vs = reshape(vv,1000,1000);
end
0.063921754

julia> vs
1000 × 1000 Array{Vec3,2}:
Vec3(0.175379, 0.930732, 0.265873)
Vec3(0.932432, 0.495334, 0.684214)
...
...
Vec3(0.476968, 0.407128, 0.41125) Vec3(0.00346352, 0.213962, 0.622112)

```

We can now use the volume of the unit sphere (i.e.,  $4\pi r/3$ ) to estimate PI, and recall that the norm of the vectors will lie in the octant of the sphere:  $k = 0$ :

```
julia> k = 0;
julia> for i in 1:length(vv)
    if norm(vv[i]) < 1.0 k +=1 end
end
julia> @printf "Estimate of PI is %9.5f\n" 6.0*k/length(vv)
Estimate of PI is  3.14281
```

## Parameterization

One obvious problem with the module as defined previously is that the components of the vector are defined in terms of the concrete type `Float64`. As we saw earlier, it is possible to define a structure in terms of a parameterized type, and we wish to ensure that all the components are of the same type,  $\{T\}$ .

The following rewriting of the module does this:

```
module V3P
# import Base.+, Base.* , Base./, Base.norm, Base.==, Base.<, Base.>
#
import Base: +, *, /, ==, <, >
import LinearAlgebra: norm, dot
export Vec3P, norm, dist
struct Vec3P{T<:Number}
    x::T
    y::T
    z::T
end
(+) (a::Vec3P, b::Vec3P) = Vec3P(a.x+b.x, a.y+b.y, a.z+b.z)
(*) (p::Vec3P, s::Real) = Vec3P(p.x*s, p.y*s, p.z*s)
(*) (s::Real, p::Vec3P) = p*s
(/) (p::Vec3P, s::Real) = (1.0/s)*p
(==) (a::Vec3P, b::Vec3P) = (a.x == b.x) && (a.y == b.y) && (a.z == b.z) ?
true : false
dot(a::Vec3P, b::Vec3P) = a.x*b.x + a.y*b.y + a.z*b.z
norm(a::Vec3P) = sqrt(dot(a,a))
(<) (a::Vec3P, b::Vec3P) = norm(a) < norm(b) ? true : false
(>) (a::Vec3P, b::Vec3P) = norm(a) > norm(b) ? true : false
dist(a::Vec3P, b::Vec3P) = sqrt((a.x-b.x)*(a.x-b.x)
+ (a.y-b.y)*(a.y-b.y) + (a.z-b.z)*(a.z-b.z))
end
```

Now we can pass other numeric types, such as complex numbers, rationals, and so on:

```
julia> using Main.V3P
julia> z1 = Vec3P{Complex}(1 + 2im, 2 + 3im, 3 + 4im); julia> z2 =
Vec3P{Complex}(3 - 2im, 4 - 3im, 5 - 4im);

julia> z1 + z2
Vec3P{Complex{Int64}}(4 + 0im, 6 + 0im, 8 + 0im)

julia> zn = norm(z1 + z2) # => sqrt(116) it IS a complex number.
10.770329614269007 + 0.0im

julia> @assert zn == sqrt(116)
julia> r1 = Vec3P{Rational}(11//7, 13//5, 8//17) Vec3P{Rational}
(11//7, 13//5, 8//17)

julia> r2 = Vec3P{Rational}(17//9, 23//15, 28//17)
Vec3P{Rational}(17//9, 23//15, 28//17)

julia> r1 + r2
Vec3P{Rational{Int64}}(218//63, 62//15, 36//17)

julia> norm(r1 + r2) # is REAL, since sqrt(Rational) is a Float
5.791603442602598
```

## Higher dimensional vectors

We can extend the 3D vector to higher dimensions, in which case, we will need to use an array to store the vector's components and pass the number of dimensions as a second parameter.

As an example, we will use a secondary package, `StaticArrays`, which provides an `SVector` structure to hold the components. The following is not a full definition, just defining sufficient operations to calculate the distance between two N-Vectors and give yet another estimate of pi:

```
module VNX
using StaticArrays
import Base: +, *, /, ==, <, >
import LinearAlgebra: norm, dot
export VecN, norm, dist
struct VecN
    sv::SVector;
end
sizeof(a::VecN) = length(a.sv)
sOK(a::VecN, b::VecN) =
```

```

(sizeof(a) == sizeof(b)) ? true : throw(BoundsError("Vector of
different lengths"));
(+) (a::VecN, b::VecN) = [a.sv[i] + b.sv[i]
                           for i in 1:sizeof(a) if sOK(a,b)]
(*) (x::Real, a::VecN) = [a.sv[i]*x for i in 1:sizeof(a)]
(*)(a::VecN, x::Real) = x*a
(/) (a::VecN, x::Real) = [a.sv[i]/x for i in 1:sizeof(a)]
(==) (a::VecN, b::VecN) = any([(a.sv[i] == b.sv[i])
                               for i in 1:sizeof(a) if sOK(a,b)])
dot(a::VecN, b::VecN) = sum([a.sv[i]*b.sv[i]
                             for i in 1:sizeof(a) if sOK(a,b)])
norm(a::VecN) = sqrt(dot(a,a));
(<) (a::VecN, b::VecN) = norm(a) < norm(b) ? true : false;
(>) (a::VecN, b::VecN) = norm(a) > norm(b) ? true : false;
dist(a::VecN, b::VecN) = sum(map(x -> x*x, [a.sv[i]-b.sv[i]
                                              for i in 1:sizeof(a) if sOK(a,b)]))
end

```

### ***Calculating PI (again)***

The extension of the sphere to a higher dimension is termed the n-ball ( $n > 3$ ), and the volume of n-balls is available via this Wikipedia page:

[https://en.wikipedia.org/wiki/N-sphere#Volume\\_and\\_surface\\_area](https://en.wikipedia.org/wiki/N-sphere#Volume_and_surface_area)

Here is a version for 4 balls; recall that now the volume comprising all vectors with positive components is  $2^N$ , that is, 1/16th for the 4-sphere:

```

# Generate K 4-vectors
using Main.VNX, StaticArrays
K = 10^5;
vv = Array{VecN}(undef,K);
for j = 1:K
    vv[j] = VecN(@SVector [rand() for i = 1:4])
end
# Sum up the vectors which lie within the 4-ball
s = 0;
for j = 1:K
    if (norm(vv[j]) < 1.0)
        global s += 1
    end
end

```

```
# Volume of the unit 4-ball is 2*π*π
# The count is 1/16th of the volume, (again) we estimate pi as:
#
julia> mypi = sqrt(32*s/K)
3.1357550924777273
```

The technique can be extended to the n-balls inscribed into unit hypercubes. The volumes of the first 15 n-balls are provided at [https://en.wikipedia.org/wiki/Volume\\_of\\_an\\_n-ball](https://en.wikipedia.org/wiki/Volume_of_an_n-ball).

Note that from this reference, while the volume is proportional to the radius  $r$  raised to the dimension of the n-ball, oddly, the power of  $\pi$  increases by one only when the dimension increases by two.

However, there is a formula for the n-content (i.e., volume) of an n-dimensional hyperball, which is as follows:

if  $n$  is even:  $(1/(n/2)!) \pi^n / 2r^n$

if  $n$  is odd:  $(2n((n-1)/2)!/n!) \pi^{(n-1)}/2r^n$

## Summary

In this chapter, we have looked at how the Julia type system defines common numeric and string types and how the use of parametric types provides an efficient mechanism for formulating more complex type structures.

We developed a set of types for a class of vehicle types and then added data to create and manipulate some specific instances.

Finally, we looked at a numeric example of 3D vectors and showed how this could be extended to greater dimensions. We applied these types to an example we saw earlier – the Monte Carlo estimation of  $\pi$ .

In the next chapter, we are going to look at some of the topics that make Julia stand out from other computer languages, such as multiple dispatch and defining runtime macros, as well as a more in-depth discussion of using modules in Julia.

# 4

## The Three Ms

This chapter is devoted to a discussion of Julia's three Ms, which, when combined, are relatively unique in common programming languages.

These are as follows:

- **Multiple dispatch:** The feature in Julia that gives it the ability to generate functional code specific to the parametric types of the arguments passed, which provides speed equivalent to more conventionally compiled languages
- **Metaprogramming and macros:** Dynamic features to extend Julia using “genuine” macros created at runtime, adding Lisp-like functionality to the language
- **Modularity:** The packaging of type structures, functions, and macros into modules that are used as the basis for packages

We have come across some examples of all these before in the preceding chapters and will take some time out here to discuss them in more detail.

### Multiple dispatch

Multiple dispatch is the backbone of the Julia programming language, but the concept dates back to a language called ML, which was devised by Robert Milner et al in 1973 at the University of Edinburgh. Julia has made it one of the main tenets on which the language is designed.

It may be viewed as an example of polymorphism, which allows a language/program to make decisions during runtime on which method is to be invoked based on the types of parameters sent to that method.

The number of parameters used determines the “type” of polymorphism supported by a language. You are likely to be more familiar with single dispatch, which is commonly introduced, occupying a central role of the object orientation paradigm, as seen in *Figure 4.1*.

Single dispatch (e.g. Python)	Multiple dispatch (e.g. Julia)
<ul style="list-style-type: none"> <li>The first argument is special and determines a method.</li> </ul>	<ul style="list-style-type: none"> <li>All arguments are equally responsible to determine a method.</li> </ul>
<pre>class Serializer:     def write(self, val):         if isinstance(val, int)             # ...         elif isinstance(val, float)             # ...         #...</pre>	<pre>function write(dst::Serializer,               val::Int64)     # ... end  function write(dst::Serializer,               val::Float64)     # ... end  # ...</pre>

Figure 4.1 – Comparison between the single and multiple dispatch paradigms

Function names are usually selected to be descriptive of the function's purpose. It is sometimes desirable to give several functions the same name – for example, the `plot()` function occurring in most visualization packages. They perform similar tasks, that is, displaying graphic style output, but operate on different types of input data and in different contexts.

Because multiple dispatch occurs at runtime, languages of the Lisp family represent some of the major examples: **Common Lisp Object System (CLOS)**, Dylan, Clojure, and Nice. In general, these execute quite slowly, but Julia, because of LLVM compilation, differs here, producing something akin to compiled code (C/C++/Fortran) speeds.

We saw in the previous chapter that the Julia-type system consists of a hierarchy of abstract types, each terminated by concrete types that can have methods (functions) to operate on them. It is *not* possible to create a subtype of a concert.

This allows Julia to implement a form of object orientation termed delegation, as opposed to the more familiar inheritance/polymorphic approach.

In the context of Julia, delegation typically refers to a design pattern where an object forwards or delegates certain methods or operations to another object. This pattern is used to achieve code reusability and modular design by separating concerns and promoting encapsulation.

In an object-oriented context, delegation can be used to create objects that “delegate” some of their functionality to other objects, often referred to as “delegate objects.” This can be accomplished through composition, where an object contains another object as a field and forwards method calls to it.

By means of delegation, Julia can create code for functions appropriate to all the parameters and will generate different versions for individual instantiations without having to define operations on arrays passing it off to the routines in the `array.jl` file.

Let us define a trivial function that returns the reciprocal of a numeric type, as follows:

```
julia> recip(x::Number) = (x == zero(typeof(x))) ?  
           error("Invalid reciprocal") : one(typeof(x))/x;  
julia> recip(2)  
0.5  
julia> recip(11//17)  
17//11  
julia> recip(11 + 17im)  
0.02682926829268293 - 0.041463414634146344im
```

This simple definition can be applied to any type of argument for which the division operator applies without having to do anything more than a single-line definition of the `recip()` function.

The function is restricted to numeric types by use of the `::Number` annotation on the parameter and since there is the possibility of division by zero, the function checks for this using the `zero(typeof())` construct to allow for this.

Although we can pass simple numeric types, the preceding definition will not work with arrays.

```
julia> aa = rand(3)  
3-element Vector{Float64}:  
 0.3600120948340193  
 0.16984704635808245  
 0.69505376037909  
  
julia> recip(aa)  
ERROR: MethodError: no method matching recip(::Vector{Float64})  
Closest candidates are:  
 recip(::Number)  
 @ Main REPL[7]:1
```

To extend `recip()`, we need to define an additional form using a list comprehension, or equivalently (in version 1.0), use of the `map()` function:

```
julia> recip(a)::Array = map(recip,a)  
recip (generic function with 2 methods)  
julia> recip(aa)  
3-element Vector{Float64}:  
 2.777684456576499  
 5.887650220844793  
 1.4387376301001364
```

We can also map functions to an array:

```
julia> map(sin, recip(aa))
3-element Vector{Float64}:
 0.3559292028193566
 -0.38530201654469326
  0.9912929152883915
```

Although the definition of `recip()` was in terms of a vector, we can still do this:

```
julia> bb = [2.1 3.2 4.3; 9.8 8.7 7.6]
2×3 Matrix{Float64}:
 2.1   3.2   4.3
 9.8   8.7   7.6

julia> recip(bb)
2×3 Matrix{Float64}:
 0.47619   0.3125    0.232558
 0.102041  0.114943  0.131579
```

We can also do this:

```
julia> cc = recip(aa)' .* recip(bb)
2×3 Matrix{Float64}:
 1.32271   1.83989   0.33459
 0.283437  0.676741  0.189308
```

So, matrix multiplication using broadcasting works. Remember that the `'` is a short-hand notation to the `transpose()` function.

## Code generation

To investigate what is happening in Julia when running a function in the REPL, IDE, or command, it is easier to start with a very simple function, `incr()`, defined as follows:

```
# A simple function to increment its argument
julia> incr(x) = x + 1

# Test it to see it works
julia> incr(2)
3
```

Julia parses the code to produce an **abstract syntax tree (AST)**, which is akin to a functional representation that one may get in more familiar languages such as Lisp. Julia then produces three intermediate representations:

- **Lowered code:** This converts the AST into a representation that Julia can interpret, regardless of the actual types of parameters, variables, and so on
- **Typed code:** This presents an implementation for a particular set of argument types after type inference and in-lining
- **LLVM code:** This transforms the typed code into standard LLVM **internal representation (IR)**

Finally, the LLVM compiler will produce assembly/machine code corresponding to the processor on which the code is executing. The various stages of code creation are shown in *Figure 4.2* and will be investigated later in this chapter.

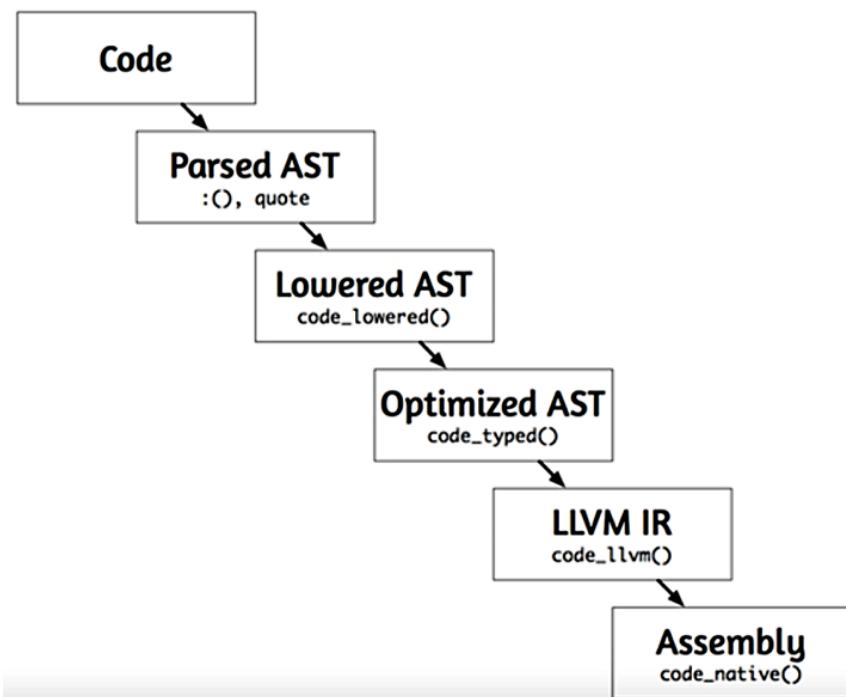


Figure 4.2 – The different stages in the Julia compilation process

All these are the same regardless of the operating system or the platform on which Julia is running.

At the LLVM IR stage, the specific LLVM compiler kicks in and produces native code.

The machine I am using is a Mac Pro with OS X 10 and an Intel X processor, so naturally any native code I display will be in Intel x86 assembly languages. Other processors will generate different code according to the LLVM backend.

Julia provides a set of routines to show the intermediate stages from the AST to native code, which are usually invoked by a set of macros.

First, let us look at the native code of the `incr()` function for various types of parameters:

```
# Increment an integer argument

julia> @code_native incr(2)
    .section __TEXT,__text,regular,pure_instructions
    .globl _julia_incr_281
    .p2align 4, 0x90
    _julia_incr_252:
; r @ REPL[21]:1 within 'incr'
    .cfi_startproc
## %bb.0:
; | r @ omt.jl:87 within '+'
    leaq    1(%rdi), %rax
; | L
    retq
    .cfi_endproc
; L
    .subsections_via_symbols
```

The only specific instruction is a single call to add 1 to an integer register: `leaq 1(%rdi), %rax`.

Now look at the code for a real (float) number:

```
julia> @code_native incr(2.7)
    .section __TEXT,__text,regular,pure_instructions
    .section __TEXT,__literal8, 8byte_literals
    .p2align 3
LCPI0_0:
    .quad 0x3ff0000000000000
    .section __TEXT,__text,regular,pure_instructions
    .globl _julia_incr_274
```

```
.p2align 4, 0x90
_julia_incr_274:
; r @ REPL[21]:1 within 'incr'
    .cfi_startproc
## %bb.0:
    movabsq $LCPI0_0, %rax
; | r @ promotion.jl:410 within '+' @ float.jl:408
    vaddsd (%rax), %xmm0. %xmm0
; | L
    retq
    .cfi_endproc
; L
.subsections_via_symbols
```

This is a little longer, using float-point (extended) registers and a different addition instruction:

```
vaddsd (%rax), %xmm0, %xmm0
```

This is still quite short, however, not all generated code is this compact. If we increment a rational (via the `rational.jl` module), we get the following:

```
julia> @code_native incr(2//7)
    .section __TEXT,__text,regular, pure_instructions
    .build_version macos, 11, 0
    .globl _julia_incr_281
    .p2align 4, 0x90
_julia_incr_281:
    .cfi_startproc
## %bb.0:
    subq $8, %rsp
    .cfi_ddef_cfa_offset 16
    movq %rsi, %rax
    movq (%rsi), %rsi
    movq 8(%rax), %rdx
    movq %rsi, %rax
    addq %rdx, %rax
    jo LBB0_2
```

```

## %bb.1
    movq    %rax, (%rdi)
    movq    %rdx, 8(%rdi)
    movq    %rdi, %rax
    popq    %rcx
    retq
LBB0_2:
    movabsq    $_j_throw_overflowerr_binaryop_283, %rax
    movabsq    $4407144664, %rdi
    callq    *%rax
    .cfi_endproc
.subsections_via_symbols

```

For a complex argument, it is possible to spot the increment code for the real part embedded here:

```

julia> @code_native_incr(2.0 + 7.0im)
    .section __TEXT,__text,regular,pure_instructions
    .build_version macos, 11, 0
    .section __TEXT,__literals8,8byte_literals
    .p2align 3
LCPI0_0:
    .quad 0x3ff0000000000000
    .section __TEXT,__text,regular,pure_instructions
    .globl _julia_incr_291
    .p2align 4, 0x90
_julia_incr_291:
    .cfi_startproc
## %bb.0"
    movq    %rdi, %rax
    vmovsd    (%rsi), %xmm0
    vmovsd    8(%rsi), %xmm1
    movabsq    $LCPI0_0, %rcs
    vaddsd    (%rcx), %xmm0, %xmm0

```

```
vomvsd    %xmm0, (%rdi)
vomsd     %xmm1, 8(%rdi)
retq
.cfi_endproc
.subsections_via_symbols,%eax)
```

### Important

Recall that native code is the end of the (virtual) processing chain.

First, the AST must be computed, then the code is lowered, the data types are inserted, and finally, the LLVM representation is created.

To reiterate, this will be common on all machines. Only the native code will be different.

The following is for the `incr()` function for a real argument. The rest I'll leave to you.

Use the `dump` function to view the parsed AST:

```
julia> dump(:(:incr(2.7)))
Expr
head: Symbol call
args: Array{Any}((2,))
 1: Symbol incr
 2: Float64 2.7
```

The expression has `head`, which refers to a `call` operation on the `Symbol incr` (which is the function call) and the argument of 2.

After this, the three intermediate stages of compilation are shown here:

```
julia> @code_lowered incr(2.7) CodeInfo(
CodeInfo(
 1 - %1 = x + 1
└── return %1
julia> @code_typed incr(2.7)
CodeInfo(
 1 - %1 = Base.add_float(x, 1.0)::Float64
```

```
└──      return %1
) => Float64
julia> @code_llvm(incr(2.7))
;   @ In[12]:1 within `incr`
define double @julia_incr_2351(double %0) #0 {
top:
; r @ promotion.jl:410 within `+` @ float.jl:408
    %1 = fadd double %0, 1.000000e+00
; L
    ret double %1
}
```

Viewing the intermediate code is principally for interest, but we will see an example that illustrates this later.

How intermediate code representation is presented has changed over different versions of Julia. Whereas we are told there will be no breaking changes in version 1, clearly how code is compiled can change, and even the LLVM generated can change.

### Important!

At the time of writing, the code generated by v1.9.0 is accurate. The code depends on the LLVM compiler, so may vary in the future.

## Metaprogramming

Julia is homoiconic, which means that a program can be written symbolically in a way that it can be manipulated as data using the language. This adds great power in as much as the program can create genuine code as it is executing, modifying due to data types, circumstances occurring at runtime, and so on.

The ability of a programming language to be its own metalanguage is termed “reflection”, and this is a valuable language feature to facilitate metaprogramming, popular in list-processing languages such as Lisp.

As remarked on earlier, in compiling its code, problems with long execution times, which arise in most scripting languages, are not present in Julia.

## Symbols and expressions

Before discussing macros, we need to understand the role of symbols and expressions in Julia.

The symbol is a type in Julia identified by a colon : () prefix:

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

For complex expressions, combining variables, constants, and functions can also be converted to a symbol with the colon notation. An alternative representation is to enclose the code in a `quote` `end` block:

```
julia> ex1 = :( (x^2 + y^2 - 2*x*y)^0.5 )
:( ((x ^ 2 + y ^ 2) - 2 * x * y) ^ 0.

julia> ex2 = quote
(x^2 + y^2 - 2*x*y)^0.5
end
quote
#= In[45]:2 =#
((x ^ 2 + y ^ 2) - 2 * x * y) ^ 0.5
end
```

If we instantiate the variables `x` and `y`, then we can evaluate the expression.

This is *not* a function call, so values need to be known beforehand:

```
julia> x = 1.1; y = 2.5;
julia> eval(ex1)
1.4
```

Note that the `@eval` macro does *not* behave as the function call unless the expression is prefixed with `$`:

```
julia> @eval ex1
:((x ^ 2 + y ^ 2) - 2 * x * y) ^ 0.5

julia> @eval $ex1
1.4

julia> @eval $(ex1) === eval(ex1)
true

julia> ex1 == ex2
false

julia> eval(ex1) == eval(ex2)
true
```

Although `ex1` and `ex2` are equivalent, evaluating the same result, the AST of the two expressions is slightly different. Here it is for `ex1` (we'll leave `ex2` to you).

We see that the first term in the AST is a call, followed by an array of arguments, which may be other symbols corresponding to variables, functions, and/or constants:

```
julia> dump(ex1)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol ^
    2: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol -
        2: Expr
          head: Symbol call
          args: Array{Any}((3,))
            1: Symbol +
            2: Expr
              head: Symbol call
              args: Array{Any}((3,))
                1: Symbol ^
                2: Symbol x
                3: Int64 2
  3: Expr
    head: Symbol call
    args: Array{Any}((3,))
      1: Symbol ^
```

```
2: Symbol y
3: Int64 2
3: Expr
  head: Symbol call
  args: Array{Any}((4,))
    1: Symbol *
    2: Int64 2
    3: Symbol x
    4: Symbol y
3: Float64 0.5
```

Dumping the expression can get very verbose even for quite short, so using the `Meta.show_sexpr` function outputs a version of the AST as a Lisp-style S-expression:

```
julia> Meta.show_sexpr(ex1)
(:call, :^, (:call, :-, (:call, :+, (:call, :^, :x, 2)), (:call, :^,
:y, 2)), (:call, :*, 2, :x, :y)), 0.5)
```

We can more clearly see how expressions are parsed into a series of symbols such as `call`, `:^`, `:x`, and constant values, that is, `2`, `0.5`.

In processing an expression (in a macro) we often need to identify the various components, for example, in the `@show` macro.

The following function(s) traverse an expression tree:

```
# Traversing a tree
# Catchall version, other than symbols or expressions
function traverse!(ex, symbols) end

# If ex is a symbol push it onto the tree
function traverse!{ex::Symbol, symbols)
  push!(symbols, ex)
end

# Main processing function.
# Distinguish between a :call and other arguments (recursively).
#
function traverse!{ex::Expr, symbols)
  if ex.head == :call # function call
    for arg in ex.args[2:end]
      traverse!(arg, symbols) # recursive
    end
  else
    for arg in ex.args
      traverse!(arg, symbols) # recursive
    end
  end
end
```

```

    end
end
end

```

Let's define a wrapper function around the `traverse!` function(s) and an empty `symbols` array in order to push any found:

```

# Notice the use of unique to prune the symbols array
#
julia> function traverse(ex::Expr)
    symbols = Symbol[]
    traverse!(ex, symbols)
    return unique(symbols) # Don't output duplicates
end

julia> ex1 = :(mad(a,b,c))
# And apply it to ex1 above.

julia> traverse(ex1)
3-element Vector{Symbol}:
:a
:b
:c

```

## Manipulating the code tree

As mentioned earlier, Julia is homoiconic, which means that there is no distinction between code and the data on which a script is operating. With Julia's metaprogramming features, it is possible to use code in order to write (or rewrite) code and modify the running program.

This is the basis of functional languages such as Lisp, OCaml, and Clojure. Normally, low execution speeds are a problem, but not so when using Julia.

To achieve this requires manipulating the symbol table of an expression, as the following example shows:

```

#= 
Consider a simple expression such as 2*3 + 7, which evaluates to 13.
Now dump the symbol table, noting that we are most interested in the
args array.
=#
julia> ex3 = :(2 * 3 + 7)
:(2 * 3 + 7)
julia> eval(ex3)
13

```

```
julia> dump(ex3)
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: Int64 3
    3: Int64 7
#=
args[2] also consist of a (sub-)args array, which again can be dumped
=#
julia> dump(ex3.args[2])
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol *
    2: Int64 2
    3: Int64 3
```

Now let's try and flip the `+` and `*` operators in the two `args` arrays:

```
julia> ex3.args[1] = :*
:*
julia> (ex3.args[2]).args[1] = :+
:+
```

Re-evaluate the expression:

```
julia> eval(ex3)
35
```

Now the new expression is equivalent to  $(2 + 3)*7 \Rightarrow 35$ , even though the brackets have not been specified, whereas with the “normal” present rules of arithmetic, we would expect  $2 + (3*7) \Rightarrow 23$ .

This is because the actual tree structure was not changed, this would mean totally different parsing, which is clear when expressing the calculation as a Lisp-style S-expression:

```
julia> Meta.show_sexpr(:(:2 * 3 + 7))
(:call, :+, (:call, :*, 2, 3), 7)
```

We can see why flipping the `+` and `*` operators gives the actual result of 35. As an exercise, it might be instructive to dump the `2 * 3 + 7` expression and see what syntax tree results.

## Macros

Armed with an understanding of symbols and expressions, we can now discuss creating macros.

A macro in Julia constructs boilerplate code by substituting its argument(s) after parsing and has no knowledge of the argument values. However, this is a great improvement to macros in (say) C/C++, which are essentially merely preprocessors prior to the parsing phase.

The first macro is very simple – if an expression is passed, it prints out its argument list; otherwise, it just returns:

```
# Check 'ex' is an expression, else just return it.
julia> macro pout(ex)
    if typeof(ex) == Expr
        println(ex.args)
    end
    return ex
end
julia> x = 1.1; @pout x
1.1
# For an expression return its arguments and then evaluate it.
julia> y = 2.3;
julia> @pout (x^2 + y^2 - 2*x*y)^0.5
Any[:^, :(x ^ 2 + y ^ 2) - 2 * x * y, 0.5]
1.199999999999997
```

The following is a slightly more complex macro, which executes a body of code a number of times. Notice the `$(`esc(n)`)` construct, which is used to stop the macro from creating a local copy of the value of `n`. This is termed macro hygiene and I'll discuss it in a little more detail in the next section:

```
julia> macro dotimes(n, body)
    quote
        for i = 1:$(`esc(n)`)
            $(`esc(body)`))
        end
    end
end

julia> @dotimes 3 print("Hi ")
HiHiHi
julia> i = 0; @dotimes 3 [global i += 1; println(i*i)]
```

```
1
4
9
```

Recall the `addOne()` closure, which we defined in the previous chapter and, coupled with this macro, can be used for a more general incrementing function:

```
julia> reset()
julia> @dotimes(3, global k = addOne())
julia> k
3
```

In the second of the preceding examples, we need to specify that the `i` variable is in the `global` scope, otherwise, we will get three 1s printed.

An alternate version of the `@dotimes` macro is `@until`, which creates a loop and breaks out when a condition fails:

```
macro until(condition, block)
    quote
        while true
            $(esc(block))
            if $(esc(condition))
                break
            end
        end
    end
end
julia> i = 0; @until (i >= 3) [global i += 1; println(i*i)]
1
4
9
```

Again, we need to specify that `i` is in the `global` scope, and because of the testing of the value of `i`, need to initialize it prior to calling the macro.

The `@until` macro can be used to implement a simple and immediate `if-then-else` construct, `iif` – equivalent to Julia's built-in `cond ? body1 : body2` statement:

```
macro iif(cond, body1, body2)
    :(if !$cond
        $(esc(body1))
    else
        $(esc(body2))
    end)
end
```

Let's test this by printing out a factorial. We will use the version in `stdlib` in `SpecialFunctions`, although the versions we have written previously could be used here too:

```
julia> using SpecialFunctions
julia> n = 10;
julia> @iif (n < 1) factorial(n) ArgumentError("$n not positive")
3628800
julia> n = -1;
julia> @iif (n < 1) factorial(n) ArgumentError("$n not positive")
ArgumentError("-1 not positive")
```

## Timing macros

We have made use of macros to time the execution of code: these were `@time`, `@elapsed`, and (from the `BenchmarkTools` package) `@benchmark` and `@bt`.

To see how these operate, we will formulate our own version.

Before coding this, we will define a function, which executes very slowly by looking at the sum of the Kempner series.

We know that the sum  $(1/n)$  diverges, but in a Kempner sequence, all values of  $n$  containing a 9 are ignored. This does converge to the value of the sum being 22.92067, but needs around 1,028 iterations to even achieve a value over 22!

In the following function, we use regular expressions to detect the terms to be ignored. If there is no match, the function returns “nothing” and it is these terms that we wish to include in the sum.

The Regex matches values containing one or more 9, that is, 9, 19, ..., 91, ..., 99, 109:

```
function kempner(n::Integer)
    @assert n > 0
    s = 0.0
    r9 = r"9" # Match a string containing a 9
    for i in 1:n
        if (match(r9,string(i)) == nothing)
            s += 1.0/float(i)
        end
    end
    return s
end
```

Now exercise the function, which will validate it and also perform the first JIT compilation:

```
julia> [kempner(10^i) for i in 1:7]
7-element Vector{Float64}:
 2.8178571428571426
```

```

4.78184876508206
6.590720190283038
8.223184402866208
9.692877792106202
11.015651849872553
12.206153722565858

```

Even for  $10^7$  terms, this is barely 50% of the known converged total.

Now we want to calculate the elapsed time for executing the `kempner()` function and will write our own `@bmk` macro to be it.

We pass a function name, the first parameter, and an integer, which is the number of times to execute the function. The different execution times are summed and the total is averaged.

Note that, before the summing loop, there is an initial call to the function, done so that any compilation time is not included in the sum.

To see the basis of our timer, we will write a modified version of `@elapsed`, called `@bmk`:

```

macro bmk(fex, n::Integer)
  quote
    let s = 0.0
    if $(esc(n)) > 0
      val = $(esc(fex))
      for i = 1:$esc(n))
        local t0 = Base.time_ns()
        local val = $(esc(fex))
        s += Base.time_ns() - t0
      end
      return s/$(esc(n)) * 10e9
    else
      end
    end
  end
end

```

Note the following:

- We need to wrap the code in a `let/end` block so that the `s` variable is visible inside the loop even if called from the top level in the REPL due to the crazy, new scoping rules.
- The `Base.time_ns()` routine returns the current clock time in nanoseconds.
- We call the `$(esc(fex))` function, escaping it for hygiene purposes (see the following).
- The function is executed  $(n+1)$  times. The first is ignored to ensure that compilation times are not taken into account.

- The code returns the mean time, so is equivalent to the `@elapsed` macro but averaged over a number of trials.

Now run it against the `kempner` function:

```
julia> @bmk kempner(10^7) 10  
0.22505771144
```

For a discussion of the Kempner series see the *arXiv.org* paper: <https://arxiv.org/pdf/0806.4410.pdf>

The paper also discusses the Irwin series. This is where only terms containing *one* 9 are selected, that is, 9, 19, 29, 90, 91, and so on, ignoring 99, 909, and so on. This also converges to the sum 23.04428, but even slower as there are many fewer terms to include in the summation.

You might like to formulate a function to compute the Irwin sequence, my version is included in the Jupyter notebook accompanying this chapter.

At first sight, it would seem that these are the missing terms in the harmonic series and so it should diverge, but it is not, so denominators such as 99, 909, 990, 991, and so on are not in either series and these make the difference as they become more common later, that is, for very large integers.

An exhausting approach needing in excess of  $10^{28}$  terms is not practicable even for Julia.

The aforementioned paper discusses ways that the sums of these series can be computed.

## Macro hygiene

Macros must ensure that the variables they introduce in their returned expressions do not accidentally clash with existing variables in the surrounding code they expand into. This is normally done by creating local variables beginning with a # character, which would be illegal in standard Julia code since the hash character marks the start of a comment but is quite valid in intermediate representations.

However, expressions that are passed to a macro as arguments may be expected to evaluate in the context of the surrounding code, interacting with and modifying the existing variables. So, such a variable should *not* be replaced by local copies.

We saw in our `@bmk` macro that values of the passed arguments can be copied into (local) variables, used instead of the \$ symbol, like the usage in strings and as we will see in tasks in the next chapter.

In fact, we used the expression of the form `$ (esc (n))`. This is because a second problem may occur with name clashes. Consider the following code in the REPL:

```
julia> import Base.@time  
# define a routine called time  
time(n::Integer) = kempner(n)
```

```
julia> @time time(10^7)
2.621327 seconds (35.65 M allocations: 1.675 GiB, 8.25% gc time)
12.206153722565858
```

Clearly, there is a difference between the `time()` routine and the `@time` macro as the former refers to `Main.time()`.

So we need to ensure that code in any argument in `@time` is resolved in the macro call environment, and this is the purpose of the escaping expression with `esc()` defined in `@time`, as well as in `@bmk`, with `local val = $(esc(ex))` rather than just `$ex`. The latter will usually work, but it does not hurt to use `esc(ex)`.

## Macro expansions

We have now seen how to write relatively simple macros, but it will be convenient, especially when debugging more complex macros, to see how the code is generated.

Julia provides the `macroexpand()` function to do just this.

Let's start by looking at a simple case of `@assert`:

```
julia> macroexpand(Main,:(@assert n > 0))
:(if n > 0
    nothing
else
    (Base.throw)((Base.AssertionError) ("n > 0"))
end)
```

This construct is a little cumbersome, so it has been turned into a `@macroexpand` macro, which eliminates the first `Main` argument and the `:()` construct, producing the same output:

```
julia> @macroexpand @assert n > 0
:(if n > 0
    nothing
else
    Base.throw(Base.AssertionError("n > 0"))
end)
```

The code is straightforward. It checks a condition, specified by a combination of *all* the arguments and does nothing if the condition is met. Otherwise, it throws an assertion error, with the text constructed from the condition:

```
julia> n = -1; @assert n > 0
ERROR: AssertionError: n > 0
```

Expanding our `@dotimes` macro is also equally clear:

```
julia> @macroexpand @dotimes 3 [global i += 1; println(i*i)]
quote
    #= REPL[22]:3 =#
    for var"#4#i" = 1:3
        #= REPL[22]:4 =#
        [global i += 1; println(i * i)]
        #= REPL[22]:5 =#
    end
end
```

We can see the local values for the `var"#4#i` loop variable, which is quite different from the `i` global variable.

The `@macroexpand` macro also provides block comments such as `#= REPL[22]:4 =#` to indicate the position of the code in the macro.

For long expansions, this can be quite distracting, so it is possible to wrap the entire expression in a `Base.remove_linenums!()` function.

Looking at expanding our `@bmk` macro, it is a little more complex with many more local variables, but the `kempner()` function and loop count of 10 are passed as is.

```
julia> Base.remove_linenums!(@macroexpand @bmk kempner(10^7) 10)
quote
    let var"#32#s" = 0.0
        if 10 > 0
            var"#35#val" = kempner(10 ^ 7)
            for var"#33#i" = 1:10
                local var"#34#t0" = (Main.Base).time_ns()
                local var"#35#val" = kempner(10 ^ 7)
                var"#32#s" += (Main.Base).time_ns() - var"#34#t0"
            end
            return var"#32#s" / (10 * 1.0e10)
        else
        end
    end
```

This is an example that is also discussed in the Julia documentation.

Horner's method is used to reduce the evaluation of polynomials to the nth power to a series of  $(n-1)$  multiplications and  $n$  additions.

The algorithm is shown in the following, continually nesting the terms from the nth term in the `x` variable, adding the next lowest coefficient, multiplying all by `x`, and so on until reaching the final coefficient:

$$\begin{aligned}
 P(x) &= a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 \\
 &= (\underbrace{a_n x^{n-1} + a_{n-1} x^{n-2} + \cdots + a_1}_\text{n-1}) x + a_0 \\
 &= ((\underbrace{a_n x^{n-2} + a_{n-1} x^{n-3} + \cdots}_\text{n-1}) x + a_1) x + a_0 \\
 &= (\dots (\underbrace{(a_n x + a_{n-1})}_\text{n-1} x + \cdots) x + a_1) x + a_0
 \end{aligned}$$

The two snippets of code here are for the “conventional” `poly_native()` power expansion and Horner method:

```
# This is NOT a macro version
function poly_native(x, a...)
    p=zero(x)
    for i = 1:length(a)
        p = p + a[i] * x^(i-1)
    end
    return p
end
```

The `a...` parameter indicates that it corresponds to a variable length argument, which, in this case, contains the value of all the coefficients. Clearly, a variable length parameter must be specified last in the list of function arguments:

```
# Define a specific instance of poly_native
julia> f_native(x) = poly_native(x,1,2,3,4,5)
julia> f_native(2.1)
152.71450000000002
```

The actual value of the polynomial is 152.7145, using `round(152.7145000000002, digits=4)` to eliminate the rounding errors:

```
#= Neither IS this, but it is better as it does not contain a power
function of x, rather it computes the value by working backwards thru'
the coefficients.
=#
function poly_horner(x, a...)
    b = zero(x)
    for i = length(a):-1:1
        b = a[i] + b * x
    end
    return b
end
julia > f_horner(x) = poly_horner(x,1,2,3,4,5)
```

```
julia > round(f_horner(2.1), digits=4)    # Chop off the rounding error
152.7145
```

Note in most languages, such as Python, and earlier versions of Julia, the second version (Horner's) executes faster than the former. Latterly, code optimization in Julia is so good that there is little difference in the elapsed times even for large polynomials.

Now let's consider a version to generate a Horner expansion with a macro.

For this, we will use a `helper` function to multiply two numbers and add a third:

```
# Define the 'helper' function: mad(x,a,b)
# [In fact Julia has this function too as - muladd(x,a,b)]
julia> mad(x,a,b) = a*x + b;
julia> mad(2.1,5,4)
14.5
```

And now, we can use `mad()` in a macro:

```
# p is a variable list of arguments, passed in an array
macro horner(x, p...)
    ex = esc(p[end])
    for i = length(p)-1:-1:1
        ex = :(mad(t, $ex, $(esc(p[i]))))
    end
    Expr(:block, :(t = $(esc(x))), ex)
end
# Check this behaves as expected
# Notice that this works but using a different calling method for the
macro.
julia> round(@horner(2.1,1,2,3,4,5), digits=4)
152.7145
```

We can look at the expansion of the `@horner` macro, which is a series of nested `mad` function calls that mimics the form  $(x) \rightarrow (((5*x + 4)*x + 3)*x + 2)*x + 1$ .

So, our macro has eliminated the looping variable, relegating it to just a series of function calls:

```
@macroexpand @horner 2.1 1 2 3 4 5
quote
    var #68#t = 2.1
    (Main.mad) (#68#t, (Main.mad) (#68#t, (Main.mad) (#68#t,
        (Main.mad) (#68#t, 5, 4), 3), 2), 1)
end
```

## MacroTools

Mike Innes has authored several packages; notable are the Juno IDE and the ML Flex and Zygote packages, which we will look at later in this book. His MacroTools package has a useful set of macros and utility functions, and we'll look at a couple of examples from it here.

We saw previously that after dumping an expression, it is possible to alter the “sense” of the code by manipulating the `args` list of the AST. It is not always that easy to discern the precise changes to make and MacroTools has some functions/macros designed to assist in the process.

First, let's introduce the `postwalk()` function, which splits an expression into symbols and then reconstructs it, so we can apply different operations to each symbol:

```
# The postwalk function is not exported from MacroTools,
# so either we must fully qualify it or "use" it by name.
julia> using MacroTools:postwalk
julia> ex0 = :(2*(1 + 3) + 4)
julia> p = postwalk(ex0) do x
       x isa Integer ? fac(x) : x
end
:(2 * (1 + 6) + 24)
julia> eval(p)
38
```

Here, `postwalk()` is involved in changing all the instances of integers into factorials, using the `fac()` function we wrote earlier.

We should note that we must apply the macro with care as it is literal in its application, with no knowledge of the underlying arithmetic or data structure, as the following will illustrate:

```
julia> ex1 = :(2*(1//3) + 4)
julia> p = postwalk(ex1) do x
       x isa Integer ? fac(x) : x
end
:(2 * (1 // 6) + 24)
julia> eval(p)
73//3
```

The `1//3` rational is converted to `1//6` since the factorial of the denominator is changed to `fac(3)`, that is, 6.

If we walk through the expression (rather than the result), it is clear how the function constructs its tree:

```
julia> p = postwalk(ex1) do x
           @show x
end
```

```
x = :+
x = :*
x = 2
x = ://
x = 1
x = 3
x = :(1 // 3)
x = :(2 * 1 // 3)
x = 4
x = :(2 * 1 // 3 + 4)
:(2 * 1 // 3 + 4)
```

Let's change the expression slightly and apply the `gamma()` function rather than the factorial. Recall that `gamma(x+1) = fac(x)` for integer values of `x`:

```
julia> using SpecialFunctions
julia> ex2 = :(3*(2//5) + 4π)
julia> p = postwalk(ex2) do x
    x isa Integer ? gamma(x) : x
end
:(2.0 * 1.0 // 24.0 + 6.0π)
julia> eval(p)
ERROR: MethodError: no method matching //(::Float64, ::Float64)
```

There are a few things wrong with this expression. Because the `gamma()` function returns a real result, even when applied to an integer, as the error suggests, the `//` operation can't handle two floats. We could write `Integer(gamma(x))` in `postwalk`. This would not raise an error, but as we saw earlier, the rational would be changed, in this case to `(1//24)`.

Less obvious is that the last term, `4π`, has been changed to `6.0π`.

This is because `gamma(4) ≈ factorial(3)`, returned as `float` since there is an implicit multiply in the expression which `postwalk` misses, and so the expression should have been constructed as `4 . 0 π` to escape the logic of the `if-then-else (?:)` statement.

Sometimes it might be necessary to grab parts of an expression by pattern matching.

Consider the following expression. We wish to put the values into variables by separating (something) with the `+` operator. We can do this by using the `@capture` macro, which returns `true` if the match succeeds:

```
julia> ex4 = :(5 + 2π + (4 + 1.5im) + 3//7)
:(5 + 2π + (4 + 1.5im))
julia> using MacroTools
julia> @capture(ex4, a_ + b_ + c_ + d_)
true
```

```
julia> (a,b,c,d)
(5, :(2π), :(4 + 1.5im), :(3//7))
```

Variables of the form `a_`, `b_`, `c_`, and `d_` are placeholders, indicated by postfixing an underscore suffix, which will copy values into `a`, `b`, `c`, and `d` respectively. Notice that we need four placeholders since the `+` in the complex number is skipped since the value is in brackets.

If there are more or fewer placeholders than four, the pattern match will fail, but it is possible to “slurp” an array of values into a single variable by postfixing it by two underscores consecutively.

The slurping variable need not come last, but clearly, there can only be a single one:

```
julia> @capture(ex4, a_ + b__ + c_)
true
julia> (a,b,c)
(5, Any[::(2π), :(4 + 1.5im)], :(2 // 5))
```

Finally, consider the expression representing the complex number  $2\pi + 4im$ . We can resolve the expression into two components on the `+`, but recall that  $2\pi$  and  $4im$  both have implicit multiplication, so the pattern: `* + *` works and returns four variables:

```
julia> ex5 = :(2π + 4im)
:(2π + 4im)
julia> @capture(ex5, a_ + b_)
true
julia> (a,b)
(::(2π), :(4im))
julia> @capture(ex5, a_ * b_ + c_ * d_)
true
julia> (a,b,c,d)
(2, :π, 4, :im)
```

## Macro reductions

Earlier we introduced the `sum()` and `reduce()` functions. Here is a quick recap.

The `sum()` function applies a unitary function to each item in an array and then, as the name implies, returns their summation. It is possible to omit the function, in which case the default is the `(x -> x)` identity function, which plainly returns the arithmetic sum of the items.

The following code uses the `recip()` function defined at the beginning of the chapter to sum the harmonic series:

```
julia> recip(x::Number) = (x == zero(typeof(x))) ?
           error("Invalid reciprocal") : one(typeof(x))/x;
julia> @time sum(recip,1:10^9)
```

```
3.832015 seconds
21.300481502347957
```

Even for the first billion reciprocals, this is pretty quick. The harmonic series is divergent and for large numbers, ( $n$ ) is approximately equal to its natural logarithm plus the Euler–Mascheroni constant ( $c$ ) @ 0.5772:

```
julia> log(10^9) + 0.5772
21.30046583694641
```

The items should all be arithmetic, in order to sum them up, and the result is prompted to the “highest” numeric type:

```
julia> A = Any[1, 2.3, 4+5im, 6//7];
julia> sum(A)
8.157142857142857 + 5.0im
```

Here is a familiar series summation:

```
julia> round(sum(x -> 1/(x*x), 1:100), digits=6)
1.634984
```

This is the `Base` problem we met in *Chapter 1*, in which Euler showed the value of the infinite series to be  $\pi^2/6$  (≈ 1.64493), so even with 100 terms, there is still a little way to go.

Conversely, `reduce()` applies a dyadic function to an array, again returning a single value as its result.

It is clear that for numeric array  $A$   $\text{sum}(A) \equiv \text{reduce}(+, A)$ :

```
# Consider the following function
julia> nm(x,y) = abs((x*x + y*y)^0.5)
julia> reduce(nm, [1,3,5,7])
9.16515138991168
```

The operation is commutative, so any permutation of the array items still returns the same result.

Julia has a function in `Base` called `hypot()`, and also another in `LinearAlgebra` called `norm()`. Both are similar to our `nm()` version but with slightly different definitions. I’ll leave it to you to use Julia’s help system to see what these differences are.

Let us employ `reduce()` to do something a little more involved, using a symbolic expression:

```
# Recall the equivalence
julia> reduce(+, 1:10) # => sum(1:10)

55
#=
```

```
Define a function plus() in the form of a symbolic expression, perform
the same reduction and evaluate the. Result
=#
julia> plus(a, b) = :($a + $b)
julia> p = reduce(plus, 1:10)
:((((((1 + 2) + 3) + 4) + 5) + 6) + 7) + 8) + 9) + 10)
julia> eval(p)
55
```

Below is the series expansion for the trigonometric SINE function:

```
julia> pp = [:($((-1)^k)*x^(1+2k) / $(fac(1+2k))) for k = 0:5]
6-element Array{Expr,1}:
:(1 * x ^ 1) / 1
:(-1 * x ^ 3) / 6
:(1 * x ^ 5) / 120
:(-1 * x ^ 7) / 5040
:(1 * x ^ 9) / 362880
:(-1 * x ^ 11) / 39916800
# We can reduce this to a single expression
julia> reduce(plus,pp)
:((((1 * x ^ 1) / 1 + (-1 * x ^ 3) / 6) + (1 * x ^ 5) / 120) + (-1 *
x ^ 7) / 5040) + (1 * x ^ 9) / 362880) + (-1 * x ^ 11) / 39916800)
# ... and evaluate it for a specific value of x
julia> x = 2.1;
julia> eval(reduce(plus,pp))
0.8632069372306019
```

Let's check it in a more conventional way, to the same precision:

```
julia> 2.1 |> (x -> x - x^3/fac(3) + x^5/fac(5) - x^7/fac(7) + x^9/
fac(9) - x^11/fac(11))
0.8632069372306019
```

## Lazy evaluation

**Lazy** is a more specialist module by Mike Innes, which uses `MacroTools` to provide Julia with the cornerstones of functional programming—lazily evaluated lists and a large library of functions for working with them.

The following examples are largely taken from the `Lazy.jl` documentation.

For the unfamiliar, “laziness” just means that the elements of the list aren’t calculated until you use them. This allows you to perform all sorts of magic, such as working with infinite lists or lists of items from the future.

This code scratches the surface. By using the `@lazy` macro, we create a list of Fibonacci numbers and pick off the first 15. Because of the Lazy evaluation, these are only evaluated at the taken time:

```
julia> using Lazy
julia> import Lazy: cycle, range, drop, take
julia> fibs = @lazy 0:1:(fibs + drop(1, fibs));
julia> take(15, fibs)
(0 1 1 2 3 5 8 13 21 34 55 89 144 233 377)
```

The Lazy module defines a set of macros that permit a functional style of writing:

```
# Pass the argument π/6 to a function sin and then onto exp
@> π/6 sin exp    # ==> exp(sin(π/6))
1.6487212707001282
```

The `@>` macro can also have functional arguments.

In functional programming terminology, this is termed “currying,” i.e. creating an intermediate function with some of the parameters defined and the remainder filled in at a later stage:

```
julia> f(x, y) = (x + y)^2
julia> @> π/6 f(1.6)
4.509671759722594
```

The `@>>` macro reverses the order of the arguments. Let us use this to output the first 15 even squares:

```
julia> esquares = @>> range() map(x -> x^2) filter(iseven);
julia> take(15, esquares)
(4 16 36 64 100 144 196 256 324 400 484 576 676 784 900)
```

We can use this macro to create a list of primes and then check whether a number is a prime.

A helper function, `takewhile()`, defined in Lazy is required here:

```
isprime(n) =
  @>> primes begin
    takewhile(x -> x<=sqrt(n))
    map(x -> n % x == 0)
    any;!
  end
```

We need to initialize the `primes` list and can also do a quick check on a value we know to be prime:

```
julia> primes = filter(isprime, range(2));
julia> isprime(113)
true
```

## Generated functions

A generated function is a user-defined function that gets expanded at compile time, allowing it to generate code based on its argument types. It has proved to be of value in some specialized areas but is not widely used.

To define a generated function, we use the `@generated` macro, which indicates that the function should be expanded at compile time, rather than executed at runtime.

The body of the generated function has access to the types of arguments but not their values, so it is possible to perform type-based dispatch based on the function's arguments.

They are seen as being useful when the behavior of the code depends on the specific types involved. For example, when mathematical operations need to be performed differently for integers and floating-point numbers, a generated function can create optimized code for each case.

It should be noted that work on the compilation and execution of “regular” Julia code has currently made any speedup of historic examples of generalized functions very hard to source.

Their major use currently seems to be in the provision of sizes for some array data types.

### Note

One of the places where generated functions are used in the Julia Base source is the `multidimensional.jl` module. Also, refer to the source for the `StaticArrays.jl` package, which employs the use of generated functions over 100 times.

I'll include a couple of short examples to demonstrate creating and executing simple generated functions:

```
#=
This is a generated function of raising a variate to a number and is
clearly just a recast of the x^p expression.
=#
julia> @generated function power(x,p)
           @show x,p
           return :(x^p)
       end
power (generic function with 1 method)
#=

The first time it is called it outputs the argument types before the
function's value.
=#
julia> power(2,3)
(x, p) = (Int64, Int64)
8
```

The same is true if one of the argument types is changed. Here, the power is now real and not an integer, but notice that it is not the case if the types of arguments remain the same:

```
julia> power(2,1.5)
(x, p) = (Int64, Float64)
2.8284271247461903
julia> power(2,0.5)
1.4142135623730951
#= Changing again, this time the first parameter is an irrational (π)
#=#
julia> power(π,2)
(x, p) = (Irrational{:π}, Int64)
9.869604401089358
```

#### Note

The function returns a symbol,  $(x^p)$ , rather than the actual value.

Here is the second example, which multiplies all the items in a tuple, returning the product. (As in most cases, this can be done by conventional coding.)

A tuple is an abstraction of the arguments of a function, which we have not formally met as yet, that is, a set of values delimited by standard brackets. Further, as an extension, `NTuple{N, T}` is a way of representing the type for a tuple of length N where all the items are of type T.

The function creates a list of the items in the `NTuple` using comprehension and returns the product of the items as symbols. In this example, the items can all be of different types as long as the product can be promoted to a single type:

```
julia> @generated function prodit(t::NTuple{N, Any}) where {N}
           items = [ L t[$i] ) for i = 1:N ]
           return :(*($items...)) )
       end
prodit (generic function with 1 method)

julia> prodit((1,2.2,3//7,4*π))
11.848292293538648
julia> typeof(ans)
Float64
# So items need not necessarily be numeric
julia> prodit(("Hello, ", "Blue ", "Eyes."))
"Hello, Blue Eyes."
julia> typeof(ans)
String
```

---

It's important to note that generated functions may be a powerful feature in the right context, and they should be used with care. Proper testing, profiling, and benchmarking are necessary to ensure that they provide actual performance benefits.

For an example that raises some problems at present, look at the following code:

```
#= Define a generated function to return the cube of its argument =#
julia> function cube(x)
           @eval @generated function cube_gf(x)
               return :(x * x * x)
           end
           return cube_gf(x)
       end
cube (generic function with 1 method)
# Running it produces an error
julia> cube(2)
ERROR: MethodError: no method matching cube_gf(::Int64)
The applicable method may be too new: running in world age 32448,
while current world is 32450.
Closest candidates are:
  cube_gf(::Any) at REPL[9]:2 (method too new to be called from this
world context.)
```

As an aside, I don't find messages such as *running in world age* useful, especially as they date back to (at least) the Julia v1.6.7 LTS release. They may be funny initially but serve no useful purpose, especially when the closest candidate is the one that has been used, together with a similar homily.

Moreover, although this fails on the first attempt, the following example demonstrates running the function again, with it working even when passing a different datatype:

```
julia> cube(2)
8
julia> cube(π)
31.006276680299816
julia> cube("Hi")
"HiHiHi"
```

#### Note

Because string concatenation in Julia uses the "`*`" operator, the final test above also works with a character argument or any type that defines the "`*`" operation.

## Functions or macros?

To misquote the Bard: to macrotize or not to macrotize? That is the question.

Most of the material in this section has been very much under the hood, explaining how metaprogramming in Julia works and how this leads to the ability to write genuine runtime macros.

If you are new to developing macros, you might find it overwhelming. One question you may well be asking is whether you can achieve everything you wish by using conventional functions rather than resorting to solutions involving macros, and the answer is mainly: *yes!*

It is possible, albeit somewhat simplistically, to distinguish between a couple of different types of macros.

The first are the ones we have met already, which are wrappers around some relatively trivial boilerplate code in order to save having to repeat that coding. These are (*among others*) the display macros such as `@assert`, `@print`, `@show`, and so on, and the timing macros such as `@time`, `@benchmark`, and `@btime`.

The latter couple are part of the `BenchmarkTools.jl` module. It is well worth a look as it demonstrates that the implementation can be quite involved, utilizing a few helper functions, but nevertheless, the code is quite easy to follow.

In the second class, several more recent Julia packages have been created by metaprogramming gurus who make extensive use of macros. We will be introduced to some of these later in this book as we delve into the topic-specific chapters, such as Flux, JuMP, Zygote, and Turing, among others.

Here, the use of macros can be seen as useful for implementing domain-specific languages.

Since Julia packages are open source, I'd advise looking at the source, possibly when using packages that are of particular interest to you, and that contain some sophisticated macros.

Additionally, there are now a wealth of video presentations on YouTube and elsewhere. One worthy of mention is from David Sanders, famous for his tutorial workshops, at JuliaCon 2021, which features a series of Jupyter notebooks. The reference is [https://github.com/dpsanders/Metaprogramming\\_JuliaCon\\_2021](https://github.com/dpsanders/Metaprogramming_JuliaCon_2021).

For now, we will turn to our third “M” (viz., modularity), and switch to more conventional programming.

## Modularity

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

One or more modules can be stored in a package, and these are managed using the Git version control system. Most Julia packages, including the official ones distributed by Julia, are stored on GitHub, where each package, conventionally, has a `.jl` or `.jl.git` extension.

We will be discussing the use of Git in the last chapter of this book.

We saw in the first chapter that packages are managed by use of the new Julia package manager, which was introduced in version 1.0, and via the use of the interactive shell mode on how to add, update, and remove them.

In addition, there is also a separate programmable API mode (*via the Pkg module*), which can be used for similar operations, and the two are completely equivalent.

We saw a few examples of Julia modules in the preceding chapters, but it is recommended to take a little time to focus on some general aspects of Julia modules:

- Modules in Julia are separate variable namespaces, introducing a new global scope that is delimited syntactically, inside the module name and the matching `end` statement.
- They allow you to create top-level definitions (i.e., global variables) without worrying about name conflicts when your code is used together with somebody else's.
- Within a module, you can control the names that are visible from other modules (via importing), and specify which of your names are intended to be public (via exporting).
- For example, this might occur if more than one module exports a function with the same name. We have seen this already with the `PyPlot` and `UnicodePlots` modules, both of which export a `plot()` routine. In this case, if both modules have been added, the function calls need to be fully qualified, that is, `PyPlot.plot()` and `Unicode.plot()`.
- Adding is possible either by using the `using` or the `import` commands. The difference is that after `using`, all the routines from a module marked to be exported are visible and can be called without qualification, whereas with `import` they are not, and need to be fully specified.
- Modules can be located via the `LOAD_PATH` variable, which we will discuss next.

As we have seen, there are three important standard modules: `Main`, `Core`, and `Base`:

- `Main` is the top-level module, and Julia starts with `Main` set as the current module.
- `Core` contains all identifiers considered “built in” to the language, that is, part of the core language and not the libraries. Every module implicitly specifies a `using core`, since nothing can be done without those definitions.
- `Base` is a module that contains basic functionality (the contents of `base/`). All modules implicitly contain `using Base`, since this is required in most situations.

Variables defined at the REPL prompt go into `Main`, and the `varinfo()` function lists variables in `Main`.

This is how my REPL looks after some of the preceding code:

```
julia> varinfo()
name           size           summary
-----  -----  -----
```

```

@bmk          0 bytes  @bmk (macro with 1 method)
Base          Module
Core          Module
InteractiveUtils 531.916 KiB Module
Main          Module
aa            7.852 KiB 1000-element Vector{Float64}
aax           7.867 KiB 10×5×5×4 Array{Float64, 4}
ans           1.358 KiB Core.CodeInfo
kempner       0 bytes kempner (generic function with 1 method)
cube          0 bytes cube (generic function with 1 method)

```

In addition to using `Base`, modules also automatically contain definitions of the `eval` and `include` functions, which evaluate expressions/files within the global scope of that module.

If these default definitions are not wanted, modules can be defined using the `baremodule` keyword instead, although as mentioned, `Core` is necessary and still imported.

## Loading a module

The `LOAD_PATH` global variable contains the directories Julia searches for in modules when calling what is required. It is in the form of a string vector.

It can be extended as `push! (LOAD_PATH, "/Users/malcolm/.myjulia")`:

```

julia> myjuliadir = joinpath(ENV["HOME"], ".myjulia")
"/Users/malcolm/.myjulia"
julia> push!(LOAD_PATH, myjuliadir)
4-element Vector{String}:
 "@"
 "@v# .#"
 "@stdlib"
 "/Users/malcolm/.myjulia"

```

Putting this statement in the `~/.julia/config/startup.jl` file will extend the `LOAD_PATH` every time Julia starts up. One feature of this method is that the additional directory comes as the final directory in the vector.

An alternate method is to put the directory into an environment variable: `JULIA_LOAD_PATH`.

On macOS (or Linux) this takes the form:

```
export JULIA_LOAD_PATH=$HOME/.myjulia:$JULIA_LOAD_PATH
```

With Windows the syntax is different, using `%` as the environment variable delimiter and `;` as a separator.

This environment variable needs to be set *before* Julia starts, so must be put in the relevant shell startup file, for example, `.profile`, `.bashrc`, `.kshrc`, and so on:

```
julia> LOAD_PATH
4-element Vector{String}:
 "/Users/malcolm/.myjulia"
 "@"
 "@v# . #"
 "@stdlib"
```

Once set, any local modules can be placed in this directory and loaded either with `using` or `import`.

An alternative “quickie” method for loading a module (say `mymodule`) while working with the REPL, is simply to source the module file via `include ("mymodule")`, which is added to `Main`, and then to reference it by a `using Main.mymodule` statement.

## Modular integers

Modular arithmetic is a system of arithmetic for integers, where numbers “wrap around” upon reaching a certain value—the modulus (plural moduli).

The concept of modular arithmetic was introduced by Carl Friedrich Gauss in his book *Disquisitiones Arithmeticae*, published in 1801.

If this seems unusual, think of how we assess time. Hours are `mod(60)`, as are minutes, whereas days are `mod(24)` or perhaps `mod(12)` depending on which clock configuration we use: 24 hr or 12 hr.

Julia v0.1 distributed an `examples` folder that includes a `ModInt` implementation. Currently, in v1.0+, this has been dropped. It needs a little revision, so it’s included next.

In modular arithmetic, integers are taken with respect to a modulus, which is a positive integer denoted by `n`. When performing calculations using modular arithmetic, essentially this is equivalent to working with the remainders of division operations.

Integers in modular arithmetic are classified by considering them equivalent if their difference is a multiple of the modulus. So, given `n`, two numbers, `a` and `b`, are congruent modulo `n`, if their difference `a - b` is an integer multiple of `n`, if there is an integer `k` such that `a - b = kn`, which is typically denoted as `a ≡ b (mod n)`.

Operations such as addition, subtraction, and multiplication are possible between values having the same modulus. It is also possible to define an inverse function, namely a value that, when multiplied, will give one `mod(n)`, and using this, it is possible to formulate a kind of division operator.

Here is a Julia representation of modular integers:

```
module ModInts
export ModInt
import Base: +, -, *, /, inv
struct ModInt{n} <: Integer
    k::Int
    ModInt{n}(k) where n = new(mod(k,n))
end
Base.show(io::IO, k::ModInt{n}) where n =
print(io, get(io, :compact, false) ? k.k : "$(k.k) mod $n")
(a::ModInt{n} + b::ModInt{n}) where n = ModInt{n}(a.k+b.k)
(a::ModInt{n} - b::ModInt{n}) where n = ModInt{n}(a.k-b.k)
(a::ModInt{n} * b::ModInt{n}) where n = ModInt{n}(a.k*b.k)
-(a::ModInt{n}) where n = ModInt{n}(-a.k)
inv(a::ModInt{n}) where n = ModInt{n}(invmod(a.k, n))
(a::ModInt{n} / b::ModInt{n}) where n = a*inv(b)
Base.promote_rule(::Type{ModInt{n}}, ::Type{Int}) where n = ModInt{n}
Base.convert(::Type{ModInt{n}}, i::Int) where n = ModInt{n}(i)
end
```

Conventionally, in a module, the code is not indented, otherwise, *all* of it would be so.

- The `ModInt` is defined as a struct and this includes a specific constructor
- Basic operations such as `+`, `-`, `*`, `/`, as well as the `inv()` function, are imported from `Base` so that they can be overloaded
- Also, a special `show()` routine is included, which is used by Julia to display `ModInt` values
- Promote and convert rules are included to deal with bare integers

Let's test the `ModInts` module as follows:

```
# Assume it is on the LOAD_PATH
julia> using ModInts
julia> m1 = ModInt{11}(2)
julia> m2 = ModInt{11}(7)
julia> m3 = 3*m1 + m2
2 mod 11 ; # => 13 mod 11, i.e 2 mod 11
```

Because of multiple dispatches, we can operate on arrays of modular integers and do the following:

```
julia> mm = reshape([ModInt{11}(rand(0:10)) for i = 1:100], 10, 10);
julia> ma = [ModInt{11}(rand(0:10)) for i = 1:10];
julia> mm.*ma'
```

```
10x10 Matrix{ModInt{11}}:
3 1 9 3 3 7 7 1 0 0
2 6 1 2 10 9 2 0 0 2
8 5 1 5 2 4 8 3 0 1
1 1 4 5 7 8 1 8 0 0
9 1 3 3 4 1 9 9 0 9
8 1 3 10 8 7 10 4 0 5
10 4 5 3 7 6 2 8 0 1
3 9 3 7 8 4 9 10 0 4
2 5 3 0 8 0 6 9 0 10
9 9 0 5 0 4 7 10 0 7
```

For some fun, let's apply the `prodit` generated function we defined previously to a vector of `ModInt` modules:

```
julia> mb = [m = ModInt{11}(rand(0:10)) for i = 1:10];
julia> prodit(Tuple(filter(m -> (m.k > 0), mb)))
4 mod 11
```

We need to eliminate any items having the value `(0 mod 11)` since that will result in a product also having the value `(0 mod 11)`. For this, we filter on the `k` value of the modular integer.

## Methods

A function takes a list of arguments (aka a tuple) and returns a value. It is common to define the function in different ways depending on the types of arguments. Many languages term this as function overloading, however, in Julia, the overloaded versions are called methods, because parts of a multiple dispatch are often quite compact.

List the methods and the sources in which they are to be found, together with the appropriate line number:

```
julia> methods(*)# 312 methods for generic function "*" from Base
i.e.
[275] *(x ::Rational, y ::Rational)
@ rational.jl :347
```

The Julia documentation contains an extensive discussion on methods – refer to it for additional information:

<https://docs.julialang.org/en/v1/manual/methods/>

As an example, let's use the `cube()` function defined earlier:

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

This can be considered a catch-all method for any type on which the (\*) operator can be applied. For the modular integer defined, we do the following:

```
julia> cube(ModInt{11}(7))  
2 mod 11
```

Now create a second method for the cube of an array as follows:

```
julia> cube(X::Array) = X*X'*X # X' is shorthand for adjoint(X)  
cube (generic function with 2 methods)
```

I know this is not a conventional `cube`, but indulge me. It has the nice property of returning an array of the same shape as its argument:

```
julia> A = [1 4 9]  
1x3 Matrix{Int64}:  
julia> cube(A)  
1x3 Matrix{Int64}:  
98 392 882
```

This need not be limited to integers, of course:

```
julia> B = [1 2.1 3; 1 4 9π]  
2x3 Matrix{Float64}:  
1.0 2.1 3.0  
1.0 4.0 28.2743  
julia> cube(B)  
2x3 Matrix{Float64}:  
108.633 407.153 2707.32  
910.661 3463.62 23366.9
```

The (\*) operator is used in Julia for string concatenation. Again, let's slightly modify our function – this time, when working on a string:

```
julia> cube(s::String) = string(s, " ", s, " ", s)  
cube (generic function with 3 methods)
```

This creates a third method, which we can use to create the British Bobby's infamous greeting:

```
julia> cube("Hello")  
"Hello Hello Hello"
```

## Testing

A cornerstone of creating any production-quality software is to have the ability to add some degree of test harness. This may be useful in the following situations:

- The software is finally completed
- It is modified, to implement further changes, address bugs, and so on
- There are changes to the environment, for example, new versions of the compiler or the operating system

Julia has a module in Base called `Test`, which furnishes a set of macros to aid in the testing process:

```
#=
In the simplest case the @test macro behaves in a similar fashion to
@assert, except it outputs Test Passed or Failed
=#
julia> using Test
julia> x = 1;
julia> @test x == 1
Test Passed
```

When a test fails, it is more verbose, indicating where the test fails and why:

```
julia> @test x == 2
Test Failed at REPL[7]:1
Expression: x == 2
Evaluated: 1 == 2
There was an error during testing
```

Other macros can be used, for example, to test for an argument, domain, bounds errors, and so on. In the following example, we are testing that a specific error *is* trapped:

```
# Generate an array of 10 elements and try to set the 11th
julia> a = rand(10);
julia> @test_throws BoundsError a[11] = 0.1
Test Passed
Thrown: BoundsError
#=
```

The above is a bounds error, so if we check for a different error type, we still get the error report but this time the test did not succeed:

```
=#
julia> @test_throws DomainError a[11] = 0.1
Test Failed at REPL[12]:1
```

```
Expression: a[11] = 0.1
Expected: DomainError
Thrown: BoundsError
ERROR: There was an error during testing
```

The Test suite can also define a series of tests that can be executed as a whole. The following is an example to exercise some well-known trigonometric formulae, in which I have deliberately got one wrong.

It is run by using a `@testset` macro, followed by a title (as a string) and a `begin/end` block containing normal Julia code and a set of `@test` macros:

```
julia> @testset "Trigonometric identities" begin
x = 2/3*pi
@test sin(-x) ≈ -sin(x)
@test cos(-x) ≈ -cos(x)
@test sin(2x) ≈ 2*sin(x)*cos(x)
@test cos(2x) ≈ cos(x)^2 - sin(x)^2
end;
Trigonometric identities: Test Failed at REPL[16]:4
Expression: cos(-x) ≈ -(cos(x))
Evaluated: -0.4999999999999998 ≈ 0.4999999999999998
Test Summary: | Pass Fail Total
Trigonometric identities | 3 1 4
ERROR: Some tests did not pass: 3 passed, 1 failed, 0 errored, 0
broken.
```

#### Note

The report generated helps identify the test that failed, `cos (-x) == +cos (x)`, and not `-cos (-x)` as was specified.

## Ordered pairs

For a more complex example, I will close this chapter by defining some operations on an “ordered pair.”

This is a structure with two parameters (`a` and `b`) where the order *does* matter. Some operations will be different when applied to the pair (`b` and `a`) as opposed to (`a` and `b`).

We want to ensure that the parameters of the ordered pair are numbers, not, say, strings, dates, and so on, and wish to create a set of arithmetic operations, all of which need to be imported from `Base`.

The ordered pair is defined by `OrdPair(a,b) = a + be`, where `e` satisfies `e2 = 0`.

This is analogous to complex numbers, except `e2 = 0` instead of `i2 = -1`.

Or, think in terms of dropping higher-order (`O(e2)`) terms.

The arithmetic rules are slightly amended complex numbers, in the form of the second value:

$$\begin{aligned}(a + be) \pm (c + de) &= (a + c) \pm (b + d)e \\(a + be) * (c + de) &= (a*c) + (bc + ad)e \\(a + be) / (c + de) &= (a/c) + (bc - ad)/d^2e\end{aligned}$$

Note that as well as the normal functions such as `+`, `-`, `*`, `/`, and so on, comparison rules such as `==`, `<`, `<=`, need to be defined, and also a set of functions such as `abs`, `zero`, `one`, and so on. Not all now reside in `Base`, some (such as `norm`) are in the `LinearAlgebra` module.

We also need to export the `OrdPair` definition.

This is not an exhaustive implementation but we can do a lot with it, as we will see later in this section:

```
module OrdPairs
using LinearAlgebra
# Import basic functions so they can be overloaded
import Base: +,-,*,/,,==,!,,>,<,>=,<=
import Base: abs,conj,inv,zero,one,show
import LinearAlgebra: transpose,adjoint,norm
# Define the OrdPair structure
struct OrdPair{T<:Number}
    a::T
    b::T
end
OrdPair(x::Number) = OrdPair(x, zero(Number))
# Create a show function which replaces the standard constructor
function show(io::IO,u::OrdPair)
    op::String = (epsilon(u) < 0.0) ? " - " : " + ";
    print(io,value(u),op,abs(epsilon(u)), " ∈")
end
# Define some local functions for use in the module
value(u::OrdPair)      = u.a
epsilon(u::OrdPair)     = u.b
zero(::Type{OrdPairs.OrdPair}) = OrdPair(zero(Number),zero(Number))
one(::Type{OrdPairs.OrdPair}) = OrdPair(one(Number),zero(Number))
abs(u::OrdPair)         = abs(value(u))
norm(u::OrdPair)        = norm(value(u))
# Add some basic arithmetic and logical functions
+(u::OrdPair, v::OrdPair) = OrdPair(value(u)+value(v),
epsilon(u)+epsilon(v))
-(u::OrdPair, v::OrdPair) = OrdPair(value(u)-value(v), epsilon(u)-
epsilon(v))
*(u::OrdPair, v::OrdPair) = OrdPair(value(u)*value(v),
epsilon(u)*value(v)+value(u)*epsilon(v))
```

```

/(u::OrdPair, v::OrdPair) = OrdPair(value(u) /
value(v), (epsilon(u)*value(v) - value(u)*epsilon(v)) /
(value(v)*value(v)))
==(u::OrdPair, v::OrdPair) = norm(u) == norm(v)
!=(u::OrdPair, v::OrdPair) = norm(u) != norm(v)
>(u::OrdPair, v::OrdPair) = norm(u) > norm(v)
>=(u::OrdPair, v::OrdPair) = norm(u) >= norm(v)
<(u::OrdPair, v::OrdPair) = norm(u) < norm(v)
<=(u::OrdPair, v::OrdPair) = norm(u) <= norm(v)
# Extend these functions to mixed number types
+(x::Number, u::OrdPair) = OrdPair(value(u) + x, epsilon(u))
+(u::OrdPair, x::Number) = x + u
-(x::Number, u::OrdPair) = OrdPair(x - value(u), epsilon(u))
-(u::OrdPair, x::Number) = OrdPair(value(u) - x, epsilon(u))
*(x::Number, u::OrdPair) = OrdPair(x*value(u), x*epsilon(u))
*(u::OrdPair, x::Number) = x*u
/(u::OrdPair, x::Number) = (1.0/x)*u
# Finally define some additional function
conj(u::OrdPair) = OrdPair(value(u), -epsilon(u))
inv(u::OrdPair) = one(OrdPair)/u
transpose(u::OrdPair) = transpose(uu::Array{OrdPairs.OrdPair,2}) =
[uu[j,i] for i=1:size(uu)[1], j=1:size(uu)[2]]
adjoint(u::OrdPair) = u
convert(::Type{OrdPair}, x::Number) = OrdPair(x, zero(x))
promote_rule(::Type{OrdPair}, ::Type{<:Number}) = OrdPair
export OrdPair    # Also we must export the OrdPair definition
end

```

At first sight, this seems to be a different representation of complex numbers, but if we inspect the multiplication rule more closely, that is,  $(u_1, v_1) * (u_2, v_1) = (u_1u_2, u_1v_2 + u_2v_1)$  is a different form from that for complex arithmetic, namely  $(u_1u_2 - v_1v_2, u_1v_2 + u_2*v_1)$ .

In fact, these form the basis of a numeric type, termed a **Dual Number**.

In linear algebra, dual numbers extend real numbers by adjoining the new element  $\epsilon$  with the property, so that terms in  $\epsilon$  of order two or more are zero. Because of this convention, the preceding package uses the `value()` and `epsilon()` functions to retrieve the first and second components of the number respectively.

With this property in mind, we can list the normal rules of dual numbers that define their arithmetic operations.

Dual numbers were introduced in 1873 by William Clifford and were used at the beginning of the twentieth century by the German mathematician Eduard Study, who used them to represent the dual angle, which measures the relative position of two skew lines in space. Another important application of dual numbers is that of automatic differentiation.

---

This is quite a minimal implementation of the ordered pair type, but we can exercise it and see what can be achieved.

The code for the module is available as a file accompanying this book, so let's include this in the REPL and then use it:

```
julia> include("OrdPairs.jl")
Main.OrdPairs
julia> using Main.OrdPairs
```

First, define a couple of `OrdPair` instances and test to see whether some basic arithmetic and logical operations work:

```
julia> p = OrdPair(2.3,5.1)
2.3 + 5.1 ∈
julia> q = OrdPair(4.4,-0.9)
4.4 - 0.9 ∈
julia> p*q
10.12 + 20.37 ∈
julia> p/q
0.5227272727272726 + 1.2660123966942147 ∈
julia> p/2
1.15 + 2.55 ∈
julia> p < q
true
```

Note the neatness of the output of the `show()` function rather than the default `OrdPair(a,b)` style, which would be displayed without it.

We have a problem with the `norm()`, as it was not exported in the module, but can still get a value by fully qualifying the function call:

```
julia> norm(p)
ERROR: UndefVarError: `norm` not defined
julia> OrdPairs.norm(p)
2.3
```

The reciprocal function is visible, so this works:

```
julia> inv(p)
0.4347826086956522 - 0.9640831758034027 ∈
```

Because of the presence of multiple dispatch, we can create arrays of ordered pairs and apply (some) operations as long as these are based on the elementary arithmetic functions that have been defined:

```
julia> pp = [OrdPair(rand(),rand()) for i in 1:6]
6-element Vector{OrdPair{Float64}}:
```

```

0.7908174446759783 + 0.6347655483824015 ∈
0.4864409560879196 + 0.4216451574480897 ∈
0.8320783176314903 + 0.9410827278207129 ∈
0.12926386323068983 + 0.1416596845400162 ∈
0.9835957252634453 + 0.7835029308544909 ∈
0.06590101152043704 + 0.7325420838214045 ∈
julia> sum(pp)
3.288097318409961 + 3.655198132867116 ∈

```

Can we apply some statistical analysis? Yes and no. Again, it depends on the implementation and methods used:

```

julia> using Statistics
julia> mean(pp)
0.5480162197349935 + 0.6091996888111859 ∈
julia> std(pp)
ERROR: MethodError: no method matching length(::OrdPair{Float64})

```

We can reshape the vector to create a matrix and compute the transpose:

```

julia> qq = reshape(pp, 3, 2)
3×2 Matrix{OrdPair{Float64}}:
 0.790817 + 0.634766 ∈ 0.129264 + 0.14166 ∈
 0.486441 + 0.421645 ∈ 0.983596 + 0.783503 ∈
 0.832078 + 0.941083 ∈ 0.065901 + 0.732542 ∈
julia> qq'
2×3 adjoint(::Matrix{OrdPair{Float64}}) with eltype OrdPair{Float64}:
 0.790817 + 0.634766 ∈ 0.486441 + 0.421645 ∈ 0.832078 + 0.941083 ∈
 0.129264 + 0.14166 ∈ 0.983596 + 0.783503 ∈ 0.065901 + 0.732542 ∈

```

If you recall the `mad()` function defined earlier, this features simple arithmetic methods, so it works:

```

julia> mad(a,b,c) = a*b + c
mad (generic function with 1 method)
julia> mad(2,p,3) # i.e., 2*p + 3
7.6 + 10.2 ∈
# Or even ...
julia> r = one(OrdPair)
1 + 0 ∈
julia> mad(p,q,r)
11.12 + 20.37 ∈

```

Finally, define a simple cube function and broadcast it over an array of ordered pairs:

```

julia> cube(x) = x*x*x      # Don't define it as x^3
cube (generic function with 1 method)

```

```
julia> cube(p)
12.16699999999996 + 80.93699999999998 ∈
julia> cube.(qq)
3x2 Matrix{OrdPair{Float64}}:
 0.494571 + 1.19093 ∈ 0.00215989 + 0.00710104 ∈
 0.115104 + 0.299315 ∈ 0.95159      + 2.27402 ∈
 0.576093 + 1.95469 ∈ 0.000286204 + 0.00954417 ∈
```

Complete implementations of ordered pairs (viz., dual numbers) can become quite extensive

The **JuliaDiff** community group, which contains a set of differentiation tools in Julia, is one you can refer to for further study:

<https://github.com/JuliaDiff/DualNumbers.jl>

We will meet other modules from the JuliaDiff group when discussing automatic differentiation later in the book.

## Summary

In this chapter, we have discussed three aspects of Julia that serve to make it stand out from other scripting languages.

First, we looked at the idea of multiple dispatch. This is a relatively new paradigm within object-oriented programming, very different from the more common polymorphic/inherence one, which uses a single dispatch method. The advantage we saw was that it permitted Julia to compile specific, compact, well-optimized code, and in addition, when coupled with delegation, meant that there was no need to implement routines for (say) array operations, broadcasting, and so on.

Secondly, we discussed homoiconic representations of the Julia code, in terms of symbols and expressions and how these can be instanced and then evaluated as part of the running process. This leads to a system for creating macros that can inject code into the program, which stands in place of (often) considerable boilerplates.

Finally, we saw how Julia “knits” all these together in the form of modules, which extend the language by adding datatypes, functions, and macros pertinent to aspects. Several such modules are provided by Julia’s Base and standard libraries, but the majority are from third parties, covering a variety of disciplines: statistics, analytics, visualization, database, and so on.

We will be meeting some of the main ones in later chapters of this book.

First, we are going to change the topic somewhat, looking at how Julia can interact directly with other programming languages such as C, Python, R, and Java, or by scripting with the underlying operating system.



# 5

## Interoperability

In this chapter, I will discuss cooperation between Julia and other languages with the underlying operating system, and how Julia naturally encourages the use of parallel processing without the need for frameworks such as Hadoop or Spark.

The developers of Julia naturally focused on calling from Julia, and most of the discussions will be concerned with that; however, handles were provided to go the other way and call Julia from C and hence incorporate it into foreign code. A brief overview of this is given later.

Julia has a rich and varied syntax, but a vast wealth of code has always existed in object libraries covering a wide range of specialties. So, it was considered that a simple mechanism to call functions from these libraries would prove to be useful, and a one-line native `ccall()` instruction call was added. This became a cornerstone of the way Julia operates, as we will see later.

C-style connectivity led to the development of packages that effectively wrap code around existing **application programming interfaces (APIs)**.

For C and Fortran, a natural interface was created upon Julia's introduction. For others, such as Java, Python, and R, this interface has been used to create convenient means of using these languages.

What of others, such as Perl and Ruby? In fact, it is possible to run these as “tasks,” usually as part of a command pipeline, and if necessary, capture the result from the task and then postprocess the output within Julia.

In this chapter, we will cover working with non-native languages such as:

- Calling functions written in C and its related siblings: Fortran and C++
- Interfacing with other languages via Julia packages; for example, Python, R, and Java
- Running tasks via operating system commands and applying this to non-supported languages such as Perl and Ruby

**Note**

By non-native languages, I am referring to any programming language other than Julia.

Spawning tasks and pipelining were introduced as concepts in Unix, and operating systems such as OS X and Linux are especially well suited, whereas some of the code we encounter later may not execute on Windows.

## C and Fortran

Julia has a single `ccall()` function that will run a routine compiled and bundled into a shared object library.

The call has the form

```
ccall((func,libr),ret_type,(arg1_type,arg2_type,...),arg1,arg2,...)
```

where the object code for the `func` function is present in the `libr` shared library and returns a value of `ret_type`, followed by an optional tuple of arguments and the actual argument values.

If the library name is omitted, then it defaults to the default Julia library, `libjulia`, and in this case, the first argument is now specified as a string rather than a tuple.

If necessary, Julia can “find” the library, which usually requires adding it to a library on the `LD_LIBRARY_PATH` load library path or modifying the latter to contain a new directory.

**Note**

As of version 1.0+, `ccall()` may have a function pointer as a first argument.

An alternate, more convenient form, is to use the `@ccall` macro:

```
@ccall lib.fun(arg1::arg1_type,arg2::arg2_type,...)::ret_type.
```

For example, `@ccall strlen(s::Cstring)::Csize_t` returns the length of a C-type string using the standard C library.

While it is possible to create **static** libraries (*typically with a .a extension*), most C (*and Fortran*) libraries are compiled into shared libraries and are distributed as such.

On Unix/Linux systems, these (usually) have a `.so` extension; on OS X, the extension is `dylib`, and on Windows, it is `dll`.

To assemble your own C code, it is necessary to make it position-independent by compiling with the `-fPIC` switch, which we will do later.

---

To clarify the `ccall()` function, we will start with the Hello World example of accessing the system clock.

The routine is part of the `libc` system library and has no input parameters, returning a 32-bit integer:

```
julia> ctime() = ccall((:clock, "libc"), Int32, ())
julia> ctime()
10590297
```

The unit of the clock ticks is defined by the `CLOCKS_PER_SEC` constant, defined under the POSIX specification as  $10^{-6}$  s (that is, 1  $\mu$ s).

For a 32-bit integer, this is a little over 71 minutes and corresponds to the *approximate* time that is consumed by a process, so is useful for timing tasks, similar to (say) Julia's `@time` macro.

To get the genuine system time (that is, the number of seconds since 01/01/1972), we need to return the value of (`unsigned long`) `time(NULL)`, again from `libc`.

Both these functions are defined in the `myfuncs.c` source code accompanying this book, which contains a few other functions we will meet later in this chapter.

As a second example, we will link to a `mad` function, similar to the one we met in the previous chapter, to calculate the expression  $(a \times b + c)$ .

Again, this is available from a routine in `libc`; here, it is called `fma`, taking 64-bit floats, and returning the same:

```
julia> mad(a,b,c) = ccall((:fma, "libc"),
                           Float64, (Float64, Float64, Float64), a, b, c)
julia> mad(3.1,5.2,7.4)
23.52
```

The following code can be used to generate some integer random numbers, again 32-bit, and so in the range 0:4294967295:

```
julia> [ccall((:rand, "libc"), Int32, ()) for I = 1:6]
6-element Array{Int32,1}:
1000393465
963493892
1415144464
951615923
1498098652
1445766736
```

This is equivalent to the `rand(UInt32, 6)`, Julia function, except that the default output for `UInt` is displayed as a hexadecimal.

## Mapping C types

It is important to exactly match a declared C type with the equivalent declaration in Julia.

Julia automatically inserts calls to the `cconvert()` function in Base to convert each argument to the specified type.

It has corresponding types for all C types, such as `Clong`, `Cdouble`, and `Cstring`. Refer to the Julia online documentation for a comprehensive list: <https://docs.julialang.org/en/v1/manual/calling-c-and-fortran-code>.

### Note

One caveat is that Julia's `Char` type is of 32-bit length, so C characters must be passed as bytes (that is, `UInt8`).

Composite types such as structs in C (or TYPE in Fortran 90 or later) can be supported in Julia by creating `struct` definitions with a corresponding field layout.

## Calling Fortran routines

Fortran code, when compiled, creates exactly equivalent object code to that written in C, so there is no difference between object libraries in either language; it is possible, but not common, to mix the two in a single library.

There is a wealth of existing Fortran routines, especially in the fields of scientific programming. From the beginning, Julia has utilized these rather than implementing the same natively.

The only thing we need to be aware of in expressing the call is that Fortran passes scalar arguments by reference, not by value; that is, as the addresses of the parameters.

As an example, we will call a Fortran routine from libLAS, to compute the dot product between two real (double) arrays:

```
function dotprod(XX::Vector{Float64}, YY::Vector{Float64} )
    @assert length(XX) == length(YY)
    n = length(XX)
    ix = iy = 1
    prod = ccall((:ddot, "libblas"), Float64,
                 (Ptr{Int64}, Ptr{Float64}, Ptr{Int64},
                  Ptr{Float64}, Ptr{Int64}),
                 Ref(n), XX, Ref(ix), YY, Ref(iy))
    return prod
end
```

The ddot routine requires two arrays (XX and XY), the size of the arrays (n), and also the increment to step through each array (ix, iy). Normally, these are both equal to one, but it is possible to step through by different increment(s), in which case n is interpreted as the number of steps, and the arrays must be sufficiently large that the computation does not exceed the array bound(s).

Note that the integer values are passed by using `Ref()`:

```
# Test it out
julia> aa = [rand() for i = 1:1000]
julia> bb = [rand() for i = 1:1000]
julia> dotprod(aa,bb)
250.52877113928432. # Should be ~ 250.0
```

The equivalent source using the macro is as per the previous instance, except for a change to the statement:

```
prod = @ccall "libblas".ddot( n::Ref{Int32},
    XX::Ptr{Float64}, ix::Ref{Int32},
    YY::Ptr{Float64}, iy::Ref{Int32})::Float64
```

We can see that the values of n, ix, and iy are computed by the macro and passed using `Ref()`, whereas the XX and XY arrays are passed as is using `Ptr()`.

## Basel and Horner functions in C

In *Chapter 1*, we looked at a couple of examples of functions written in Julia:

```
basel: To compute the sum of 1/(x^2) to a given number of terms (N)
horner: To evaluate a polynomial for specific values and an array of
coefficients
```

Although the book is about Julia and not C, the code in these cases is straightforward and the reader should be able to make sense of the procedures, even if they had not previously been versed in coding in C.

First, consider implementing the Basel series evaluation in C.

We pass an integer from Julia, which will be a `Clong` type (that is, 64-bit) by default, and return a 64-bit (real) value for the sum, which will be a `Cdouble` type:

```
#include<stdio.h>
#include<stdlib.h>
double basel(int N) {
    double s = 0.0L;
    int i;
    double x;
    if (N < 1) return s;
```

```

    for (i = 1; i <= N; i++) {
        x = 1.0L/((double) i);
        s += x*x;
    }
    return s;
}

```

For the Horner routine, there is a scalar value ( $x$ ) and an array of coefficients ( $aa$ ), which are all 64-bit floats (that is, `Cdouble` types), plus the array size ( $n$ ), an `Int64 Clong` type:

```

#include<stdio.h>
#include<math.h>
double horner(double x, double aa[], long n) {
    long i;
    double s = aa[n-1];
    if (n > 1) {
        for (i = n-2; i >= 0; i--) {
            s = s*x + aa[i];
        }
    }
    return s;
}

```

Both routines are included in the `myfuncs.c` file, and we can build a shared library that contains both, together with the timing routines listed previously.

The command to create a library on macOS is this:

```
$> gcc -shared -fPIC -o mylib.dylib myfuncs.c
```

It should be the same on Linux and Windows (*assuming the use of MinGW*), except that the extension for the shared library will be `.so` and `.dll` respectively.

The `-fPIC` switch is required so that the compiler creates position-independent code.

The `-o` switch creates a library with the name `mylib.dylib`. Normally, the linker would expect to find it in a standard folder that keeps other system libraries.

Since this requires “root” access, it may be necessary to use a non-privileged directory and modify the `DYLD_LIBRARY_PATH` environment variable (on macOS) or to fully qualify the library specification.

A convenient shorthand is to prefix the library reference with `./`, which is resolved by the command shell to be the current directory.

The nm instruction can be used to list entry points (globals) in the library:

```
$> nm mylib.dylib
0000000000008008 d __dyld_private
0000000000003e80 T _basel
0000000000003e40 T _ctime
0000000000003f10 T _horner
0000000000003e70 T _systime
    U __time
    U dyld_stub_binder
```

And we can see the entry points for the four functions marked by T.

Now, we can use `ccall` to run these functions in our library, in a similar fashion to any other utility library.

This is quite a convenient fashion to create an interaction between Julia and C. There is no need for a complex API, and the call reduces to a single native instruction, so there is no additional overloading to the machine code.

Here is a run of the Basel function, first just for values of 1000 terms and then to get a reasonable timing for 1\_000\_000 terms:

```
julia> ccall(:basel,"mylib"), Float64, (Int64,), 1000)
1.6349339668472596
# The actual value of the sum is  $\pi^2 / 6$ ; that is, 1.64493:
julia> @elapsed ccall(:basel,"./mylib"),Float64,(Int64,),1000000)
0.191911357
```

The second function can be run to evaluate a polynomial using Horner's rule.

As you might expect, we get the same values as previously using the Julia version:

```
# Using Cdouble, Clong rather than Float64, Int64 etc.
julia> x = 2.1;
julia> aa = [1.0, 2.0, 3.0, 4.0, 5.0];
julia> ccall(:horner,"./mylib.dylib"),Cdouble,
        (Cdouble, Ptr{Cdouble}, Clong), x, aa, length(aa))
152.71450000000002
```

Interfacing with C functions is plainly quite straightforward, as indeed is the case with functions written in Fortran, except that the Fortran argument passing method makes the calls, in this case, a little more involved.

This is not the case when interfacing with C++ routines, as we will see in the next section.

## C++

Support for C++ was started over 8 years ago (around 2015) with a `Cxx.jl` package; it was not an easy package to use and was no longer supported after Julia v 1.3.

There are currently (as of 2023) a set of packages tackling interoperation with C++, each with a separate approach. We will highlight four of these next:

- `CxxWrap`: Now supported by the *JuliaInterop* group. Writes code for the Julia wrapper in C++, and then uses a one-liner on the Julia side to make the wrapped C++ library available there. It requires a C++ compiler installed that supports, at least, the C++17 standard.
- `CxxInterface`: Written by Erik Schnetter. The design is simpler than `CxxWrap.jl`. Wrapper functions are written in Julia and generate respective C++ wrapper functions. It is not well documented, so needs piecing together from the included examples.
- `jluna`: Written by Clem Cords. Relatively recent, comprehensive, and well documented (<https://clemens-cords.com/jluna/basics.html>). It requires a C++20 standard compiler but looks interesting for the future.
- `CxxCall`: To quote the author, Jan Weidner, `CxxCall.jl` is “*syntactic sugar*,” sitting on top of `CxxInterface.jl`, which does the actual work. Relatively easy to use as it aims to simplify the `CxxInterface` approach.

So, we will look at an example using the `CxxCall` package to implement the `mad()` function (that is, which computes the expression  $a*b + c$ ); the Julia and C versions have both been met before.

The calculation is relatively trivial, obviously, so the following serves mainly as an illustration of the build process:

```
# A Mad Example: Function Overloading in C++
#
filename = "MyMad.cxx"
lib = "libMyMad"
func = :mymad
using CxxCall
```

This is standard to call a setup function and add the C++ header(s) required.

```
eval(cxxsetup())
eval(cxxnewfile(filename,
"""
#include <iostream>
using namespace std;
""")

```

After that we define the actual C++ code using the @cxx macro.

```
@cxx lib function
    $func(a::Clong, b::Clong, c::Clong)::Clong
    """
    return a*b + c;
    """
end
# Overload the our function for real arguments (float64)
@cxx lib function
    $func(a::Cdouble, b::Cdouble, c::Cdouble)::Cdouble
    """
    return a*b + c;
    """
end
# And then create cxx file
cxx_write_code!()
```

Upon the execution of this code, a `MyMad.cxx` file will be created.

The reader can view their own; however, the file is included with the source accompanying this chapter and must be used for linkage if not using OS X, to generate the appropriate shared library.

Switching to the shell, a `libMyMad.dylib` shared library can now be created:

```
$> g++ -fpic -shared MyMad.cxx -o libMyMad.dylib
```

On macOS, `clang`, `gcc`, and `g++` are normally all the same LLVM compiler, the aliases being set up via the installation of the Xtools app, but it is possible to install the actual GNU compiler if you wish; however, to little advantage:

```
#=
Use julia's Libdl package and create pointer to the shared library
=#
julia> using Libdl

julia> dlopen(lib)      # lib => libMyMad.dylib
Ptr{Nothing} @0x00007f800ee4dcd0
# ... and test the c++ function(s) in Julia
julia> mymad(2, 3, 5)
11
```

```
julia> mymad(2.1, 3.2, 5.6)
12.32
```

Or, possibly, you could use these equally verbose calls:

```
julia> mymad(Clong(2), Clong(3), Clong(5))
julia> myadd(Cdouble(2.1), Cdouble(3.2), Cdouble(5.5))
```

However, the C++ overloading is not as flexible as multiple dispatch in Julia:

```
julia> mymad(2//5,3//7,5//13)
ERROR: MethodError: no method matching mymad(::Rational{Int64},
::Rational{Int64}, ::Rational{Int64})
# We will need to do the conversion explicitly
julia> mymad(Float64(2//5),Float64(3//7),Float64(5//13))
0.5560439560439561
```

Plus, we can't mix the argument types; even though there are two versions, these must be all integers and/or all floats:

```
julia> mymad(2.1, 3, 5.6)
ERROR: MethodError: no method matching mymad(::Float64, ::Int64,
::Float64)
```

Leaving interfacing to C and related languages, let's look at a selection of other programming languages for which specific Julia modules have been implemented.

## Python, R, and Java

A Julia/Python interface has been supported since the early version of Julia but only to the work of Steven G. Johnson, of MIT, on the PyCall module. Further, this was used in creating the PyPlot visualization package, which is a wrapper around the Python `matplotlib` package.

We will see this already in this book and will be discussing it in more detail in the chapter on Graphics. Also, Steven Johnson wrote a kernel interface to the IPython IDE (viz. IJulia), and the work between the IPython and Julia teams led to a new version of the former, Jupyter, in which the code examples accompanying this book have been distributed.

These modules are now supported by the *JuliaPy* community group, and in addition, there is a “reverse” package, PyJulia, which permits Julia to be called from Python; I will describe that briefly shortly.

In addition, there are some wrapper packages, notably those that implement the Pandas, scikit-learn, Seaborn, and SymPy modules. I will look at the last of these later in this chapter.

Other languages are supported by an alternative community group, *JuliaInterop*. R and Java are up to date, but others such as MATLAB, Mathematica, and C++ (that is, Cxx), at the time of writing, have not been touched for a long time and are certainly not version 1.0 compliant.

*JuliaInterop* also supports an interface to ZeroMQ, described as an opinionated, lightweight, blazing-fast messaging library, which is often employed for language-to-language communications and is covered in *Chapter 11*.

## Python

As already mentioned, the basis of the Python interface is the PyCall module. This package provides the ability to import arbitrary Python modules from Julia, call Python functions, define Python classes from Julia methods, and share large data structures between Julia and Python without copying them.

Also, it provides automatic conversion of types between Julia and Python and switching between one- and zero-based indexing of separate languages.

It can be called directly via the use of the `@pyimport` macro, which will import Python packages into Julia.

The following example creates a link to the NumPy package and uses the `rand()` routine, from the Python package and not a Julia one:

```
julia> using PyCall
julia> @pyimport numpy.random as nr
julia> aa = nr.rand(4,5)
# aa is a Julia array generated by Numpy
4×5 Array{Float64,2}:
 0.639591  0.654699  0.58607   0.235346  0.648967
 0.250739  0.793587  0.108476  0.836333  0.837293
 0.469522  0.774591  0.980525  0.555424  0.569421
 0.89431   0.947623  0.004079  0.120476  0.40064
```

The `aa` array is now in Julia, and we can create a slice that has no overhead in Julia:

```
julia> aa[2:3,2:3]
2×2 Array{Float64,2}:
 0.793587  0.108476
 0.774591  0.980525
```

Next, let us look at utilizing another well-known Python package, SciPy.

SciPy is large, with many sub-packages for various disciplines to apply to scientific problems. As in the previous example, `@pyimport` can be used to set up an alias to a sub-package, and the routines contained in the sub-package run via the alias.

The example that follows is part of the `optimize` package and uses a routine to find the root of the  $x \cos(x)$  function over the  $[1, \pi]$  interval, and therefore must have been added to Python in order to be imported:

```
julia> @pyimport scipy.optimize as so
julia> so.ridder(x -> x*cos(x), 1, pi)
1.5707963267958964
```

A second example uses the `integrate` sub-package to find a quadrature (that is, an integral) for the  $x \sin(x)$  function over the same interval:

```
julia> @pyimport scipy.integrate as si
julia> si.quad(x -> x*sin(x), 1, pi)
(2.840423974650036, 3.153504096353772e-14)
```

Finally, we note that it is possible to use `PyCall` directly to produce graphs, but this is done by importing `matplotlib` rather than using the `PyPlot` wrapper package.

Again, `matplotlib` must be present as a module in Python, which is true for the Anaconda package, but not necessarily for a vanilla installation:

```
julia> @pyimport matplotlib.pyplot as plt
julia> x = range(0, stop=10, length=1000);
julia> y = sin.(3*x + 4*cos.(2*x));
julia> plt.plot(x, y, color="red",
                linewidth=2.0, linestyle="--")
julia> plt.show()
```

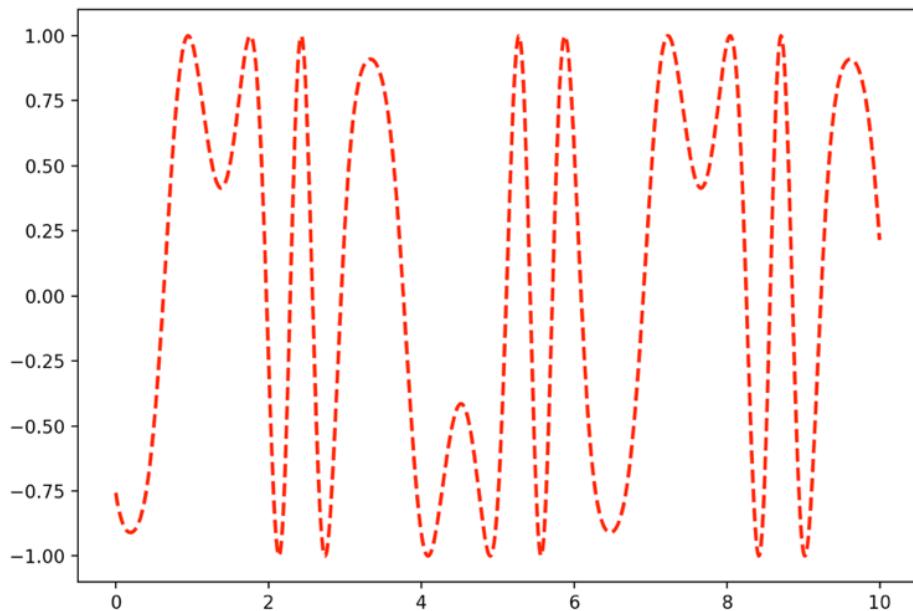


Figure 5.1 – Graph generated using Python’s matplotlib package

## Going the other way

In the case of Python, it is relatively easy to reverse the process and access Julia.

The package to use is called `pyjulia.jl` and is maintained by the *JuliaPy* group.

A detailed discussion is outside the scope of this book, but the following should be instructive for those who are interested:

```
# Get the code from github
$> git clone https://github.com/JuliaPy/pyjulia
# Check that python is installed
$> which python
/usr/bin/python
# Setup the Julia module
$> cd pyjulia
$> python setup.py install
```

This is all that is required, and we can now test out the Julia module.

The example I have provided is from what is known as the *low-level* interface:

```
$> python3
>>> from julia import Julia
>>> jl = Julia()
>>> pi = jl.pi      # Pickup the Julia built constant PI
>>> jl.sin(0.25*pi) # Now this works!
0.7071067811865475 # => 1/sqrt(2)
```

For more information and examples, I refer the reader to look at the web page and, in particular, to read about the high-level interface.

## Packages with Python wrappings

As mentioned previously, there are several packages based on PyCall that interface with Python packages rather than emulating them *natively*.

The advantage of this approach is that it is relatively easy to do and exposes a wealth of stable, powerful routines. The disadvantage is that the language (Python) and associated packages need to be present and accessible.

As a rule, it is best to use the Anaconda distro for Python as this installs a large number of popular packages, including most of the ones needed for Julia wrappers. Any additional packages can be added using the Conda package manager.

### Note

If you are using the Jupyter IDE to tackle the code in this book, then most likely you will have Anaconda installed already.

Apart from PyPlot, which has been mentioned, there are the Pandas, Seaborn, and SymPy packages. I am going to have a quick look at the last of these here. SymPy works with a special type: the symbol (Sym).

Calls to `sympy.symbols` produce symbolic variables and symbolic functions, and note the following about symbol definitions:

- Both `sympy.symbols` and `sympy.Symbol` allow the construction of symbolic variables and functions and in Julia, the `symbols` function (say) is an alias for `sympy.symbols`. A simpler form is encapsulated by the `@syms` macro.
- Assumptions on variables can be specified at creation, such as `x=symbols("x", real = true, positive=true)`.
- Multiple symbols can be created in a single statement:
- `(x, y) = symbols("x y")` or, alternatively, `@syms x, y`.

The following code defines a set of symbols; notice that it is possible to place restrictions on a symbol, such as only taking integer values, being positive, and/or both:

```
julia> using SymPy
julia> z = symbols("z")
julia> typeof(z)
Sym
```

*Note:* the *type* is *Sym*, not *Symbol*.

With one of the symbols already defined, we can solve some algebraic equations; for example, u:

```
julia> solve(z^2 + 1)
2-element Vector{Sym}:
-i
 i
julia> solve(z^2 + 1) |> print
Sym[-I, I]
```

In the REPL, SymPy does the best it can, but the output is not as easy to read and even harder to reproduce in book form. So, in general, I will redirect the output (`| >`) to print but the reader is urged to have SymPy pretty print its results.

A Jupyter notebook is provided (`SymPy.ipynb`) accompanying the code to this chapter with the examples fully worked (that is omitting the `| > print` statements).

Here is a much more complex polynomial equation and its solution:

```
p = (z - 3)^2*(z - 2)*(z - 1)*z*(z + 1)*(z^2 + z + 1)
solve(p) |> print
Sym[-1, 0, 1, 2, 3, -1/2 - sqrt(3)*I/2, -1/2 + sqrt(3)*I/2]
```

The next expression generates a product of a set of sine waves:

```
p = expand(prod([sin(x^(-i)) for i in 1.0:1.0:5.0]))
|> print sum(x^(-5.0))*sin(x^(-4.0))*sin(x^(-3.0))*sin(x^(-2.0))
    *sin(x^(-1.0))
x = symbols("x")
p = expand(prod([sin(x^(-i)) for i in 1.0:1.0:5.0]))
|> print
    sin(x-5.0)*sin(x-4.0)*sin(x-3.0)*sin(x-2.0)*sin(1/x)
```

... and this can be plotted for (say)  $x \sim [0.0, 1.0]$ :

```
using Plots
plotly()
plot(p, 0.0, 1.0)
```

Notice that although  $p$  is of type `Sym`, the Plots API can handle it *as is*; that is, without translation being necessary, as this code and *Figure 5.2* show:

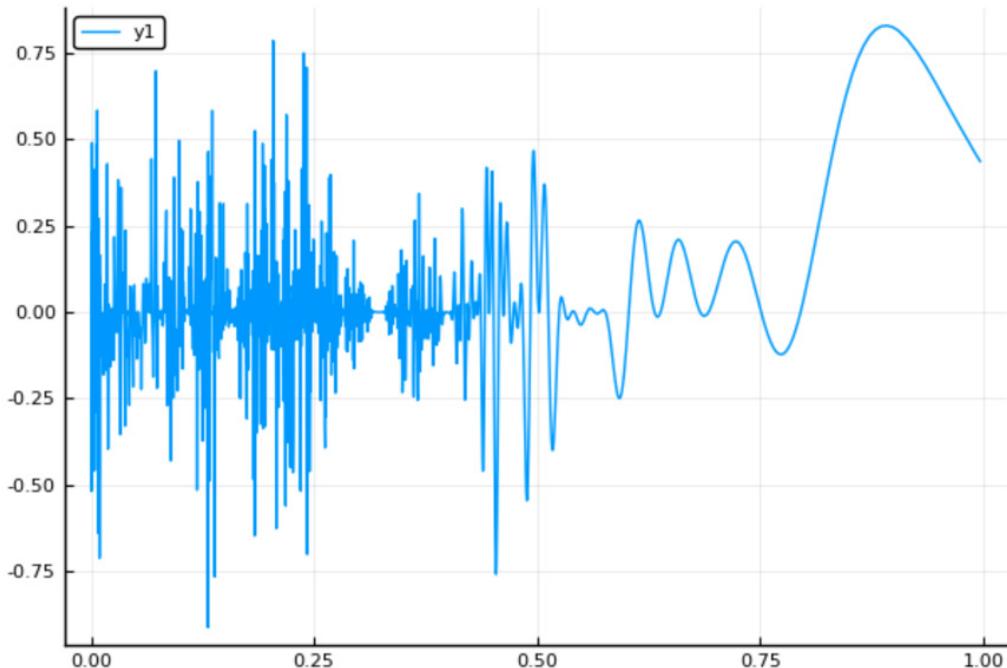


Figure 5.2 – Plot of function generated by the SymPy package

The SymPy package can do much more than what was just described and includes specialist sub-packages on physics, optics, and so on.

Other Julia packages wrapping native Python ones are available, such as the popular Pandas library, which is (also) mirrored, in Julia, by its own `DataFrame` package that we have met already and will discuss in more detail in the next chapter.

Also, there is a wrapper around the Seaborn graphics library, which will be touched upon in *Chapter 8* on visualization.

## The R (language)

In the previous edition, interfacing with R was via a package called `Rif.jl`, which was quite cumbersome to use and necessitated building a version of R as a shared executable.

Since then, the introduction of `RCall` has greatly simplified and extended the usability of R when called by Julia.

`RCall` provides multiple ways of interacting with R:

- R REPL mode
- `@rput` and `@rget` macros
- `R ""` string macro
- RCall API: `reval`, `rcall`, `rcopy`, `robject`, and so on
- `@rlibrary` and `@rimport` macros

We will briefly look at each of these.

### R REPL mode

The R REPL mode allows real-time switching between the Julia prompt and the R prompt. After a `using RCall` statement, keying \$ will switch to the R REPL mode, and the R prompt will be shown.

*Backspace* is used to leave the R REPL mode, in the same way as for the help system and package manager.

In the following snippet, the `mtcars` dataset is loaded on the Julia side from `RDatasets`, then, switching to the R REPL mode, by typing \$:

```
julia> using RCall, RDatasets
julia> mtcars = dataset("datasets", "mtcars")
```

	Model	MPG	Cyl	Disp	HP	DRat	WT	QSec	VS	AM	Gear	Carb
1	Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
2	Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
3	Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
4	Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
5	Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
6	Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
7	Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
8	Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
9	Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
10	Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
11	Merc 280C	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
12	Merc 450SE	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3

We can load the `ggplot2` library on the R side and then create a graph from the dataset by referencing it as `$mtcars`.

The example shown is that of miles per gallon versus car weight:

```
R> library("ggplot2")
R> ggplot($mtcars, aes(x=WT,y=MPG)) + geom_point()
```

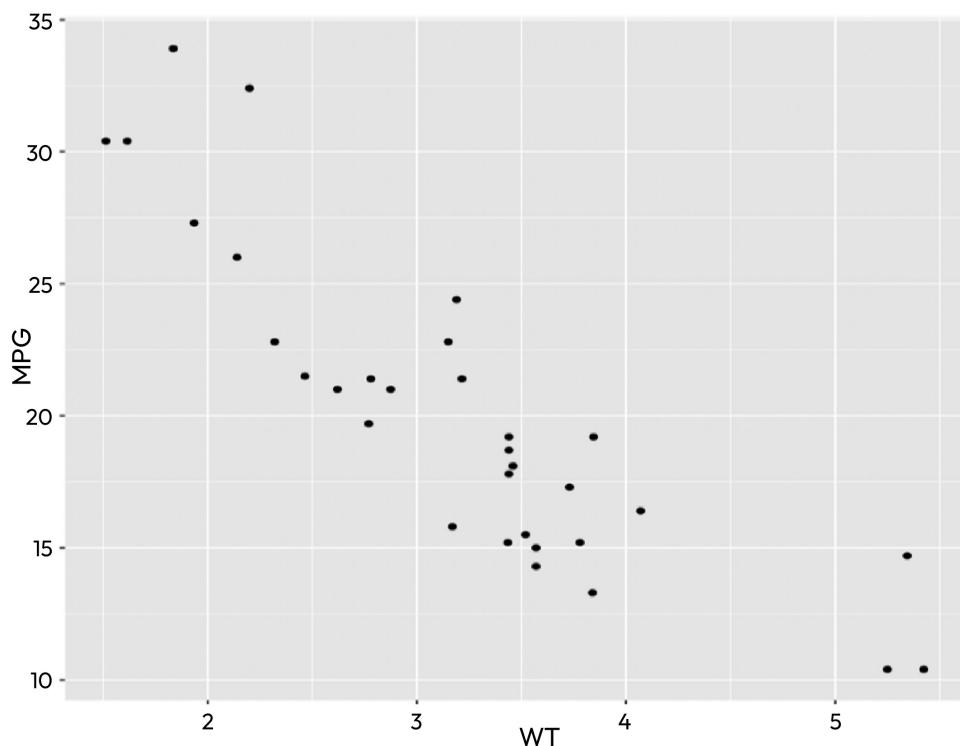


Figure 5.3 – R plot using the `ggplot` library of the miles per gallon versus weight of the `mtcars` dataset

### ***Exchanging variables with @rget and @rput***

Referencing a Julia variable on the R side can be done by using the `@rput` macro; this can be applied to arrays as well as scalar variables:

```
julia> aa = randn(5);
julia> @rput aa;
R> aa
[1] 0.9830668  0.1355977 -0.6660837 -0.4434384  0.6045229
```

The same is true for other structures of the @rput mtcars from the first example, resulting in the following on the R side.

The alternate is possible too, using the @rget macro:

```
R> bb = c(1,1,2,3,5,8,13)
julia> @rget bb;
# [ Show as the transpose to save space ]
#
julia> bb'
1x7 adjoint(::Vector{Float64}) with eltype Float64:
 1.0  1.0  2.0  3.0  5.0  8.0  13.0
```

*Note:* The @rget macro brings the values across as floats rather than integers.

### ***Using the R”...” string macro***

The R”...” string macro does not require flipping to R REPL mode, In the following example, we generate a set of 1000 normally distributed variates and perform a Student’s t-test. We expect the distribution to have a zero mean and unit variance, so the null hypothesis that  $\mu \sim 0$  should be true:

```
julia> using RCall
julia> x = randn(1000);
julia> R"t.test(\$x)"
RObject{VecSxp}
One Sample t-test
data: `#JL`$x
t = -1.1056, df = 999, p-value = 0.2692
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -0.09713193  0.02712335
sample estimates:
 mean of x
-0.03500429
```

R has a number of ways of optimizing functions, called from the optim routine and specifying which algorithm to use.

These are based on the Nelder-Mead, quasi-Newton, and conjugate-gradient algorithms.

The next example defines a non-linear function, applying one such as the **Broyden–Fletcher–Goldfarb–Shanno (BFGS)** method:

```
julia> f(x) = 10*sin(0.3*x)*sin(1.3*x^2) + 0.00001*x^4 + 0.2*x + 40;
julia> R"optim(0, \$x -> f(x)), method='BFGS')"
```

```
RObject{VecSxp}
$par[1] -6.685958
$value[1] 29.61426
$counts
  function gradient
    28          8
$cconvergence
[1] 0
$message
NULL
```

Finally, in the section, we will use R to read some financial data from the CSV file provided with the chapter's code, creating a data frame of price, date, and ticker (that is, stock code).

The prices for separate stocks are normalized against the first value:

```
julia> pwd()
"/Users/malcolm/MJ2/Chp05/Files"
R"""
library(data.table)
library(scales)
library(ggplot2)
link <- "fin_data.csv"
dt <- data.table(read.csv(link))
dt[, date := as.Date(date)]
# create indexed values
dt[, idx_price := price/price[1], by = ticker]
"""
```

And we can view the data table on the R side:

```
julia> R"dt"
RObject{VecSxp}
  date ticker      price idx_price
 1: 2000-01-03    IBM 90.66385 1.0000000
 2: 2000-02-01    IBM 87.95363 0.9701069
 3: 2000-03-01    IBM 86.64813 0.9557076
 4: 2000-04-03    IBM 88.68945 0.9782228
 5: 2000-05-01    IBM 83.74264 0.9236607
  --
532: 2016-01-04   JPM 58.04395 1.9229179
533: 2016-02-01   JPM 56.71510 1.8788951
534: 2016-03-01   JPM 59.04963 1.9562350
535: 2016-04-01   JPM 61.44905 2.0357244
536: 2016-05-02   JPM 61.84250 2.0487589
```

## The R API

First, we use \$ to get the switch to R REPL mode and then generate an array of 100 random numbers in the [0.0, 1.0] interval:

```
R> rr <- runif(100,0.0,1.0)
```

We could bring this over to Julia using the evaluate @rget macro.

Alternatively, we can use the `reval()` function to evaluate the array in the Julia context:

```
julia> ra = reval("rr")
RObject{RealSxp}
[1] 0.069845643 0.807724489 0.812904286 0.372850300
[97] 0.088333914 0.029721416 0.144311317 0.655515486
```

Notice that this is still an R object, so we have to apply R functions to it, not Julia functions, which we do with `rcall()`, passing the routine name as a symbol:

```
julia> rcall(:mean,ra)
RObject{RealSxp}
[1] 0.4931235
```

Alternatively, this can be copied to a Julia object with `rcopy()`, and now, the Julia routine is used:

```
#=
To use the mean in Julia we need to import the Statistics package
=#
julia> using Statistics
julia> rb = rcopy(R"rr"); mean(rb)
0.49312349389307203
```

## @rlibrary and @rimport macros

In this section, we consider the use of the `@rlibrary` macro to create a reference to R packages. One popular one is **Modern Applied Statistics with S (MASS)**, which contains a range of statistical routines and accompanying datasets.

The final “S” refers to the fact that R is a clone of the S+ commercial production.

Here, we will use `@rlibrary` to access MASS and then `rcopy()` to get some data about the Old Faithful geyser in Yellowstone Park. This is a simple data frame with a couple of columns referring to the duration of and time between eruptions (in minutes) recorded between August 1 and August 15, 1985:

```
@rlibrary MASS
geyser = rcopy(R"MASS::geyser")
299x2 DataFrame
```

```

Row  waiting duration
  Float64  Float64
1  80.0  4.01667
2  71.0  2.15
3  57.0  4.0
4  80.0  4.0
5  75.0  4.0
.....

```

This has been copied to a Julia Data Frame, and we can estimate the ratio of the time between eruptions to their duration (on the Julia side):

```

round(sum(geyser[!, :waiting]) /
      sum(geyser[!, :duration]), digits=5)
20.8952

```

Also, we can use the R `summary` routine to calculate statistics (on the R side):

```

julia> rcall(:summary, geyser[!, :waiting])
RObject{RealSxp}
Min. 1st Qu. Median Mean 3rd Qu. Max.
43.00 59.00 76.00 72.31 83.00 108.00
julia> rcall(:summary, geyser[!, :duration])
RObject{RealSxp}
Min. 1st Qu. Median Mean 3rd Qu. Max.
0.8333 2.0000 4.0000 3.4608 4.3833 5.4500

```

Note that *BOTH* datasets were denoted as geysers but are different.

Import the base package as an alias. This exists in all R runtimes:

```

julia> @rimport base as rbase
julia> ra
RObject{RealSxp}
[1] 0.069845643 0.807724489 0.812904286 0.372850300
... ...
[97] 0.088333914 0.029721416 0.144311317 0.655515486
julia> rbase.sum(ra)
RObject{RealSxp}
[1] 49.31235
julia> rbase.summary(ra)
RObject{RealSxp}

```

Min.	1st Qu.	Median.	Mean	3rd Qu.	Max.
0.0051	0.2836	0.4761	0.4931	0.7503	0.9915

## Java

The interface to Java is via the `JavaCall` package.

The first call is to initialize the runtime; it is possible to pass additional switches to (say) specific the Java classpath:

```
# Initialise and ask for additional memory using JavaCall
julia> JavaCall.init(["-Xmx128M"])
```

*Notes:*

We can set the (say) current directory to be the classpath in the `JavaCall.init()` routine by an expression such as `["-Xmx128M -Djava.class.path=$(@DIR)"]`.

On macOS, there may be a segmentation fault (-11), but this does not halt a Jupyter notebook or the REPL.

See <http://juliainterop.github.io/JavaCall.jl/faq.html>.

In the REPL, Julia may need to be started with a `--handle-signals=no` option in order to disable Julia's signal handler.

*NB: This may cause issues with handling C in Julia.*

Static and instance methods with primitive or object arguments and return values are callable.

One-dimensional array arguments and return values are also supported.

Primitive, string, object, or array arguments are converted as required.

The following snippet converts a string in Julia to a `JString` value:

```
julia> a = JString("Hello, Blue Eyes")
JavaObject{Symbol("java.lang.String")}
(Ptr{Nothing} @0x00007fe4d3027c90)
# The JString as a single field (:ref)
julia> fieldnames(typeof(a))
(:ref,)
# The pointer can be used in a Java function
julia> b = JavaCall.JNI.GetStringUTFChars(Ptr(a),
                                         Ref{JavaCall.JNI.jboolean}())
Cstring(0x00007fc620761ae0)
# Check that 'b' is the original string
```

```
julia> unsafe_string(b)
"Hello, Blue Eyes"
```

In a similar fashion as with Python (PyCall), we can use the JVM (JavaCall) to perform some mathematics:

```
# Import Java Math classes and don't need the explicit call
julia> jlm = @jimport "java.lang.Math"

# Calculate the function exp(pi^2 / 6)
julia> jcall(jlm,"exp",jdouble,(jdouble,),pi*pi/6.0)
5.180668317897116
```

Finally, let's import the `ArrayList` class, define an instance of it, and add some items:

```
julia> JArrayList = @jimport (java.util.ArrayList)
julia> a = JArrayList()
julia> jcall(a,"add",jboolean,(JObject,),"'Twas ");
julia> jcall(a,"add",jboolean,(JObject,),"brillig, ");
julia> jcall(a,"add",jboolean,(JObject,),"and ");
julia> jcall(a,"add",jboolean,(JObject,),"the ");
julia> jcall(a,"add",jboolean,(JObject,),"slithy ");
julia> jcall(a,"add",jboolean,(JObject,),"toves,");
```

Now iterate thru' the array and push it on a Julia array converting the bit type to an (unsafe) string.

```
julia> t = Array{Any, 1}()
julia> for i in JavaCall.iterator(a)
         push!(t, unsafe_string(i))
     end
# Evaluate the result.
julia> join(t)
"'Twas brillig, and the slithy toves,"
```

After looking at how Julia modules have been written for a small selection of languages, we will turn to an approach to interface with others using the operating system and command shell in a more general fashion.

It should be noted that the discussion here applies mainly to the Linux and OS X variants, but similar methods can be applied under Windows, especially with Unix-type frameworks such as MinGW.

## Working with the OS and pipelines

Up to now, we have been discussing ways for Julia to operate with other programming languages.

For C (and Fortran), this was relatively straightforward, as Julia was designed with a mechanism to interface directly with shared libraries (aka DLLs on Windows). So, any system that effectively operates via a shared library is immediately available to Julia.

This means that graphics frameworks, **database management systems (DBMS)**, and so on can all be made (more or less) easily accessible, and several successful packages have been implemented whose code depends on (and requires the installation of) third-party libraries.

These have been termed *wrapper* packages, as opposed to “native” ones, written purely in Julia. In practice, many packages are often a combination of both paradigms rather than being principally one or the other.

For some other languages, notably Python, R, and Java, effort was expended on creating usable interface packages, and in the case of Python, these have led to additional packages wrapping around some of Python’s better-known ones. At present, the list, which included Matlab, Mathematica, and C++, has been reduced, but perhaps that work will be taken up again in the future.

For other languages such as Perl, Ruby, and so on, an alternative approach can be employed involving spawning command-line tasks.

### Running commands

We have seen that Julia is able to communicate with utilities in the underlying operating system. In fact, languages that can execute from a command line are no different from “standard” utilities and can be used by Julia by introducing a few additional features.

#### Note

Although task spawning and pipelining are possible in Windows, they exist very much in the spirit of Unix-based OSs, such as Linux and OS X, and the examples here have been run on a MacPro running macOS 11.7.

As a first example, we will use `curl` (or `wget`) to retrieve the `mastering-julia.html` web page, which we have done a couple of times previously.

First, check that `curl` is available; otherwise, you will need to install it:

```
# Notice that backticks demarcate the command.  
julia> cmd = `which curl`  
julia> typeof(cmd)  
Cmd
```

In the early version of Julia, commands execute immediately, but now, a `Cmd` object is returned and must be run explicitly.

Since commands run as separate tasks, it is usually preferable to suppress the output of the `run()` function (by terminating with `;`).

The operating system provides three pre-opened channels for input (`stdin`), output (`stdout`), and errors (`stderr`). The task output will go to `stdout` (by default) but can be captured and used by the Julia program:

```
julia> run(cmd)
/usr/bin/curl
Process(`which curl`, ProcessExited(0))
```

Since the command indicates a location, the command exists and is on the executable path. We can use `curl` to get the web page and can do it in one step.

Here is an “under construction” HTML index file from one of my websites:

```
proc = run(`curl "http://mj2.website/uc/index.html"`);
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Under Construction</title>
    <style>
img.display {
    display: block;
    margin-left: auto;
    margin-right: auto;
}
</style>
</head>
<body bgcolor="#ffffff">
    
</body>
</html>
```

Look at the `process.jl` file in Base for a definition of a process in Julia. The struct is mutable since (clearly) some of the fields (viz. exit codes and terminal signals, and so on), need to be modified:

```
#=
This has several useful fields including the command to run,
references to the STDIN, STDOUT and STDERR channels and the process
exit code.

mutable struct Process <: AbstractPipe
    cmd::Cmd
    handle::Ptr{Cvoid}
    in::IO
    out::IO
    err::IO
    exitcode::Int64
    termsignal::Int32
    exitnotify::Condition
    closenotify::Condition
end
=#
# For the previous process
julia> proc.exitcode
0          # Zero is the normal non-error exit status
```

We can use the `less()` function as a conventional way to view a file:

```
#=
First set an environment variable to point to the Julia shared folder.
=#
ENV["JULIA_SHARE"] =
    string(Sys.BINDIR[1:end-4], "/share/julia");

# . . . and view the file
less(ENV["JULIA_SHARE"] * "/base/process.jl")
```

### Note

A macro called `@less` can be used to display functions in Base that “knows” their location, so the `JULIA_SHARE` environment variable is not needed.

The functions have to be fully qualified; that is, with arguments specified.

As an example, try `@less sin(2.3)`.

## Text processing and pipes

Accompanying this book, there is a folder called `Alice` containing poems by Lewis Carroll from the *Alice* books and some others.

I will use these here to demonstrate executing some code snippets; the directory in the snippets needs to be set according to where the folder is located:

```
# Where the Alice text files are stored on my computer
#
julia> cd(ENV["HOME"]*"/MJ2/Alice"); pwd()
"/Users/malcolm/MJ2/Alice"
```

The Unix `wc` word count program is common to OS X and Linux, but a Windows system will need to have a POSIX-compliant shell installed on the executable path.

As with many Unix utilities, it is a misnomer since the output is (by default) a count of the lines, words, *AND* characters in a text file.

We can pass a filename using the usual `$` convention and should check it is a “plain” file and not a binary by defining a Julia function:

```
#=

This will produce output only if the textfile exists.
The trailing ';' will suppress the output from Julia, since wc runs a
separate task and writes its output to STDOUT
=#
julia> wc(f) = isfile(f) && run(`wc $f`);
```

We will count the number of occurrences of “beaver” in Lewis Carroll’s poem *The Hunting of the Snark* in the `Alice` folder and redirect the output to a text file as part of a `pipeline()` call:

```
# Assume we are in the correct folder ...
# ... otherwise the files need to be fully specified
julia> txtfile = "hunting-the-snark.txt";
julia> logfile = "hunting-the-snark.log";
```

Run a command using the Unix utility `grep` to search for lines with `beaver` in them the `-i` switch will ignore case:

```
julia> io = open("logfile", "w");
julia> run(`grep -i beaver $txtfile`, (stdin, io, stderr));
```

To save the output, the `run` command can have a second argument, which is a tuple of the `stdin`, `stdout`, and `stderr` channels, and here, we replace the output channel with one created by opening the log file for writing (that is, creating) by specifying the `w` parameter:

```
# Apply the wc function to the log file.  
julia> wc(logfile);  
19 138 821 hunting-the-snark.log
```

This can be simplified by running the two commands together, without the need for an intermediate file, by using a `pipeline()` function:

```
julia> run(pipeline(`grep -i beaver $txtfile` , `wc`));  
19      138      821
```

We can repeat this for the text `bellman` but append the output from `grep` to the log file:

```
julia> run(`grep -i bellman $txtfile` ,  
        (stdin, open("logfile","w"), stderr));  
julia> wc(logfile);  
49 371 2202 hunting-the-snark.log
```

Of course, this output may not be accurate. Why? Because we may have double-counted any lines containing both `beaver` and `bellman`.

Note that in `grep`, we ignored case (`-i`) but will still find punctuation marks and plurals, such as (say) `Beavers.`, as the following snippet indicates:

```
julia> run(pipeline(`grep -bellman $txtfile` ,  
                    `grep -i beaver`));  
2523:He could only kill Beavers. The Bellman looked scared,  
# 2523 is the line number in the textfile
```

This is OK here but not great for syntactical text analysis, where we will want to work with a single case and first strip off any punctuation. I'll attend to this later.

Piping the preceding code through `wc` will give us the correction needed to the number of lines containing both `beaver` and `bellman`.

## Finding large files

I have included next a function that I find useful for finding any large files in a specific directory or attending its subdirectories. It is based on the use of three Unix utilities, which are all run in a pipe:

- `find`: This is used to get a listing of all files in a specified directory and then to execute a command ('`du`') to get the sizes

- **sort**: Takes the output from the `find` command and sorts on size
- **head**: Gets the output from the `sort` command and displays the first `nf` lines

Here is the code (listed in the `ftop.jl` file):

```
"""
Find the largest files in a specified directory.
Call by: ftop(dir,nf), both arguments are optional.
dir is the directory to search and nf is the number of files to
return.
Defaults are the current directory and top 10 files.
"""

function ftop(dir=". ", nf=10)
    @assert nf > 0
    ef = `find $dir -type f -iname "*" -exec du -sh "{}" + `
    try
        run(pipeline(ef, `sort -rh`, `head -$nf`))
    catch
        println("No files found.")
    end
end
```

In the file listing, the function has been preceded by a multiline string, delimited by triple quotes. Including this file in the REPL will add this string to the system to provide some help text on the function, triggered in the usual way by typing the following:

```
help> ftop
# Load the function from the Chapter 5 code folder.
julia> include("ftop.jl")
# and running this against my MJ2/Alice folder
julia> dd = ENV["HOME"]*"/MJ2/Alice"
julia> ftop(dd,5)
380K /Users/malcolm/MJ2/Alice/SB-Concluded.txt
348K /Users/malcolm/MJ2/Alice/Sylvie-Bruno.txt
148K /Users/malcolm/MJ2/Alice/Wonderland.txt
32K /Users/malcolm/MJ2/Alice/red-snark.txt
28K /Users/malcolm/MJ2/Alice/Snark-Hunt.txt
```

The way this processes command lines leaves a little to be required since it is necessary to define a directory in order to change the number of files.

There is a very useful package, `ArgParse`, which provides much more flexibility, including specifying `-` style switches and accompanying values. We will discuss this in the final chapter when discussing standalone scripts and executable programs.

## Perl one-liners

Perl has fallen out of fashion somewhat with the current popularity of Python but, in my opinion, remains one of the best languages for data munging.

Unix distros and OS X (normally) have Perl available, but on Windows, it needs to be installed and on the executable path.

Julia's performance in handling strings is not one of its greatest strengths, so where normal Unix utilities fall short, the processing of large files can be successfully done using Perl.

Julia introduced an analytical engine (JuliaDB) to tackle the processing and analysis of large datasets, which soon was relegated to unmaintained status. Much of its functionality has been incorporated in the `DataFrames` and `DTables` packages, which we will look at when discussing the processing of data files in more detail later in the book.

Here, I'll restrict myself to using Perl in creating some quite powerful one-liners that can be adapted for use of the `run()` command in the same sense as other OS utility commands.

For the more interested, there are a number of web pages describing Perl one-liners and also an excellent book by Peteris Krumins entitled *Perl One-Liners: 130 Programs That Get Things Done*, available on Amazon and elsewhere.

## A couple of examples

On OS X and Linux, there is a word list in the `/usr/share/dict` directory, and the following command outputs any palindromes of six or more letters. (I am not intending to discuss Perl in any great detail; I hope that most of the syntax is relatively easily followable):

```
julia> cmd = `perl -nle 'print if $_ eq reverse &&
           length > 5' /usr/share/dict/words`
julia> run(cmd)
deeded
degged
hallah
kakkak
murdrum
redder
```

```
repaper
retter
reviver
rotator
sooloos
tebbet
terret
```

To capture the output of the palindrome command, we will use the `stdout` argument for the tuple in the `run()` function as before.

This time, we will make a temporary file; notice that the call returns a two-element tuple consisting of the filename and an open I/O channel:

```
# Grab the name for a temporary file
julia> tmpfile = mktemp(tempdir())
("/var/folders/b0/tc3zlg_x0s304wbgnzvjn6_c0000gp/T/jl_40YmUm",
IOStream(<fd 20>))
# Set up the Perl command
julia> cmd =
`perl -nle 'print if $_ eq reverse && length > 5'
/usr/share/dict/words`
```

Run the command capturing the output on the second argument of the `tmpfile` tuple and then read the input using the `filename` argument and delete the temporary file.

```
julia> run(cmd, (stdin,tmpfile[2],stderr));
julia> dmp = read(tmpfile[1]);
julia> run(`rm -f "$tmpfile[1]"`); # Remove the tmp file
```

To process further, we will need to pass a byte array to a string; note the carriage returns (\n) delimit lines:

```
julia> ss = String(dmp)
deedeed\ndegged\nallah\nkakkak\nmurdrum\nredder\nrepaper\nretter\
reviver\nrotator\nsooloos\ntebbet\nterret\n
```

We need to remove the trailing carriage returns and then split them into words in order to return a string array:

```
julia> sa = split(chomp(ss), "\n")
julia> typeof(sa)
Array{SubString{String},1}
```

Perl is good (and quick) for processing strings and works well with large files.

The following example gets the top 10 words in *The Hunting of the Snark*, ignoring blank lines. I'll leave it to the reader to reason the function of the OS and Perl command(s):

```
julia> cd(ENV["HOME"] * "/MJ2/Alice")
julia> fl = "hunting-the-snark.txt";
julia> c1 = `perl -pne 'tr/[A-Z]/[a-z]/' $(fl)`;
julia> c2 = `perl -ne 'print join("\n",split(/\s+/, $_));
               print("\n"))'`;
julia> run(pipeline(c1,c2,`sort`, `grep -ve '^$'`,
                     `uniq -c`, `sort -rn`, `head -10`));
299 the 153 and 123 a 105 to 91 it 82 with 76 of 76 in 75 he 69 that
```

Let's look at the different instances of *Bellman* (ignoring the case):

```
julia> run(pipeline(c1,c2,`sort`, `grep -ve '^$'`,
                     `uniq -c`, `sort -rn`, `grep -i Bellman`));
25 bellman 4 bellman, 1 bellman's
```

As we suspected previously, there is a problem with punctuation marks. The problem is more obvious when we look at the bottom 10 words:

```
julia> run(pipeline(c1,c2,`sort`, `grep -ve '^$'`,
                     `uniq -c`, `sort -rn`, `tail -10`));
1 'fritter 1 'friends, 1 'for, 1 'dunce.') 1 'come, 1 'candle-ends,' 1
'but, 1 'be 1 'a 1 '-jum!'
```

We need to add an extra task in the pipeline; the `c2` command in the next snippet provides this:

```
julia> function munge(f1)
    c1 = `perl -pne 'tr/[A-Z]/[a-z]/' $(f1)`;
    c2 = `perl -pne 's/()[:punct:]/g'`;
    c3 = `perl -ne 'print join("\n",split(/\s+/, $_));
                  print("\n"))'`;
    c4 = `grep -ve '^$'`;
    read(pipeline(c1,c2,c3,`sort`,c4,`uniq -c`, `sort -rn`))
end
julia> text = munge("hunting-the-snark.txt");
julia> (eltype(text),length(text))
(UInt8, 15889)
julia> lines = split(String(text),"\n");
julia> n = length(lines)
1323
```

If we now look at the bottom 10 entries, the punctuation has been removed:

```
julia> s2 = [split(lines[i]) for i = n-9:n]
10-element Vector{Vector{SubString{String}}}:
["1", "aided"] ["1", "aghast"] ["1", "ages"]
["1", "affectionate"] ["1", "advice"] ["1", "additional"]
["1", "aboard"] ["1", "able"] ["1", "abetted"]
[]
```

We saw previously that the Julia process structures had a number of useful fields.

In the next section, I want to look at how to use I/O channels to capture the I/O.

## Using process I/O channels

To discuss using process I/O channels, we need to be in the `Alice` folder. Also in that folder is a `rev.pl` Perl script, whose function (as the name suggests) is to read its input stream, reverse the lines, and write them back to the output stream:

```
# For our example use the text of the Jabberwocky poem
julia> jabber = "jabberwocky.txt";
julia> proc = open(`./rev.pl $jabber`, "r+");
julia> proc.in
Base.PipeEndpoint(RawFD(26) open, 0 bytes waiting)

julia> close(proc.in);
```

We have closed the input stream, but the output is still open, so we can read the lines from it and display the reversed poem, not forgetting first to close the output stream too:

```
julia> poem = readlines(proc.out);
julia> close(proc.out);
julia> poem
34-element Vector{String}:
"sevot yhtils eht dna ,gillirb sawT"
":ebaw eht ni elbmig dna eryg did"
",sevogorob eht erew ysmim lla"
".ebargtuo shtar emom eht dnA"
"""
"!nos ym ,kcownrebbaj eht eraweB"
"!hctac taht swalc eht ,etib taht swaj ehT"
"nuhs dna ,drib bujbuj eht eraweB"
" '!hctansrednaB suoimurf ehT"
"""
":dnah ni drows laprov sih koot eH"
```

```

"-- thguos eh eof emoxnam eht emit gnoL"
",eert mutmuT eht yb eh detser os" :
"daeh sti htiw dna ,daed ti tfel eH"
".kcab gnihpmulag tnew eH"
"""

"?kcowrebbaJ eht nials uoht tsah dnA"
"!yob hsimaeb ym ,smra ym ot emoC"
"!yallaC !hoollaC !yad suojbarf ho"
".yoj sih ni deltrohc eH"
"""

"sevot yhtils eht dna ,gillirb sawT"
":ebaw eht ni elbmig dna eryg diD"
",sevogorob ehterew ysmim llA"
"ebargtuo shtar emom eht dnA"

```

This is yet a further way to capture and process the output stream.

## Interfacing with other languages

We are not limited to just Perl. For any scripting language that has a command-line option and can process standard input, streaming it to standard output will do.

Next are three examples which use a command line scripting approach.

### Perl 6

Perl 6 is now considered a new and separate language from Perl5; some see it as a reason for the decline in the latter since the syntax of the two differs markedly in some places. Perl 6, unlike Perl, is not distributed as a standard.

The favorite distribution for Perl 6 can be obtained from <https://rakudo.org>.

It is necessary to be able to “find” Perl 6; the command we saw earlier is a convenient method to set up a symbolic link to the binary and put it in a folder on the excuse path (on OS X, my link is `perl6 -> /Applications/Rakudo/bin/perl6`, and I put it in my '\$HOME/bin' folder):

```

# Check that perl6 is available
julia> run(`which perl6`);
/usr/local/bin/perl6

```

Perl 6 is (naturally) much richer syntactically than its predecessor.

Next, we will use it for the line of the greatest length in *The Hunting of the Snark*:

```

# Check we are still in the correct directory
julia> pwd()

```

```

"/Users/malcolm/MJ2/Alice"
julia> run(`perl6 -e 'my $mx="";
for (lines) {$mx = $_ if .chars > $mx.chars};
END { $mx.say }' hunting-the-snark.txt`);
'You must know ---' said the Judge: but the Snark exclaimed 'Fudge!'

```

## Ruby

Ruby is not as popular as it was a decade ago, again probably due to the current interest in Python. It is a pure **object-oriented (OO)** language and is distributed in OS X and Linux, one reason being that it is used in a quite few system maintenance procedures.

Originally, the Ruby language gained popularity when combined with its Rails package as a way of generating **Model-View-Controller (MVC)** frameworks, but seems to have lost its original popularity more recently.

Here is a simple Ruby command to reverse the opening title and first four-line stanza from the poem, *The Hunting of the Snark*:

```

# Define the Ruby command
julia> cmd = `ruby -e 'File.open("hunting-the-snark.txt").each_line {
|n| puts n.chop.reverse }'`;
# and run it through a pipe showing only the first 8 lines
julia> run(pipeline(cmd,`head -8`));
tsrif eht tiF
GNIDNAL EHT
,deirc namlleB eht '!kranS a rof ecalp eht tsuJ'
;erac htiw werc sih dednal eh sA
edit eht fo pot eht no nam hcae gnitroppus
.riah sih ni deniwtne regnif a yB

```

## Python

We could not leave this section without giving an example in Python, which uses the command line rather than bespoke Julia packages.

This time, let's switch the first four lines of the *Jabberwocky* poem and encode it in Base64:

```

julia> cd(ENV["HOME"]*"/MJ2/Alice")
julia> f1="jabber4.txt"
julia> f2="jabber4.b64"
julia> b64encode = `python -c 'import base64,sys;
base64.encode(open(sys.argv[1], "rb"),
open(sys.argv[2], "wb"))' $f1 $f2` 
julia> run(b64encode);

```

Now, check it by running this command:

```
julia> run('cat $f2');
J1R3YXMgYnJpbGxpZywgYW5kIHRoZSBzbGl0aHkgdG92ZXMKRG1kIGd5cmUgYW5kIGdpb
WJsZSBp
bib0aGUgd2F1ZT0KQWxsIG1pbXN5IHdlcmUgdGhlIGJvcm9nb3Z1cywKQW5kIHRoZSBtb
211IHJh
dGhzIG91dGdyYWJlLgo=
```

Now, reverse the process, this time using the `base64` command:

```
julia> run(`base64 --decode $f2`);
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe
```

## Other languages supported by the JuliaInterop group

We should close by mentioning a couple of other languages that can be used from Julia: Mathematica (via MathLink) and Matlab.

The `Mathlink.jl` package provides access to Mathematica/Wolfram Engine via the MathLink library and requires the presence of either Mathematica or the free Wolfram Engine system. It will attempt to find either one or the other at build time; otherwise, the installation will fail.

The main interface consists of a `W""` string macro for specifying symbols, very similar to one of the ways that the R language can be interfaced with Julia.

See the following documentation for examples: <https://github.com/JuliaInterop/MathLink.jl>

The `MATLAB.jl` package provides an interface for using MATLAB from Julia using the MATLAB C API. Matlab syntax is very similar to Julia and so should be familiar to Julia programmers. However, it is not possible to use `MATLAB.jl` without installing a purchased copy of MATLAB from MathWorks.

The Julia package provides the creation (and manipulation) of `mxArray` types, which are data structures used in MATLAB to represent arrays and other kinds of data.

There is also the possibility to communicate with MATLAB engine sessions via a `mcall` function interface, `mat ""` custom strings, and `@mput`, `@mget` macros, but support is patchy, and at the time of writing, problems are arising in using the latest version of the Matlab engine.

Again, see the GitHub source (<https://github.com/JuliaInterop/MATLAB.jl>) for documentation and examples.

## Working with the filesystem

Julia provides a variety of functions for reading folders and processing files natively, and we will meet some of these in the next chapter. However, as a tester, I have included a few examples here.

I got an error after exiting the following process, so had wrapped it in a `try/catch` block to suppress the output of the error – note that this was after the `run()` call had succeeded:

```
#=
Running the following gave a FAILED Process error on exit,
julia> cd(string(ENV["HOME"], "/MJ2/Chp05/Code05"));
julia> run(`wc $(readdir())`)
Wrapping the run() function it in a try/catch block will trap the
error.

=#
julia> try run(`wc $(readdir())`) catch end
 3424    11555   152017 Chp05.ipynb
      791     2796    20108 Chp05.jl
     1843    10493   67627 Chp05.md
wc: Scripts: read: Is a directory
wc: Logs: read: Is a directory
      465     7399   260591 SymPy-tutorial.ipynb
      836     1816   153748 SymPy.ipynb
      ...
      ...
      ...
4423 15252 190180 total
# By defining a macro we can trap any run(cmd) exit error.
julia> macro traprun(c)
  quote
    if typeof($(esc(c))) == Cmd
      try
        run($(esc(c)))
      catch
        end
      end
    end
  end
end
```

The following function filters on a regular expression in order to get rid of the directory warnings from the previous snippet:

```
function filter(pat::Regex, dir=".")
  a = Any[]
  for f in readdir(dir)
    occursin(pat,f) && push!(a, f)
  end
```

```
    return a
end
```

Now, find all the IJulia/Jupyter notebooks in the Chp05 code directory:

```
julia> @traprun `find "." -name \*.ipynb`;
./Code/SymPy.ipynb
./Chp05.ipynb
./Code/SymPy-tutorial.ipynb
```

This concludes our discussion of Julia interoperability. It is an active topic, and at the time of writing, problems in some areas (for example, with Matlab) still exist. So, the reader is encouraged to look at the *JuliaInterop* group (<https://github.com/JuliaInterop>) for news and more information.

## Summary

In this chapter, we looked at three different ways that Julia can interact with other languages.

The first, simplest, and most effective is to refer to routines, residing in shared object libraries, usually created from C or Fortran compilers. We noted that this is also a convenient method to create some code and call it from Julia without any formal API being necessary.

Next, we considered some popular languages such as Python, R, and Java, and described how Julia packages exist to make the process simpler; also, we looked at how wrapper packages have been created to utilize existing ones, with a specific example of the SymPy Python package.

Finally, we discussed Julia's ability to interface with the operating system in conjunction with pipelining and capturing the output and showed how by this mechanism it is possible to utilize other languages in a similar fashion to Unix utilities. The examples focused mainly on Perl but demonstrated briefly how languages such as Perl 6, Ruby, and Python can be treated in a similar fashion.

In the remainder of the book, we will follow with a discussion by looking at in-built Julia methods for handling a variety of datasets, both textual and binary, and introducing some sophisticated structures such as DataFrames and DTables.



# 6

## Working with Data

In many of the examples we have looked at so far, the data we've used was a little artificial, being generated using random numbers. It is now time to consider how Julia uses data held in files. In this chapter, I'm going to concentrate on simple disk-based files and leave discussions of data stored in databases and on the web for later chapters.

We will examine various statistical approaches in Julia, including both the methods integrated into Julia Core and those found in the increasing selection of packages that deal with statistics and related subjects.

Specifically, this chapter will cover the following topics:

- Basic terminal input/output (I/O)
- Handling text and binary files
- Structured datasets
- Time series
- DataFrames
- Simple statistics

We'll begin by discussing basic I/O.

### Basic I/O

Julia views its data in terms of a byte stream. This may be from/to **standard input (stdin)** or **standard output (stdout)**, which may be superseded by a disk file.

If this is coming from a network socket or a pipe, this is essentially asynchronous, but the programmer doesn't need to be aware of this as the stream will block until cleared by an I/O operation. The primitives are the `read()` and `write()` functions. They deal with binary data. With formatted data, such as text, several other functions are layered on top.

It provides very efficient operations both locally and over a distributed system (for example, a network) and is central to Julia's efficient approach to I/O.

## Terminal I/O

Previously, we mentioned that Julia has the concept of stdin and stdout. This is inherited from an idea in **Posix**-compliant operating systems. Its purpose is to stream data through I/O channels.

All streams in Julia have at least a read and a write routine; these take a filename, a Julia channel/stream object, or, for `read()`, a command as their first argument:

```
read(filename::AbstractString, args...)
read(s::IO, nb=typemax(Int))
read(s::IOStream, nb::Integer; all=true)
read(command::Cmd)
read(command::Cmd, String)
write(io::IO,x) write(filename::AbstractString, x)
```

I/O streams need to be opened and closed for files, sockets, and so on. We will deal with how this is done in the relevant sections.

The standard streams, stdin and stdout, plus an additional *output* stream for error messages, stderr, are opened by Julia when it starts up and should not normally be closed.

## Terminal output

Outputting to either of the stdout or stderr streams is easily done by using the `write()` routine:

```
julia> write(stdout, "Help me!!!\n")
Help me!!!
11
julia> write(stderr, "Error messages should go here.\n")
Error messages should go here.
31
```

Writing is literal, so if a newline is required, it must be specified; `\n` works for Linux, Mac OS X, and Windows.

The routine returns the number of characters written; due to the asynchronous nature of the I/O, the REPL outputs the string before the number of characters that were written.

By default, messages sent to stderr go to the same display device as stdout, but it is possible to redirect one or both of these to split the output streams.

In earlier examples, using the `print` function, rather than `write`, absorbs the byte count; also, the `println()` function will append a carriage return.

---

When requiring formatted output, we saw that Julia employs the macro system for generating the boilerplate.

**Note**

The whole system is so complex that it warrants its module in STDLIB: `Printf`. This must (now) be included via a `using Printf` statement.

Also in the module is a `@sprintf` macro, which will print formatted output to a string rather than to `stderr`.

Julia does not have a `@fprintf` macro to write to a file; instead, it uses an extended form of `@printf`, as we will see later.

## Terminal input

Reading from the terminal is more complex.

In *Chapter 1*, we looked at playing a game of “Bulls and Cows” to give you a taste of Julia and remarked that terminal input within a Jupiter notebook is different from that in the REPL; it may be clearer now that this is due to the way that Jupyter handles the input stream.

The following code may help clarify this:

```
# Try the following in the Jupyter and the REPL
# (CR => carriage return, ^D => control-D)
# Type a letter + CR
julia> read(stdin,Char)
A
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
# Type a string + ^D^D (CR is part of the string)
julia> read(stdin,String)
Goodbye cruel world.
"Goodbye cruel world.\n"
# Type a number + CR
julia> read(stdin,Int32)
1234
875770417
# Type same number + CR
julia> read(stdin,UInt32)
1234
0x34333221
```

The behavior when inputting the integer seems bizarre; however, the latter result from the `UInt` gives us a clue.

Note that 0x34333221 is a representation of ASCII code 1234 but reversed.

So, the prior result (875770417) is the ASCII representation of 1234, reversed and converted into an integer!

When reading from `stdin`, which specifies an argument, we get the following:

```
# Input following type to stdin
julia> read(stdin)
a
bc
d
# Output after terminating with ^D
7-element Array{UInt8,1}:
0x
0x0a
0x
0x
0x0a
0x
0x0a
```

The input is terminated with control-D and creates a seven-element byte array, comprising the ASCII codes for the characters and the returns.

It is also possible to define the number of characters required and terminate this with a return:

```
julia> read(stdin,4)
abcd
4-element Array{UInt8,1}:
0x61
0x62
0x63
0x64
```

In the preceding code, additional characters will be interpreted as a variable and give a defined variables error. For example, inputting abcdef results in the following output:

```
julia> ef
ERROR: UndefVarError: ef not defined
```

Julia also has a `readline()` function that, because of what is described in the documentation, as a hack, works in Jupyter as well as the REPL:

```
julia> a = readline(stdin)
abcd
"abcd"
```

Note that `stdin` is the default, so it's **not** usually needed.

This always returns a string and leads to methods for inputting variables such as integers and floats by converting the strings with a bit of care in the conversion.

Let's define a function to get an integer from the command line (`stdin`) and test it:

```
julia> function getInt()
s = chomp(readline())    # Remove trailing LF using chomp()
try
    parse(eltype(1),s)
    catch ex
        println(ex, "\nCan't convert $s to an integer")
    end
end
getInt (generic function with 1 method)

julia> getInt()
1234
1234
```

We'll do the same to get a real (float) value:

```
julia> function getFloat()
s = chomp(readline())
try
    parse(eltype(1.1),s)
    catch ex
        println(ex, "\nCan't convert $s to a float")
    end
end
getFloat (generic function with 1 method)

julia> getFloat()
12.31
12.31
```

Next, let's consider how Julia handles I/O from text and binary files.

## Text files

When dealing with files on disk, they need to be opened. The process establishes a channel to the file that may be for reading, writing, or both.

How this is done depends on the arguments to the `open()` call and there are two main (equivalent) syntaxes:

```
open(filename::AbstractString; keywords...)
open(filename::AbstractString, [mode::AbstractString])
```

The difference is largely historical and inherent in the forms of operating system shell commands.

The keywords consist of a set of five Boolean arguments:

- Read open for read, not write
- Write open for write, truncate, or append
- Create if does not exist; not read and write or truncate or append
- Truncate to zero-size; not read and write
- Append seek to end of file false

Not all of the combinations of the flags are logically consistent, so it is more usual to employ the other form where the mode is passed as an ASCII string:

- `r`: Read
- `w`: Write, create, truncate
- `a`: Write, create, append
- `r+`: Read and write
- `w+`: Read and write, create, truncate
- `a+`: Read and write, create, append

The default when neither of these forms is used is to open the file for reading.

To demonstrate some simple file operations, let's output the first verse of the *Jabberwocky*, the poem by Lewis Carroll:

```
# Pick up the location of the file
# This is mine it may differ for the reader
julia> fname = String(ENV["HOME"], "/MJ2/Alice/jabberwocky.txt");
```

One way to check if this exists is to use `isfile()`.

If it does but it's not a regular file, this would also return `false`.

Note that there is a routine called `isdir()` that checks that `fname` exists and is a directory:

```
julia> isfile(fname)
true
# Open for reading and read the first 4 lines
# Allow for the Julia scoping rules
julia> fip = open(fname)
julia> k = 0;
julia> while !eof(fip)
    ln = readline(fip)
    global k = k + 1
    (k > 4) ? break : println(ln)OK
end
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.
julia> close(fip)      # Remember to close the channel
```

To save the need to monitor the number of lines and break at a certain limit, I have created a `jabber4.txt` file consisting of the first verse of the *Jabberwocky*.

An alternative syntax for the `open()` command is shown here:

```
julia> flp4 = ENV["HOME"] * "/MJ2/Alice/jabber4.txt";
julia> open(flp4) do pn4
    while !eof(pn4)
        println(readline(pn4))
    end
end
julia> eof(pn4)
ERROR: UndefVarError: pn4 not defined
```

Conveniently, we can now read all the files into a variable without explicitly opening it, which means we don't have to close it.

Take note that the data is returned as a byte array (since the file may be `binary_`).

So, for a text file, we need to convert it into `jb4 = String(read(flp4))`.

This needs to be split of '`\n`'s to reproduce the familiar first verse.

Instead of supplying a filename as the first argument in `open()`, we can precede it with a function that takes `IOStream` as its argument:

```
julia> capitalize(f::IOStream) =
           chomp(uppercase(String(read(f))));

julia> split(open(capitalize, f1p4), "\n")
4-element Array{SubString{String},1}:
 "'TWAS BRILLIG, AND THE SLITHY TOVES"
 "DID GYRE AND GIMBLE IN THE WABE:"
 "ALL MIMSY WERE THE BOROGOVES,"
 "AND THE MOME RATHS OUTGRABE."
```

## Text processing

In the previous chapter, we ended by looking at utilizing operating system utilities and scripting languages such as Perl to process (that is, modify) text.

From the preceding `capitalise` example, it is clear that this can be done purely using Julia – in this section, we will delve a little further into the subject.

Previously, we used Perl to reverse the lines of a poem. Here is a native Julia version:

```
julia> open(f1p4) do pn
           while !eof(pn4)
               println(reverse(chomp(readline(pn4))))
           end
       end
sevot yhtils eht dna ,gillirb sawT':ebaw eht ni elbmig dna eryg
diD,sevogorob eht erek ysmim llA.ebargtuo shtar emom eht dnA
```

Again, we need to `chomp` the line, reverse it, and then use `println()`; otherwise, `\n` would come at the front of each line.

Let's continue with a trickier task – that is, emulating the `wc` command. In the following routine, only the words in a file are counted; this is more difficult than counting lines and characters, which I will leave as an exercise for you:

```
# We will need to split on white space and also punctuation marks
julia> const PUNCTS =
          [ ' ', '\n', '\t', '-' , '.', ',', ':', ';', '!', '?', '\'', '\"' ];
```

The following routine works by creating a hash (`Dict`) with the unique words found as keys and the count as the values:

```
julia> function wordcount(text)
           wds = split(lowercase(text), PUNCTS; keepempty = false)
```

```

d = Dict()
for w = wds
    d[w] = get(d,w,0) + 1
end
return d
end

```

The `get (d, w, 0)` call retrieves the value of the key, `[w]`, from the `Dict` hash, `d`; when the key does not exist, an entry is created and the default value of 0 is used. We can read the entire poem into a string and apply this function since we are splitting on `\n` as well as spaces:

```

# Test it on our four line Jabberwocky
julia> wordcount(String(read(flp4)))
Dict{Any,Any} with 18 entries:
"gyre" => 1
"and" => 3
"brillig" => 1
"raths" => 1
"in" => 1
"mome" => 1
"toves" => 1
"mimsy" => 1
"twas" => 1
"did" => 1
"the" => 4
"borogoves" => 1
"were" => 1
"all" => 1
"wabe" => 1
"outgrabe" => 1
"slithy" => 1
"amble" => 1

```

Most words only occur once but `and` and `the` have a higher frequency, and agree with a quick scan of the file.

`wordcount ()` returns a dictionary of the words in a file.

If we collect the values (that is, the counts) and sum them, this gives us a total for the file:

```

#= Setup a constant to point to the Alice folder Notice the convenient
trailing '/' =#
julia> using Printf
julia> const ALICEDIR = ENV["HOME"]*"~/MJ2/Alice/";
#=
```

```
Filter to look just at the '.txt' files in the Alice directory. We can
collect all the values of the Dict in an array and sum it for the
total in the file
=#
julia> for fname in readdir(ALICEDIR)
if match(r"\.txt$", fname) != nothing
    open(ALICEDIR*fname) do f
        n = sum(collect(values(wordcount(String(read(f))))))
        @printf "%s: %d\n" fname n
    end
end
aged-aged-man.txt: 512
father-william.txt: 278
hunting-the-snark.txt:4524
jabber4.txt: 23
jabberwocky.txt: 168
lobster-quadrille.txt: 231
mad-gardeners-song.txt: 348
red-snark.txt: 5153
voice-of-the-lobster.txt:158
walrus-and-carpenter.txt: 623
```

Let's look at some of the characters in the *Hunting of the Snark*.

In true Carrollin fashion, all the occupants on the *hunt* had names beginning with *B*:

```
# Create the Dict for the Snark poem
julia> snarkDict =
    wordcount(String(read(ALICEDIR*"hunting-the-snark.txt")))
julia> wds =
["baker","banker","barrister","beaver","bellman","boots","butcher"];
julia> for w in wds
    @printf "%12s => %4d\n" w snarkDict[w]
end
baker => 10
banker => 7
barrister => 5
beaver => 18
bellman => 30
boots => 3
butcher => 13
```

In fact, because of splitting on quote ( ' ), we will treat s as a word because of entries such as Bellman's.

Our routine should take account of this and similar anomalies if it's counting all the words in the poem rather than just matching selected ones.

## Binary files

Julia can handle binary files as easily as text files using `read()` and `write()`.

Earlier, we created a simple grayscale image for a Julia set. Here, we will read the file and invert the image:

```
julia> cd(ENV["HOME"]*"/MJ2/DataSources/Files");
julia> img = open("juliaset.pgm");
julia> magic = chomp(readline(img))
```

We can open the file in the normal way. The first value is the “magic” number P5 (for a PGM file), terminated by \n, so we can get that with `readline()`.

We will be creating another PGM file so that can write the `magic` number.

The next line comprises three integers – that is, the width and height of the image and the maximum pixel value (usually 255).

These are read and copied without change:

```
julia> if magic == "P5"
    out = open("jsetinvert.pgm", "w");
    println(out, magic);
    params = chomp(readline(img));
# Should be => "800 400 255"
    println(out, params);
# Params splits to strings. we need integers
    (wd,ht,pmax) = parse.(Int64,split(params))
# Create a byte array and read ALL the image data in one call.
    np = wd*htbuf = Array{UInt8,1}(undef,np)
    readbytes!(img, buf, np);
# Invert the gray scales and write it back
    bufX = [UInt8(255 - buf[i]) for i = 1:np]
    write(out,bufX)
    close(out);
else
    error("Not a NetPBM grayscale file")
end
julia> close(img);
```

The following points in the preceding code should be noted:

1. The routine reads all the remaining bytes into a single-byte buffer of size ( $width \times height$ ).
2. For large images, it might be necessary to process the image row by row in a loop; the logic will be virtually the same.
3. `readbytes! (img, buf, n)` reads up to  $n$  bytes in the byte array, `buf`.
4. The array will be extended to size  $n$  if it is too small, so we define a 0-size array to begin with.
5. If there are insufficient bytes remaining in the file, the rest of the buffer is filled with nulls (0x00).

The resulting image is shown in *Figure 6.1*:

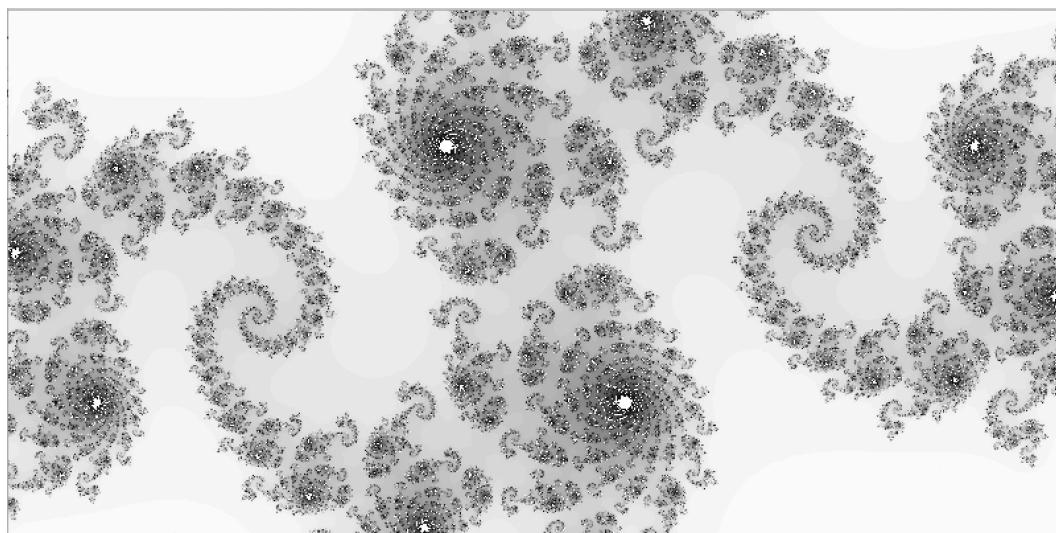


Figure 6.1 – Inverted grayscale image of a Julia set

It is also possible to add some color using the following algorithm for each pixel value.

This is called pseudo-color because there is still a single pixel value in the range [0,255]. The true color would have three values – that is, red, green, and blue:

```
julia> function pseudocolor(pix)
    if pix < 64
        pr = UInt8(0)
        pg = UInt8(0)
        pb = UInt8(4*pix)
    elseif pix < 128
        pr = UInt8(0)
```

```

    pg = UInt8(min(4*(pix - 64),255))
    pb = UInt8(255)
elseif pix < 192
    pr = UInt8(0)
    pg = UInt8(255)
    pb = UInt8(min(4*(192 - pix),255))
else
    pr = UInt8(min(4*(pix - 192),255))
    pg = UInt8(min(4*(256 - pix),255))
    pb = UInt8(0)
end
return (pr, pg, pb)
end

```

Because we are changing a PGM file to a PPM one, the magic number needs to be changed from P5 to P6.

The rest of the code is similar to what it was previously except we're writing the RGB values from the `pseudocolour()` routine in a loop:

```

julia> if magic == "P5"
    out = open("jsetcolor.ppm", "w");
    println(out, "P6");
    params = chomp(readline(img));
# => "800 400 255"
    println(out, params);
    (wd,ht,pmax) = parse.(Int64,split(params))
    np = wd*ht;
    buf = Array{UInt8,1}(undef,np)
    readbytes!(img, buf, np);
    for j = 1:np
        (r,g,b) = pseudocolor(buf[j]);
        write(out,UInt8(r))
        write(out,UInt8(g))
        write(out,UInt8(b))
    end
    close(out);
else
    error("Not a NetPBM grayscale file")
end
julia> close(img)

```

**Note**

The color version that was created in this code can be found in the IJulia notebook, as well as in the ebook accompanying this chapter.

In the next section, we will look at files that contain metadata to indicate the way that data is arranged, as well as the values themselves.

## Structured datasets

Structured datasets include simple delimited files, such as the familiar **comma-separated-values (CSV)**, and files incorporating more descriptive metadata, such as XML and HDF5.

In this section, we will discuss the important topic of Julia's DataFrames. This will be familiar to all R users. They are also implemented in Python via the `pandas` module.

### CSV and other delimited (DLM) files

Data is often presented in table form as a series of rows representing individual records and fields corresponding to a data value for that particular record, rather than the relatively unstructured forms we have seen in the previous files.

Columns in the table are consistent, in the sense that they may all be integers, floats, dates, and so on, and are to be considered as the same "class" of data.

This might be familiar to you as it maps directly to the way data is held in a spreadsheet.

#### ***CSV file format***

One of the oldest forms of representing such data is the CSV file. This is essentially an ASCII file in which fields are separated by commas, and records (rows) by a newline, `\n`.

There is an obvious problem if some of the fields are strings containing commas, so CSV files use quoted text (normally using a double quote, `"`) to overcome this. However, this gives rise to the new question of how to deal with text fields that contain the `"` character.

The CSV file was never defined as a standard and a variety of implementations exist.

However, the principle is clear: we require a method that provides a field separator and a record separator, along with a way to identify any cases where the field and record separators are to be interpreted as regular characters. These types of files are often name-delimited; Julia supports DLM in addition to more specific CSV ones.

We will start by looking at CSV files, supported by the `CSV.jl` package. In the files in the GitHub repository accompanying this book, we have some data on the stock prices of Apple:

```
# Apple stock has the abbreviation AAPL
# We will need the CSV, Statistics and Printf.
julia> using CSV, Statistics, Printf
julia> cd(ENV["HOME"]*"/MJ2/DataSources")
# Check the data file exists
julia> aaplcsv = "CSV/AAPL.csv"; isfile(aaplcsv)
true
```

The `CSV.File(aaplcsv)` routine returns the schema with 13 separate fields, with the first being a Date value and the remainder being real (Float64) values.

Notice the `Missing` and `Floating64` unions as the data may have missing values.

The first five fields, following the date, are the most common and correspond to the open/high/low/close prices and the volume traded on specific dates. The remainder refer to adjusted values, which account for possible splits and dividend payouts.

These are sometimes referred to as OHLCV values.

I have included a file called `AAPL-Short.csv` that contains only the OHLCV values:

```
julia> aapl = CSV.File(aaplcsv)
CSV.File("CSV/AAPL.csv", rows=8336):
julia> aapl[1]
CSV.Row: (
Date = Dates.Date("2013-12-31"),
Open = 554.17,
High = 561.28,
Low = 554.0,
Close = 561.02,
Volume = 7.9673e6,
var"Ex-Dividend" = 0.0,
var"Split Ratio" = 1.0,
var"Adj. Open" = 550.8915872061448,
var"Adj. High" = 557.9595251765071,
var"Adj. Low" = 550.7225929086818,
var"Adj. Close" = 557.7010633097991,
var"Adj. Volume" = 7.9673e6)
#= Print values for first 5 rows
e.g., The change from opening and closing, and the daily spread from
high and low prices. =#
julia> k = 0;
```

```
julia> for r in aapl
    rChange = r.Close - r.Open
    rSpread = r.High - r.Low
    @printf "%10s : %6.2f : %16.2f\n" r.Date rChange rSpread
    global k = k + 1
    if k > 5 break end
end
2013-12-31 : 6.85 : 7.28
2013-12-30 : -2.94 : 7.77
2013-12-27 : -3.73 : 4.91
2013-12-26 : -4.20 : 6.12
2013-12-24 : -2.22 : 5.85
2013-12-23 : 2.09 : 7.96
```

To look at all the values, the most convenient method is to *pipe* the CSV.File structure to a DataFrame. We will discuss these in more detail later in this chapter:

```
# Use the short version of the file containing the OHLCV data
julia> using DataFrames
julia> aaplcsv = "CSV/AAPL-SHORT.csv"
julia> aapls = CSV.File(aaplcsv)
julia> dfs = aapls |> DataFrame
julia> sort!(dfs)
752×6 DataFrame
  Row | Date      Open     High     Low      Close     Volume
      | Date      Float64  Float64  Float64  Float64  Float64
  ____|_____|_____|_____|_____|_____|_____
  1  | 2000-01-03  104.88  112.5   101.69  111.94  4.7839e6
  2  | 2000-01-04  108.25  110.62  101.19  102.5   4.5748e6
  3  | 2000-01-05  103.75  110.56  103.0   104.0   6.9493e6
  4  | 2000-01-06  106.12  107.0   95.0    95.0   6.8569e6
  5  | 2000-01-07  96.5   101.0   95.5   99.5   4.1137e6
```

DataFrames can be queried with the Queryverse (more on this later in *Chapter 9* when we return to the subject of data sources and databases).

The following code returns the OHLCV values versus Date from the previous DataFrame, df, for the complete dataset, starting with values beginning on 20-12-2013.

The query returns the final seven values in the dataset:

```
julia> using Query, Dates
julia> x = @from i in dfs begin
    @where i.Date >= Date(2013,12,20)
    @select {i.Date, i.Open, i.High, i.Low, i.Close}
```

```

@collect DataFrame
end
7×5 DataFrame
Row | Date      Open   High    Low     Close
    | Date      Float64 Float64  Float64  Float64
 1 | 2013-12-31 554.17 561.28 554.0 561.02
 2 | 2013-12-30 557.46 560.09 552.32 554.52
 3 | 2013-12-27 563.82 564.41 559.5 560.09
 4 | 2013-12-26 568.1 569.5 563.38 563.9
 5 | 2013-12-24 569.89 571.88 566.03 567.67
 6 | 2013-12-23 568.0 570.72 562.76 570.09
 7 | 2013-12-20 545.43 551.61 544.82 549.02

```

### **Other DLM file formats**

As was remarked earlier, CSV files are a particular instance of a **DLM**. The problem of using commas as field separators is exacerbated by enclosing strings in quotes, but this then has the effect of making " a specific markup character.

A second common type of file is one that uses **TAB** as a field separator, largely making the need for " redundant.

These are termed **tab-separated value (TSV)** files, and in this case, the DLM module is useful.

In the `Files` folder is a file called `UKH-Prcs.tsv`. This is in TSV format and provides the house prices in various UK regions monthly for 20 years from 1996 to 2017:

```

julia> using DelimitedFiles
julia> cd(ENV["HOME"]*"/MJ2/DataSources")
julia> ukhptsv = "CSV/UKH-Prcs.tsv"; isfile(ukhptsv)
true

```

To input the data, we can use the following function:

```
readdlm(source, delim::AbstractChar, eol::AbstractChar; options ...)
```

If all data is numeric, the result will be a numeric array; if some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

One of the options is `header`, which is a Boolean – if set to true, the routine returns the data and header in separate arrays, with the header being read as the first line in the dataset:

```

# The file has a header line, signal that
julia> (ukhpData,ukhpHead) = readdlm(ukhptsv, '\t'; header=true);
# The header is a 1x10 array.

```

```
julia> ukhpHead
1x10 Array{AbstractString,2}:
"Inner London" "Outer London" "North East" ... "South West"
# and the data portion is a 240x10 matrix ...
julia> (ukd1, ukd2) = size(ukhpData)
(240, 10)
# ... all of which are Floats
julia> typeof(ukhpData)
Matrix{Float64} (alias for Array{Float64, 2})
```

We can visualize this data using PyPlot:

```
julia> using PyPlot
julia> t = collect(1:ukd1);
julia> for i in 1:10
    plot(t,ukhpData[:,i])
end
```

Here's the output:

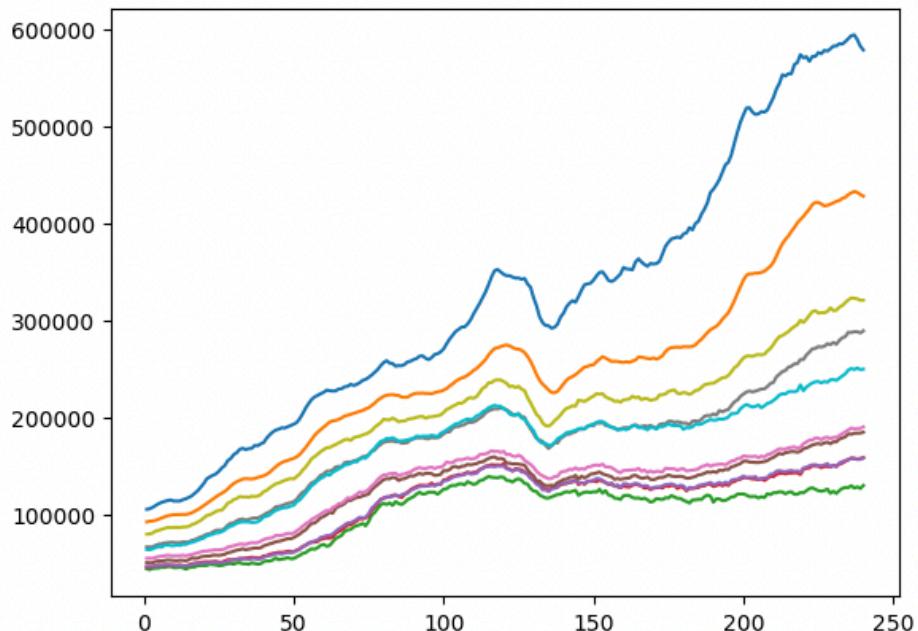


Figure 6.2 – House prices in various UK regions

As a further example, let's use the DLM routine to read the CSV file of Apple (AAPL) stock prices:

```
julia> aaplcsv = "CSV/AAPL-Full.csv"; isfile(aaplcsv)
true
julia> (aaplData,aaplHead) = readdlm(aaplcsv, ',', header=true);
julia> aaplHead
1x13 Array{AbstractString,2}:
"Date" "Open" "High" "Low" "Close" ... "Adj. Close" "Adj. Volume"
```

Now, we can use a slice to display the first 10 lines and 6 columns. We have not sorted the array, so the data is in descending date order.

To plot this, we can take each column and apply `reverse()` to each:

```
julia> aaplData[1:10, 1:6]
10 ×6 Array{Any,2}
julia> using Dates, PyPlot
julia> d0 = Date("2000-01-01");
julia> aaplDate = reverse(Date.(aaplData[:,1]));
julia> aaplOpen = reverse(Float64.(aaplData[:,2]));
julia> aaplClose = reverse(Float64.(aaplData[:,5]));
julia> const NAAAPL = length(aaplDate);
julia> aapl_days = zeros(Int64, NAAAPL);
julia> d0 = aaplDate[1];
julia> dt = [(aaplDate[i]-d0).value for i = 1:NAAAPL];
julia> aaplDifs = [(aaplClose[i] - aaplOpen[i]) for i = 1:NAAAPL];

# Plot the daily difference between Open and Close
julia> plot(dt,aaplDifs)
julia> title("Apple Stock - Daily Change")
```

Here's the output:

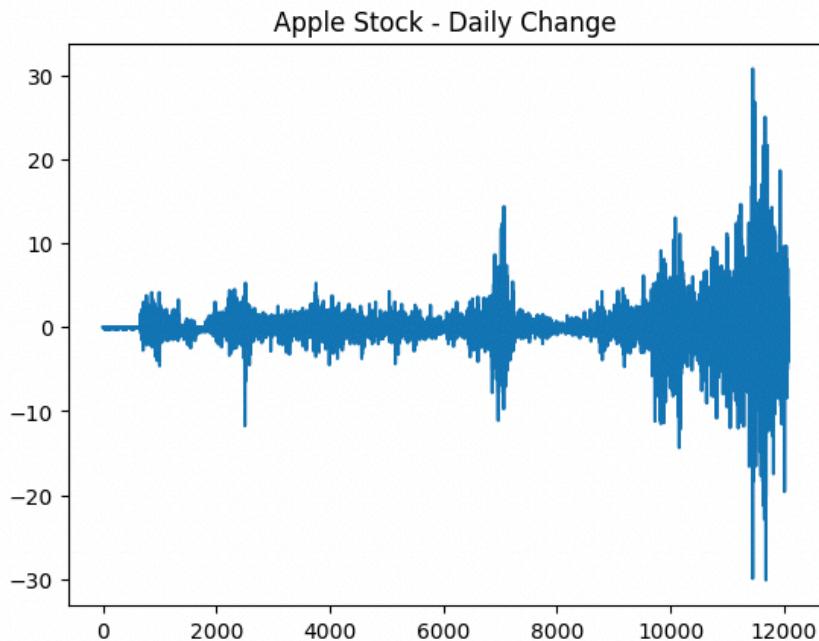


Figure 6.3 – Daily changes in Apple stock prices since 01-01-2001

## HDF5 and JLD files

**Hierarchical Data Format v5 (HDF5)** was originally developed by the NCSA in the USA and is now supported by the HDF Group. It was developed to store large amounts of scientific data that's exposed to the user as groups and datasets; these are akin to directories and files in a conventional file system. v5 was developed to overcome some of the limitations of the previous version (v4) and Julia. Similar to languages such as Python, R, and Matlab/Octave, it has extensions to be able to access files in HDF5 format. HDF5 also uses “attributes” to associate metadata with a particular group or dataset and ASCII names for these different objects. Objects can be accessed by UNIX-like pathnames, such as `/projects/juno/tests/first.h5`, where `projects` is a top-level group, `juno` and `tests` are subgroups, and `first.h5` is a dataset.

You also need to have the HDF5 library installed on your system (version 1.8 or higher is required).

Language wrappers for HDF5 may be viewed as either “low level” or “high level.” The Julia package, `hdf5.jl`, contains both levels. At the low level, it directly wraps HDF5’s functions, copying their API and making them available from within Julia. At the high level, it provides a set of functions that are built on the low-level wrapper to simplify the usage of the library. For simple types (scalars,

---

strings, and arrays), HDF5 provides sufficient metadata to know how each item is to be interpreted while representing the data in a way that is agnostic of computing architectures.

Plain HDF5 files can be created and/or opened in Julia with the `h5open` command:

```
fid = h5open(filename, mode), where mode can be any one of the
following:
"r" : read-only
"r+" : read-write, preserving any existing contents
"w" : read-write, destroying any existing contents
```

This returns an object of the `PlainHDF5File` type, a subtype of the abstract `HDF5File` type.

“Plain” files have no elements (groups, datasets, or attributes) that are not explicitly created by the user:

```
# A quick snippet creating an HDF5 file.
julia> using HDF5
julia> h5file = "Files/mydata.h5"
julia> aa = [u + v*rand() for u = 0.5:0.5:10.0, v = 0.5:0.5:6.0]
julia> h5open(h5file, "w") do f
    write(f, "aa", aa)
end
```

Alternatively, we can say `h5write(h5file, "aa", aa)` or else `@write h5file aa` using a routine call or a macro.

This can be read back without the `h5open()` statement, similar to the `h5write()` syntax we looked at previously.

We can also create a slice of the file on the fly, without reading all the dataset into memory:

```
julia> bb = h5read(h5file, "aa", (2:3:14, 4:3:10))
5x3 Matrix{Float64}:
 1.98913  1.35794  1.32819
 2.63025  3.85922  5.63247
 4.40735  4.5089   7.74393
 6.68994  7.38884  10.0386
 7.54889  7.32411  10.6304
```

## Julia data format (JLD)

The `HDF5.jl` package also provides the basis for a specific JLD to accurately store and retrieve Julia variables. While it is possible to use “plain” HDF5 for this purpose, the advantage of the JLD module is that it preserves meta-information such as the exact type of each variable.

At the end of the previous section, we read the Apple stock dataset and computed the differences between opening and closing stock.

The following snippet will save this to a JLD file:

```
julia> using JLD
julia> DS = ENV["HOME"] * "/MJ2/DataSources";
julia> aapljld = "$DS/Files/aapldifs.jld"
julia> rm(aapljld, force = true)
```

Removing the existing file is not strictly necessary since the open for "w" will remove the file. Now, rewrite the file:

```
julia> jldopen(aapljld, "w") do fid
    write(fid, "aaplDate", aaplDate)
    write(fid, "aaplClose", aaplClose)
    write(fid, "aaplDifs", aaplDifs)
end
julia> isfile(aapljld)
true
julia> run(`ls -l $DS/Files/aapldifs.jld`)
-rwxrwxrwx 1 malcolm staff 208552 14 Aug 16:20 Files/aapldifs.jld
```

We can open the JLD file and read back *some* of the data:

```
julia > fid = jldopen(aapljld, "r")
Julia data file version 0.1.3: Files/aapldifs.jld
julia> aaDateX = read(fid, "aaplDate")
julia> aaDifsX = read(fid, "aaplDifs")
julia> close(fid)
```

We can check that the two arrays are the same by using == at the array level – that is,

`aaDifsX == aaDifs`. #=> should return `true`.

## XML files

Alternative data representations are provided via XML and JSON.

We will consider the latter (JSON) in *Chapter 11* when we discuss networking, using the web and REST services.

In this section, we will look at XML file handling and the functionality available in the LightXML package. To assist in this, we will use the `books.xml` file, which contains a list of 10 books.

The first portion of the file is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<catalog>
  <book id="bk101">
    <author>Gambardella, Matthew</author>
    <title genre='Computing'>XML Developer's Guide</title>
    <price currency='GBP'>44.95</price>
    <publish_date>2000-10-01</publish_date>
    <description>An in-depth look at creating applications with XML.</description>
  </book>
  .....
  .....
</catalog>
```

LightXML is a wrapper of libxml2, which provides a reasonably comprehensive high-level interface that covers most functionalities:

- Parse an XML file or string into a tree
- Access the XML tree structure
- Create an XML tree
- Export an XML tree to a string or an XML file

I will cover parsing an XML file here as it is probably the most common procedure.

Both HDF5 and LightXML use routines called `root` and `name` to avoid name clashes. The following calls are fully qualified:

```
julia> using LightXML
julia> xdoc = parse_file("Files/books.xml");
julia> xtop = LightXML.root(xdoc);
julia> println(LightXML.name(xtop));
catalog
```

The `xdoc` variable contains the entire XML dataset, while `xtop` is the primary tag encompassing all of the remaining data:

```
julia> xdoc
<?xml version="1.0" encoding="utf-8"?>
<catalog>
  <book id="bk101">
```

```

<author>Gambardella, Matthew</author>
<title genre="Computing">XML Developer's Guide</title>
<price currency="GBP">44.95</price>
<publish_date>2000-10-01</publish_date>
<description>An in-depth look at creating applications with XML.</
description>
</book>
<book id="bk102">
<author>Ralls, Kim</author>
<title genre="Fantasy">Midnight Rain</title>
<price currency="GBP">5.95</price>
<publish_date>2000-12-16</publish_date>
<description>A former architect battles corporate zombies, an
evil sorceress, and her own childhood to become queen of the world.</
description>
</book>
.....
.....
</catalog>
```

The following code navigates all the child nodes of `xtop`, outputting the titles and the genre into which they have been classified:

```

julia> using Printf
julia> for c in child_nodes(xtop)
    if is_elementnode(c)
        e = XMLElement(c)
        t = find_element(e, "title")
        title = content(t)
        genre = attribute(t, "genre")
        @printf "\n%28s --- %s" title genre
    end
end
XML Developer's Guide --- Computing
                    Midnight Rain --- Fantasy
                    Maeve Ascendant --- Fantasy
                    Oberon's Legacy --- Fantasy
                    The Sundered Grail --- Fantasy
                    Lover Birds --- Romance
                    Splish Splash --- Romance
                    Creepy Crawlies --- Horror
                    Paradox Lost --- SciFi
.NET: The Programming Bible --- Computing
```

Next, we'll look for all the computing books and print out the full details.

The publication date is in YYYY-MM-DD format, so we'll use the `Dates` module to create a more readable string:

```
julia> using Dates
julia> for c in child_nodes(xtop)
    if is_elementnode(c)
        e = XMLElement(c)
        t = find_element(e, "title")
        genre = attribute(t, "genre")
        if genre == "Computing"
            a = find_element(e, "author")
            p = find_element(e, "price")
            curr = attribute(p, "currency")
            d = find_element(e, "publish_date")
            dc = DateTime(content(d))
            ds = string(day(dc), " ", monthname(dc), " ", year(dc))
            desc = find_element(e, "description")
            println("Title: ", content(t))
            println("Author: " , content(a))
            println("Date: " , ds)
            println("Price: " , p , " (" , curr, " )")
            println(content(desc) , "\n");
        end
    end
end
Title: XML Developer's Guide
Author: Gambardella, Matthew
Date: 1 October 2000
Price: <price currency="GBP">44.95</price> (GBP)
An in-depth look at creating applications with XML.
Title: .NET: The Programming Bible
Author: O'Brien, Tim
Date: 9 December 2000
Price: <price currency="GBP">36.95</price> (GBP)
Microsoft's .NET initiative is explored in detail in this deep
programmer's reference.
```

Finally, let's use the genre to select all the books that have been categorized as `SciFi`:

```
julia> using Dates
julia> for c in child_nodes(xtop)
    if is_elementnode(c)
        e = XMLElement(c)
```

```

t = find_element(e, "title")
genre = attribute(t, "genre")
if genre == "SciFi"
    a = find_element(e,"author")
    p = find_element(e,"price")
    curr = attribute(p, "currency")
    d = find_element(e,"publish_date")
    dc = DateTime(content(d))
    ds = string(day(dc)," ",monthname(dc)," ",year(dc))
    desc = find_element(e,"description")
    println("\nTitle: ", content(t))
    println("Author: " ,content(a))
    println("Date: " ,ds)
    println("Price:",p," (", curr, ")")
    println(content(desc),"\n");
end
end
end
Title: Paradox Lost
Author: Kress, Peter
Date: 2 November 2000
Price:<price currency="GBP">6.95</price> (GBP)
After an inadvertent trip through a Heisenberg Uncertainty Device,
James Salway discovers the problems of being quantum.

```

Next, we'll discuss how to handle specialist types of text files containing data relating to time series.

## Time series

In the previous section, we looked at data regarding Apple (AAPL) stock prices. This has a particular format with a date (or timestamp) as the first value in the row, followed by a series of related usually numeric values, and is termed a time series.

Time series are common when analyzing financial data and, in particular, the special discipline of econometrics.

Julia has a special type for time series and a package maintained by the Julia Stats group (<https://juliastats.org/TimeSeries.jl/dev/timearray/>) called `TimeSeries` that defines the type for time array and provides several routines to manipulate the data in it.

### Note

Be careful *not* to use the older package from the Julia Quant group (<https://github.com/JuliaQuant/Timestamps.jl>) as this has fallen somewhat into neglect recently.

We need to install the `TimeSeries` package in the usual way (that is, with the Pkg manager) and create a time array directly from a CSV file.

This `stocks4.csv` file contains closing stock prices weekly for four major US companies: Google (GOOG), Apple (AAPL), Amazon (AMZN), and Microsoft (MSFT) for 2018 and 2019.

We need to create an array containing the time series data and note that it is necessary to specify which field holds what (that is, `column` holds the timestamp values):

```
julia> using TimeSeries
julia> DS = ENV["HOME"]*"/DataSources";
julia> ts4 = readtimearray("$DS/CSV/stocks4.csv";
                           format="dd/mm/yyyy", delim=',')
105x4 TimeArray{Float64,2,Date,Matrix{Float64}}
2018-01-01 to 2019-12-30
|           | GOOG | AAPL | AMZN | MSFT |
|-----|-----|-----|-----|-----|
| 2018-01-01 | 1.0  | 1.0  | 1.0  | 1.0  |
| 2018-01-08 | 1.0182 | 1.0119 | 1.0619 | 1.016 |
| 2018-01-15 | 1.032  | 1.0198 | 1.0532 | 1.0205 |
| 2018-01-22 | 1.0668 | 0.9801 | 1.1407 | 1.0666 |
| ...       | ...   | ...   | ...   | ...   |
| ...       | ...   | ...   | ...   | ...   |
| 2019-12-23 | 1.2265 | 1.656  | 1.5212 | 1.8025 |
| 2019-12-30 | 1.213  | 1.678  | 1.5034 | 1.7882 |
```

This data can also be listed via the `head(ts4, 4)` and `tail(ts4, 2)` function calls. The default for both routines is six rows.

As an iterable dataset, the `first()` and `last()` functions are also available. Unsurprisingly, they do what they say on the tin – that is, they list the first and last rows in the dataset corresponding to the `head(ts4, 1)` and `tail(ts4, 1)` calls, respectively.

The data in a time array can be indexed straightforwardly:

```
julia> ts4[1:3, [:AAPL, :MSFT]]
3x2 TimeArray{Float64, 2, Date, Matrix{Float64}}
2018-01-01 to 2018-01-15
|           | AAPL | MSFT |
|-----|-----|-----|
| 2018-01-08 | 1.0119 | 1.016 |
| 2018-01-15 | 1.0198 | 1.0205 |
| 2018-01-22 | 0.9801 | 1.0666 |
```

It is always useful to visualize the dataset. We will do this using the Plots package and the GR backend:

```
julia> using Plots
julia> gr()
Plots.GRBackend()
julia> plot(ts4)
```

The resulting graph can be seen in *Figure 6.4*:

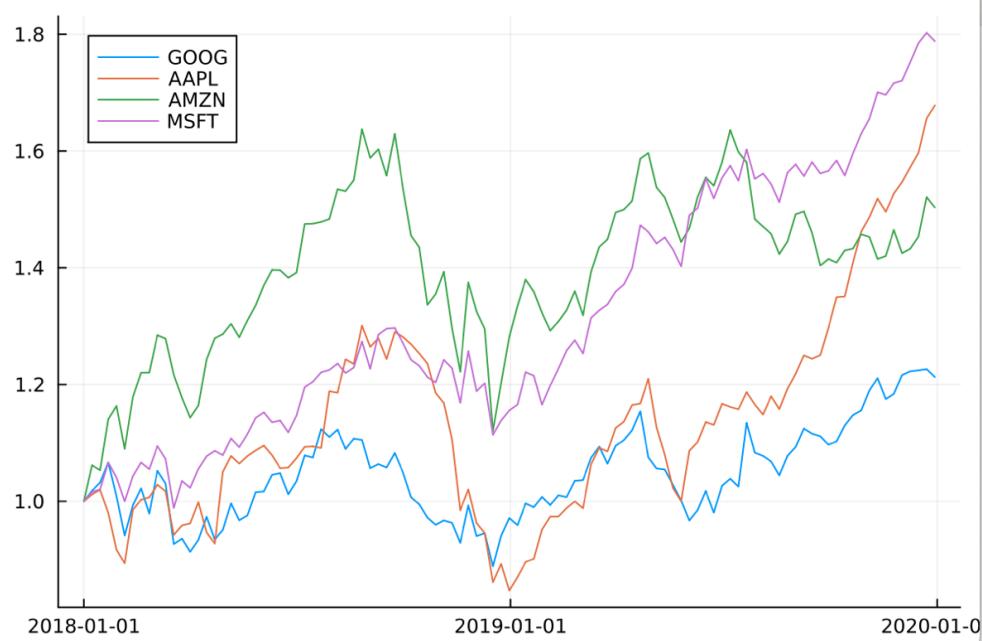


Figure 6.4 – A time series plot of four stocks from the stocks.csv file

The `TimeSeries` package contains routines for computing `lead()` and `lag()` for a set of values, as well as differences between the corresponding neighbors and the previous ones. Here, notice how the table now has 104 rows starting at the date 2018-01-08:

```
julia> head(diff(ts4))
104x4 TimeArray{Float64,2,Date,Matrix{Float64}}
2018-01-08 to 2019-12-30
|       | GOOG    | AAPL    | AMZN    | MSFT    |
| 2018-01-08 | 0.0182 | 0.0119 | 0.0619 | 0.016   |
| 2018-01-15 | 0.0138 | 0.0078 | -0.0086 | 0.0045  |
| 2018-01-22 | 0.0348 | -0.0397 | 0.0874 | 0.046   |
| 2018-01-29 | -0.058  | -0.0629 | 0.0227 | -0.0259 |
```

2018-02-05   -0.0672   -0.0234   -0.0735   -0.0408
2018-02-12   0.0517   0.0915   0.0888   0.0433

It is also possible to compute moving averages.

The following code applies the `mean` function from the `Statistics` package over four values. Again, the number of rows changes (to 102), and the starting date is now 2018-01-22:

julia> using Statistics julia> head(moving(mean,ts4,4)) 102×4 TimeArray{Float64,2,Date,Matrix{Float64}} 2018-01-22 to 2019-12-30				
	GOOG	AAPL	AMZN	MSFT
2018-01-22   1.0292   1.0029   1.0639   1.0258				
2018-01-29   1.0314   0.9822   1.1048   1.0359				
2018-02-05   1.0123   0.9527   1.1118   1.0319				
2018-02-12   1.0026   0.9441   1.1431   1.0376				
2018-02-19   0.9915   0.9498   1.1631   1.0376				
2018-02-26   0.984   0.9722   1.1774   1.0412				

Finally, let's display the cumulative sum for the weekly differences stocks to see how each performed over the 2 years:

julia> head(upto(sum,diff(ts4)),4) 4×4 TimeArray{Float64, 2, Date, Matrix{Float64}} 2018-01-08 to 2018-01-29				
	GOOG	AAPL	AMZN	MSFT
2018-01-08   0.0182   0.0119   0.0619   0.016				
2018-01-15   0.032   0.0198   0.0532   0.0205				
2018-01-22   0.0668   -0.0199   0.1407   0.0666				
2018-01-29   0.0088   -0.0829   0.1634   0.0407				

julia> tail(upto(sum,diff(ts4)),2) 2×4 TimeArray{Float64, 2, Date, Matrix{Float64}} 2019-12-23 to 2019-12-30				
	GOOG	AAPL	AMZN	MSFT
2019-12-23   0.2265   0.656   0.5212   0.8025				
2019-12-30   0.213   0.678   0.5034   0.7882				

Apple and Microsoft both performed better than Amazon, which was a surprise but we are looking at data for 2018 and 2019.

As an exercise, try plotting the graphs for `diff(ts4)`, `moving(mean, ts4, 4)`, and `upto(sum, diff(ts4))` and see how all six stocks performed.

These are provided in the Jupyter workbook accompanying this chapter.

Note that the `TimeArray` package complies with the Julia Tables interface, so one consequence is that it is quite easy to create a DataFrame from a time series:

```
julia> using DataFrames
julia> DataFrame(ts4)
105x5 DataFrame
Row | timestamp      GOOG      AAPL      AMZN      MSFT
    | Date          Float64   Float64   Float64   Float64
1   | 2018-01-01  1.0       1.0       1.0       1.0
2   | 2018-01-08  1.01817   1.01194   1.06188   1.01599
3   | 2018-01-15  1.03201   1.01977   1.05324   1.02052
4   | 2018-01-22  1.06678   0.980057  1.14068   1.06656
. . .
. . .
```

Thus, next, we will provide a more detailed discussion of DataFrames and *do* some statistics on them.

## DataFrames and statistics

We were introduced to Julia's implementation of DataFrames in the previous section and used the availability of a series of datasets, first made available by the **Comprehensive R Archive Network (CRAN)**, hence the epithet R-Datasets.

A full listing can be obtained from the R-Datasets page and also from the package maintainer's, Vincent Arel-Bundock, GitHub page.

The equivalent package in Python is `pandas`, of which there is also a Julia package (`Pandas.jl`), which is a wrapper around the Python one, available via the JuliaPy GitHub page.

When dealing with tabulated datasets, there are occasions when some of the values are missing. It is one of the features of statistical languages is that they can handle such situations.

Support for this has been changed in version 1.0 due to the introduction of the `Missing.jl` package (via the JuliaData group).

## DataFrames

The DataFrame is one of the cornerstones of Julia. Implementations go back to the very early days of Julia but the current version is a rewrite, totally in native code and many packages handle data in a DataFrame as easily as if it's in a plain array.

In essence, a DataFrame is a matrix where the columns are all the same type but may be different from each other, and may be referenced by name. An analogy would be a sheet in an Excel (or similar) workbook.

In this chapter, we will look at datasets that return a DataFrame and packages that can be used to process them. In *Chapter 9*, I will discuss how to get data sources from the web and transform these into a DataFrame.

### *Excel spreadsheets*

First, it might be worth briefly looking at the XLSX package for handling Excel spreadsheets. We will focus on how to transform the data into a data array.

As an example, we will use the `iris.xlsx` file, an Excel spreadsheet that can be found in the `Files` subdirectory.

The Iris dataset (the first sheet of the spreadsheet was first published in 1936 by the famous statistician Ronald Fisher) contains 50 samples from three iris species: `setosa`, `virginica`, and `versicolor`, which means it contains 150 entries. Various features are measured – that is, sepal length and width and petal length and width. All values are given in centimeters.

We will cover this dataset again in *Chapter 8* as it is a seminal example when looking at the Gadfly visualization package, so I'll leave a more extensive discussion until then.

We can open the file by using the `readxlsx()` routine, which displays information on the number (and names) of sheets, their size, and the worksheet range.

The XLSX module does not expose the names of the routines, so they may be imported, and the function references need to be qualified:

```
julia> import XLSX
julia> xf = XLSX.readxlsx("Files/iris.xlsx")
XLSXFile("iris.xlsx") containing 1 Worksheet
      sheetname    size        range
-----
Sheet 1  152x6          A1:F152
```

In this case, there is only a single sheet consisting of 152 rows.

The first row comprises a single row with a multi-column containing the dataset's name; the second row contains the names of the data columns and rows 3 through 152 are the actual data values.

Notice how the module handles this by returning missing data for columns 2 to 6:

```
julia> xf["Sheet 1!A1:F1"]
1×6 Matrix{Any}:
 "Fisher's IRIS dataset" missing missing missing missing missing
```

Reading the header (and transposing it to make it more readable) produces the following output:

```
julia> header = xf["Sheet 1!A2:F2"];
julia> header[:, :]
6-element Vector{Any}
"sepal_length"
"sepal_width"
"petal_length"
"petal_width"
"species"
"species_id"
```

Notice the *Excel* style referencing here – that is, "Sheet 1!A2:F2".

We can retrieve the data with the same type reference, "Sheet 1!A3:F152":

```
julia> xf["Sheet 1!A3:F8"]
6×6 Matrix{Any}:
 5.1  3.5  1.4  0.2  "setosa"  1
 4.9   3    1.4  0.2  "setosa"  1
 4.7  3.2  1.3  0.2  "setosa"  1
 4.6  3.1  1.5  0.2  "setosa"  1
 5    3.6  1.4  0.2  "setosa"  1
 5.4  3.9  1.7  0.4  "setosa"  1
```

This creates an array of the Any element type since the column values may be numeric or strings.

Similar to `TimeArrays`, the `XLSX` package implements the `DataTable` interface. So, if we wish to create a `DataFrame` from an Excel spreadsheet, we need to read it with `XLSX.readtable()`. This can then be converted into a `DataFrame` via the usual method:

```
julia> df = DataFrame(XLSX.readtable("Files/iris.xlsx",
                                         "Sheet 1", first_row=2, header=true))
150×6 DataFrame
Row | sepal_len  sepal_wid  petal_len  petal_wid  species  sp_id
 1 | 5.1        3.5        1.4        0.2        setosa    1
 2 | 4.9        3          1.4        0.2        setosa    1
```

---

3	4.7	3.2	1.3	0.2	setosa	1
4	4.6	3.1	1.5	0.2	setosa	1
. . . . . . . . . . . .						
. . . . . . . . . . . .						
148	6.5	3	5.2	2	virginica	3
149	6.2	3.4	5.4	2.3	virginica	3
150	5.9	3	5.1	1.8	virginica	3

The `sheet` parameter is required as the routine only reads a single sheet into a data table. There are also several default parameters available. Notably, we have used `first_row=2` to skip the multicolumn row and `header=true` to indicate that the next row contains the column titles.

The complement to `xls.readtable` is `XLSX.writetable`, which can be used to create an Excel file from a DataFrame.

Recall the short form of the Apple stocks data in the `Files` folder. This can be read into a DataFrame with a single call:

```
julia> using CSV, DataFrames
julia> DS = ENV["HOME"]*"~/MJ2/DataSources"
julia> dfs = CSV.File("$DS/CSV/AAPL-Short.xlsx")
```

It can be written to disk like so:

```
julia> XLSX.writetable("Files/AAPL-Short.xlsx", dfs,
                      overwrite = true,
                      sheetname = "Apple Stocks data")
```

Again, there are several default parameters. The import one here is `sheetname`, which, if not specified, defaults to `Sheet1`.

## R-Datasets

`R-Datasets` is maintained by Vincent Arel-Bundock (<https://arelbundock.com>) and `RDatasets.jl` is a package that provides (most) of these.

It can be found at <https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/00Index.html>.

Most of the datasets are grouped into 33 packages; a list can be created by using the `packages()` routine in the `RDatasets` package:

```
julia> using RDatasets
julia> RDatasets.packages()
34x2 DataFrame
Row | Package      Title
    | String15    String
```

1	COUNT	Functions, data and code for cont data
2	Ecdat	Data sets for econometrics
3	HSAUR	A Handbook of Statistical Analysis using R
4	HistData	Datasets from the history of statistics ...
5	ISLR	Data for An Introduction to Statistical ...
6	KMsurv	Data sets from Klein and Moeschberger ...
7	MASS	Support Functions and Datasets for Vena ...
...	...	...
...	...	...

Passing a package name to the `datasets` function as `RDatasets.datasets(MASS)` will output a list of the individual datasets in the package, including a description (title) and the number of rows and columns.

Note that the routine is not exported by `RDatasets`, so the call must be fully qualified (see the notebook for an example).

We will start by picking up some data on the frequency and severity of earthquakes around Fiji from the `RDatasets` package:

```
julia> using DataFrames, RDatasets
julia> quakes = dataset("datasets", "quakes");
```

The following snippet displays the first five rows:

```
julia> quakes[1:5,:]
```

5×5 DataFrame					
Row	Lat	Long	Depth	Mag	Stations
	Float64	Float64	Int64	Float64	Int64
1	-20.42	181.62	562	4.8	41
2	-20.62	181.03	650	4.2	15
3	-26.0	184.1	42	5.4	43
4	-17.97	181.66	626	4.1	19
5	-20.42	181.96	649	4.0	1

To perform some meaningful statistics on this data, we will need to access some additional routines. Strangely, some of the basic ones are in the `Statistics` package, while others (although relatively common) are in `StatsBase`, the latter of which will need to be added via the package manager; both are “used.”

---

At the same time, you may also wish to add a series of other statistical packages that we will need later in this chapter:

- `Distributions`
- `KernelDensity`
- `HypothesisTests`
- `GLM` (that is, General Linear Models)

First, let's look at a dataset on earthquakes available in `RData` to see if there is any correlation between the magnitude and depth of quakes:

```
julia> using Statistics, StatsBase

# Pickup the magnitude and depth of the quakes
julia> mags = Float64.(quakes[!, :Mag]);
julia> depth = Float64.(quakes[!, :Depth]);

# See is there is any correlation between them.
julia> cor(mags, depth)
-0.2306376976876574
```

This is a weak negative one, which is a little counterintuitive; I would have expected deeper quakes to be more powerful, but the power might have been absorbed by the surrounding rock.

Visualizing the data may be useful:

```
# Use the Plots API and the GR backend
julia> using Plots
julia> gr()
Plots.GRBackend()
julia> scatter(depth, mags)
```

The resultant plot is shown in *Figure 6.5*.

The depth (*abscissa*) is given in meters and the magnitude (*ordinate*) is on the Richter scale:

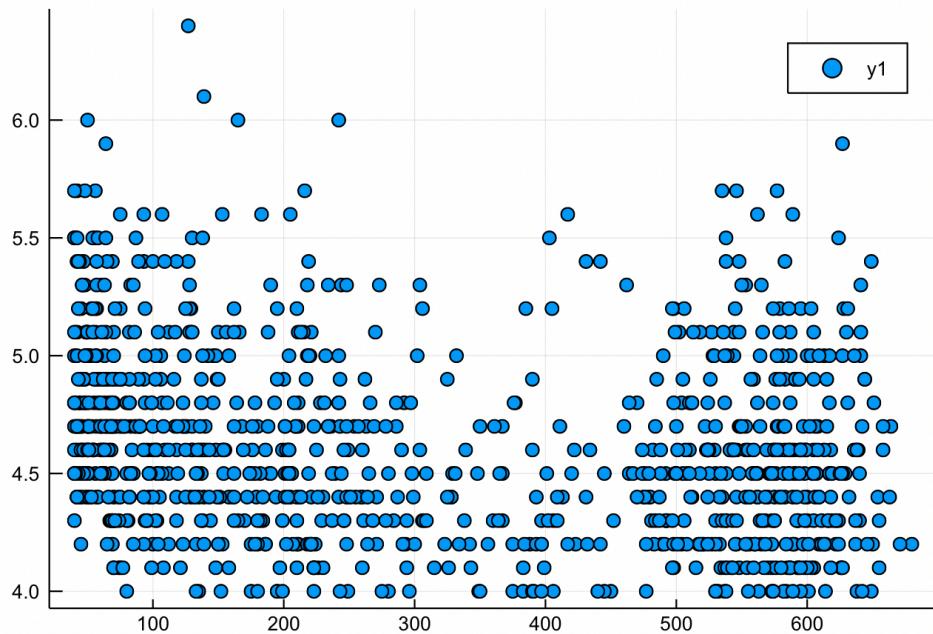


Figure 6.5 – Scatter diagram of depth versus magnitude of earthquakes

As we can see, there are no obvious trends. However, there seem to be two clusters – a deep water one and a shallow water one; the largest quakes occurred for the latter.

We can use the `describe()` routine to look at some summary statistics on quake magnitudes (the granularity of the data is only to one decimal place):

```
julia> describe(mags)
Summary Stats:
Length:           1000
Missing Count:   0
Mean:            4.620400
Minimum:         4.000000
1st Quartile:   4.300000
Median:          4.600000
3rd Quartile:   4.900000
Maximum:         6.400000
Type:            Float64
```

The summary statistics includes the mean, but not the variance, skew, and others.

For some of these, we require the `StatsBase` package:

```
julia> using StatsBase, Printf
julia> mags = Float64.(quakes[!, :Mag]);
julia> (m1, m2, m3, m4) = map(x -> round(x,digits=4),
    [mean(mags), std(mags), skewness(mags), kurtosis(mags)]);
julia> @printf "\nMean: %.4f\nStdV: %.4f\nSkew: %.4f\nKurt: %.4f\n"
m1 m2 m3 m4
Mean: 4.6204
StdV: 0.4028
Skew: 0.5103
Kurt: 0.5103
```

We can easily fit a histogram of the quake magnitudes and display it using the `Plots` package:

```
julia> histmag = fit(Histogram, mags, 4.0:0.1:6.6, closed=:left)
Histogram{Int64,1,Tuple{StepRangeLen{Float64},Base.TwicePrecision{Float64},Base.TwicePrecision{Float64}}}
edges: 4.0:0.1:6.6
weights: [46, 55, 90, 85, 101, 107, 101, 98, 65, 54 ... 9, 8, 0, 2, 3,
1, 0, 0, 1, 0]
closed:left
isdensity: false
julia> plot(histmag)
```

Here's the output:

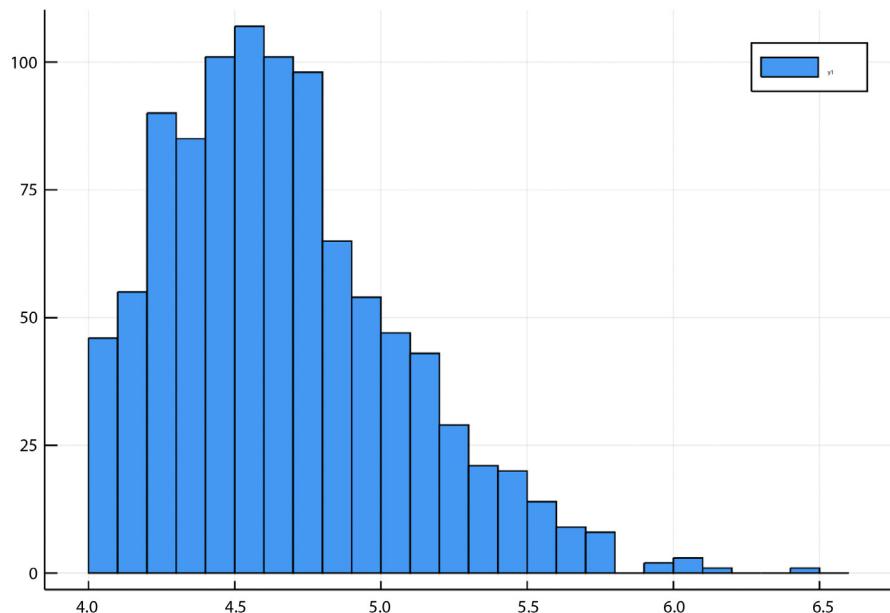


Figure 6.6 – Histogram of the magnitudes of earthquakes

The preceding histogram shows the magnitude between 4.0 to 6.6 in steps of 0.1.

## Some simple statistics

DataFrames are especially useful in the new compendium discipline commonly termed data science. Both Python and R are frequently seen as its cornerstones but with the new application of Julia's `DataFrames` modules, extensive plotting options (see *Chapter 8*), and the addition of the parallel analytical engine `JuliaDB` (see *Chapter 9*), Julia presents a really exciting (and fast) alternative.

In this section, we will look at the application of some simple statistics involving data sources from the `RDatasets` package:

```
julia> mlmf = dataset("mlmRev", "Gcsemv"); size(mlmf)
(1905, 5)
```

We will use data from `mlmRev`, which is a group of datasets from the *Multilevel Software Review*. The `Gcsemv` dataset refers to the UK's GSCE exam scores.

This covers the results from 73 schools both in terms of examination and coursework. The data is not split by subject (only school and pupil) but the gender of the student is provided. Schools are listed via a categorical variable.

For more details, refer to <https://cran.r-project.org/web/packages/mlmRev/mlmRev.pdf>:

```
# Display the first 5 rows
julia> mlmf[1:5,:]
5x5 DataFrame
Row | School  Student  Gender  Written  Course
    | Cat...   Cat...   Cat...   Float64  Float64
1   | 20920    16       M        23.0     missing
2   | 20920    25       F        missing    71.2
3   | 20920    27       F        39.0      76.8
4   | 20920    31       F        36.0      87.9
5   | 20920    42       M        16.0      44.4
```

Now, use `describe()` to output the summary statistics for each column's values (except for `Gender`):

```
julia> describe(mlmf[!, :Student])
Summary Stats:
Length:          1905
Type:           CategoricalArrays.CategoricalValue{String, UInt16}
Number Unique:  649
```

```
julia> describe(mlmf[!, :School])
Summary Stats:
Length:          1905
Type:           CategoricalArrays.CategoricalValue{String, UInt8}
Number Unique:   73
julia> describe(mlmf[!, :Written])
Summary Stats:
Length:          1905
Missing Count:  202
Mean:            46.365238
Minimum:         0.600000
1st Quartile:   37.000000
Median:          46.000000
3rd Quartile:   55.000000
Maximum:         90.000000
Type:           Union{Missing, Float64}
julia> describe(mlmf[!, :Course])
Summary Stats:
Length:          1905
Missing Count:  180
Mean:            73.387449
Minimum:         9.250000
1st Quartile:   62.900000
Median:          75.900000
3rd Quartile:   86.100000
Maximum:         100.000000
Type:           Union{Missing, Float64}
```

As we can see, there are 73 schools and 649 students; because these are categorical values, statistics such as min/max, median, and mean have no relevance.

The written and coursework values refer to both genders (M and F) and both sets contain missing values, so we need to collect all the records with a value and split these by gender.

We will only concentrate on the written (exam) scores; those for the coursework will be left for you:

```
julia> writtenF =
       collect(skipmissing(mlmf[mlmf.Gender .== "F", :Written]));
julia> writtenM =
       collect(skipmissing(mlmf[mlmf.Gender .== "M", :Written]));
```

Notice the use of `skipmissing()` to do precisely that!

Calculate the mean and standard deviation of the two groups:

```
julia> (μWM, μWF) =
    round.((mean(writtenM), mean(writtenF)), digits=3)
(48.286, 45.005)
julia> (σWM, σWF) =
    round.((std(writtenM), std(writtenF)), digits=3)
(12.905, 13.535)
```

We can use these to apply a student t-test between the means with the (null) hypothesis that they are drawn from the same population.

We will need the numbers in each subset:

```
julia> (nWM, nWF) = (length(writtenM), length(writtenF))
(706, 997)
# and evaluate a completed standard deviation
julia> σW = sqrt((σWM*σWM) / (nWM - 1) + (σWF*σWF) / (nWF - 1))
0.6481955879108801
# then compute the t-statistic:
julia> tt = round(abs(μWM - μWF) / σW, digits=4)
0.9719
```

Looking at t-tables, we have  $p \sim 0.33$ ; 95% and  $\sim 0.06$ , 90%  $\sim 0.13$ .

Based on these values, we *reject* the null hypothesis and assert that there is a statistical difference between the means – that is, between the written scores between males and females.

A similar analysis is made in the accompanying code for the coursework. The t-value in this case is found to be 1.0708, again leading to a rejection of the null hypothesis.

Considering the mean scores, males do better in the written (exam) work while females are better in the coursework.

## Kernel densities

The preceding summary statistics show that there is a significant difference between marks for coursework and examinations and also compute the kernel densities of the two groups.

Kernel density is a technique that creates a smooth curve given a set of data. This can be useful if you want to visualize just the “shape” of the dataset, as an analog of a discrete histogram for continuous data.

To analyze these on the existing dataset, we will need to extract all the records of students and have scores in *both* categories; again, we do this by using the `completetcases()` routine.

There are 1,905 students from 73 schools throughout England – the schools are identified by an ID code within the dataset:

```
julia> using RDatasets, KernelDensity
julia> mlmf = dataset("mlmRev", "Gcsemv");
julia> df = mlmf[completecases(mlmf[!, [:Written, :Course]]), :]
1523×5 DataFrame
 Row | School  Student  Gender  Written  Course
     | Cat...   Cat...    Cat...   Float64  Float64
 _____|_____|_____|_____|_____|_____
 1 | 20920    27      F       39.0    76.8
 2 | 20920    31      F       36.0    87.9
 3 | 20920    42      M       16.0    44.4
 4 | 20920    101     F       49.0    89.8
 5 | 20920    113     M       25.0    17.5
 6 | 22520    1       F       48.0    84.2
 7 | 22520    7       M       46.0    66.6
```

We need to extract the value from the DataFrame as the element type is (still) the union of `Float64` and `Missing`; for convenience, let's define a macro to operate on the array:

```
macro F64(sym)
    quote
        Float64.(skipmissing(Array($sym)))
    end
end
```

Look at the coursework marks; calculate the kernel density and output the summary statistics:

```
julia> dc = @F64 df[!, :Course];
julia> kdc = kde(dc);
julia> summarystats(dc)
Summary Stats:
Length:          1523
Missing Count:  0
Mean:            73.381385
Minimum:         9.250000
1st Quartile:   62.900000
Median:          75.900000
3rd Quartile:   86.100000
Maximum:         100.000000
```

Now, repeat the same for the written (exam) marks:

```
julia> dw = @F64 df[!, :Written];
julia> kdw = kde(dw);
julia> summarystats(dw)
Summary Stats:
Length:          1523
Missing Count:   0
Mean:            46.502298
Minimum:         0.600000
1st Quartile:   38.000000
Median:          46.000000
3rd Quartile:   56.000000
Maximum:         90.000000
```

Finally, display the kernel densities for the two groups – that is, coursework and written:

```
julia> using PyPlot
julia> PyPlot.plot(kdc.x, kdc.density)
julia> PyPlot.plot(kdw.x, kdw.density, linestyle="--")
```

Here's the output:

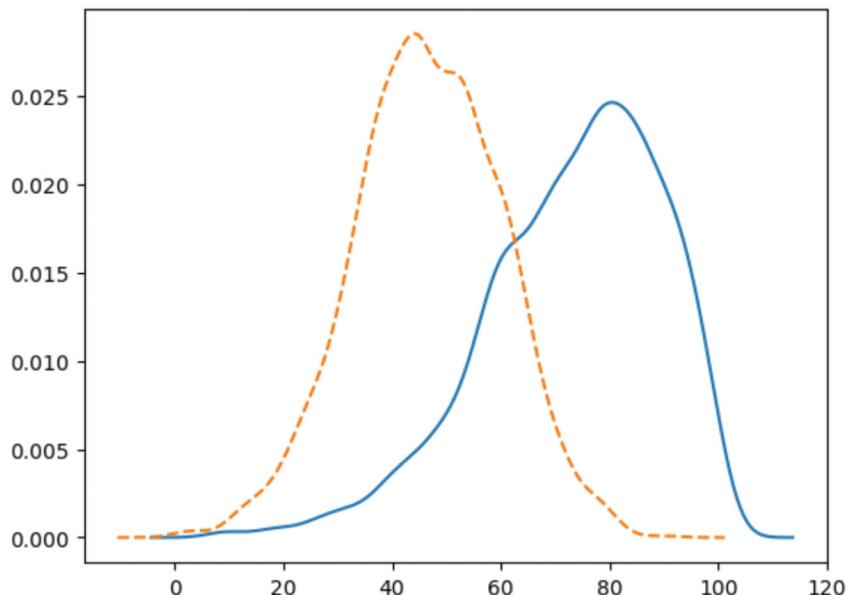


Figure 6.7 – Kernel densities between coursework and written work

This plot shows the kernel densities of the coursework (dotted) and written work (solid) for all schools with no discrimination between genders.

We will examine differences between schools in the next section.

## Testing hypothesis

So far, we have looked at differences between written work and coursework but not differentiated between various schools. Recall that a categorical variable is provided that references the school; this can be used to create a subset of the DataFrame.

To reduce the length of the results, we will restrict the analysis to schools with more than 40 students:

```
julia> using Statistics, DataFrames
julia> for subdf in groupby(df, :School)
       (size(subdf)[1] > 40) &&
       let
           sch = subdf[!, :School][1]
           msw = mean(subdf[!, :Written])
           msc = mean(subdf[!, :Course])
           nsz = size(subdf)[1]
           @printf "%10s : %8.4f %8.4f %3d\n" sch msw msc nsz
       end
   end
22520 : 35.4482 57.4580 56
60457 : 53.4773 85.9568 44
68107 : 44.9107 74.6750 56
68125 : 47.1556 77.5322 45
68137 : 28.2807 62.5373 83
68411 : 40.4615 59.4369 65
68809 : 42.7705 71.1115 61
```

Take two of these schools (#68107 and #68411) and investigate the difference in scores from each.

Again, we will apply a t-test but this time, we will be using a package from the JuliaStats group:

```
julia> using HypothesisTests
julia> df68107 = mlmf[mclf[!, :School] .== "68107", :];
julia> df68107cc =
       df68107[completescases(df68107[!, [:Written, :Course]]), :];
julia> df68411 = mlmf[mclf[!, :School] .== "68411", :];
julia> df68411cc =
       df68411[completescases(df68411[!, [:Written, :Course]]), :];
```

First, apply the test to written marks:

```
julia> df68107wri = @F64 df68107cc[!, :Written];  
julia> df68411wri = @F64 df68411cc[!, :Written];  
julia> UnequalVarianceTTest(df68107wri, df68411wri)  
Two sample t-test (unequal variance)  
-----  
Population details:  
    parameter of interest: Mean difference  
    value under h_0: 0  
    point estimate: 4.44918  
    95% confidence interval: (-0.1837, 9.082)  
Test summary:  
    outcome with 95% confidence: fail to reject h_0  
    two-sided p-value: 0.0596  
Details:  
    number of observations: [56, 65]  
    t-statistic: 1.9032531870995715  
    degrees of freedom: 109.74148002018097  
    empirical standard error: 2.337668920946911
```

Repeat the analysis for the coursework:

```
julia> df68107cou = @F64 df68107cc[!, :Course];  
julia> df68411cou = @F64 df68411cc[!, :Course];  
julia> UnequalVarianceTTest(df68107cou, df68411cou)  
Two sample t-test (unequal variance)  
-----  
Population details:  
    parameter of interest: Mean difference  
    value under h_0: 0  
    point estimate: 15.2381  
    95% confidence interval: (10.63, 19.85)  
Test summary:  
    outcome with 95% confidence: reject h_0  
    two-sided p-value: <1e-08  
Details:  
    number of observations: [56, 65]  
    t-statistic: 6.541977424916656  
    degrees of freedom: 118.13175559744462  
    empirical standard error: 2.329276904111456
```

Here, we reject the null hypothesis for coursework marks but not for written ones – that is, exam marks. This would seem to imply that there is a significant difference in the marking of the coursework between these two schools.

## Summary

In this chapter, we looked at how Julia handles data input and output. We discussed how to perform simple I/O via the console and extended that to operating on text-based files on disk, extending this to structured data in the form of CSV and other DLM files. After, we looked at performing I/O operations on binary files and files formatted via the HDF5 and JLD file schemas.

Then, we considered interacting with datasets from Julia modules such as those contained in the RDatasets package and we saw how handling these leads naturally to Julia's implementation of "R" style DataFrames, such as pandas in Python, and also of its sibling the time array.

Finally, we looked at how to visualize the datasets and further analyze the values by applying some simple statistical routines by computing descriptive metrics, kernel densities, and hypothesis testing.

Later in this book, we will introduce methods for dealing with data contained within databases and other sources available more widely over the internet but first, we will turn to the large topic of scientific methods applied to data.



# 7

## Scientific Programming

Julia was initially designed as a language aimed at finding solutions to problems arising from science and mathematics.

Current scripting languages then, and to some extent now, were slow especially when dealing with devectorized (“looping”) code, and resorted to the use of compile code (for example, written in C) to achieve acceptable executing times. This led to the “two-language” approach where analysis is made using the scripting language, whereas code needs to be compiled into a second language, usually C, in order to achieve enterprise performance.

We have seen that Julia compiles its sources to the appropriate machine code using **just-in-time (JIT)** compilation from LLVM and so achieves execution times comparable with those of C and Fortran. Naturally, the applications of Julia in the fields of scientific programming are many and varied, and in a single chapter, I can do no more than point the reader to some of the more elemental examples.

However, in a fashion like `JuliaStats` in statistics, a number of community groupings have been formed (at present, perhaps, bewilderingly so), and these will be noted as we go along.

The Julia base and standard libraries contain most of the Python modules, NumPy, and some of SciPy. In addition, Julia has a great wealth of packages for scientific programming, both specialist and general purpose.

Among the packages are those that support finance, astronomy, bioinformatics, and **machine learning (ML)**, plus many others. The general-purpose packages may be native implementations of low-level data structures or wrappers around open source libraries.

In this chapter, I am going to concentrate on a few areas between the specialist and general-purpose packages.

It is not an exhaustive list, and now with version 1.0, there is a wealth of packages (currently over 2,000 registered ones) that embrace many diverse disciplines.

This chapter will cover the following topics:

- Linear algebra
- Signal processing
- Differential equations (DEs)
- Calculus
- Optimization
- Stochastic simulations

Since Julia was designed with scientific programming in mind, a large number of packages are now available covering a variety of topics. In this chapter, we can only just skim the surface and not even scratch it.

For some of the more complex topics, I suggest you look at the *JuliaDynamics* group (<https://juliadynamics.github.io/JuliaDynamics>).

## Linear algebra

Linear algebra is one of the areas where the existing code has moved from BASE to STDLIB, so any work now needs to be prefaced with using linear algebra.

Linear algebra is a branch of mathematics concerning vector spaces and linear mappings between such spaces. It includes the study of lines, planes, and subspaces but is also concerned with properties common to all vector spaces and encompasses matrices and multidimensional arrays.

Julia has excellent support for the latter, mapping as it does higher-order arrays to 1D vectors, reinterpreted via a set of shape parameters. This means that an array, regardless of its arity, can be indexed as a 1D vector provided that that index does not exceed the total bounds of the array. It is worth noting also that for matrices, the apparent order is via columns, and for 3D arrays, from planes, then columns, and so on.

The normal rules for manipulating arrays are available in Julia, and we will not spend too much time on them.

Arrays can be defined by the `[]` convention; items on a row are separated by whitespace and individual rows by semicolons.

There are alternative methods to define arrays of a large number of items, which we will meet in examples later.

The element type is determined by the highest common type, so in the array shown next, one containing `Int64` values is created; if but one of these were specified as (say) 1.0, then a `Float64` value would be the result:

```
# Define a matrix A and calculate its determinant
julia> using LinearAlgebra
julia> A = [1 -2 2; 1 -1 2; -1 1 1];
julia> det(A)
3.0
#= Create a column vector b and perform a matrix division into A =#
julia> b = [5, 7, 5];
julia> v = A\b
3-element Vector{Float64}:
 1.0
 2.0
 4.0
# Display the transpose of v
# Take note of the new type signature of the transpose
julia> transpose(v)
1×3 transpose(::Vector{Float64}) with eltype Float64:
 1.0    2.0    4.0
```

Slicing and dicing of arrays remains the same under version 1.

*Note:* The use of `:` refers to the entire row (or column), and so on:

```
julia> A1 = A[:, 2:3]
3×2 Matrix{Int64}:
 -2    2
 -1    2
  1    1
```

Note again the new, and different, type signature of the following:

```
julia> (A1\b)'
1×2 adjoint(::Vector{Float64}) with eltype Float64:
 1.27586  3.93103
julia> A2 = A[1:2, :]; b2 = b[1:2];
julia> (A2\b2)'
1×3 adjoint(::Vector{Float64}) with eltype Float64:
 1.8  2.0  3.6
```

## Matrix decomposition

In this section, we are going to consider the process of factorizing a matrix into a product of a number of other matrices. This is termed **matrix decomposition** and proves useful in several classes of problems.

The \ operator hides how the problem is actually solved. Depending on the dimensions of the A matrix (say), different methods are chosen to solve the problem. An intermediate step in the solution is to calculate a factorization of the A matrix. This is a way of expressing A as a product of triangular, unitary, and permutation matrices.

When the matrix is square, it is possible to factorize it into a pair of upper and lower diagonal matrices (U, L), together with a P permutation matrix such that  $A = PLU$ .

Working with the A matrix defined previously, then its LU decomposition is created like so:

```
julia> Alu = lu(A)
LU{Float64, Matrix{Float64}, Vector{Int64}}
L factor:
3x3 Matrix{Float64}:
 1.0   0.0   0.0
 1.0   1.0   0.0
 -1.0  -1.0   1.0
U factor:
3x3 Matrix{Float64}:
 1.0  -2.0   2.0
 0.0   1.0   0.0
 0.0   0.0   3.0
```

We can check that multiplying the L and U components generates the original matrix:

```
julia> Alu.L*Alu.U
3x3 Matrix{Float64}:
 1.0  -2.0   2.0
 1.0  -1.0   2.0
 -1.0  1.0   1.0
```

Now, we will apply what has just been discussed in the solution of linear simultaneous equations.

## Simultaneous equations

A set of  $n$  equations in  $n$  unknown quantities will have a unique solution, provided that one of the equations is not a multiple of another. In the latter case, the system is termed *degenerate* since effectively, we only have  $n-1$  equations.

Clearly,  $n$  is a positive number, with  $n \geq 2$ , since  $n = 1$  is trivial.

The solution of such equations is obtained by matrix methods such as those demonstrated previously.

Consider the case of the following system of equations:

$$\begin{aligned}x - 2y + 2z &= 5 \\x - y + 2z &= 7 \\-x + y + z &= 5\end{aligned}$$

In matrix notation, we write this as  $Av = b$ , where  $v$  is the  $[x \ y \ z]$  vector of unknowns,  $A$  is a matrix of coefficients, and  $b$  is a vector of constants on the right-hand side; that is,  $A = [1 \ -2 \ 2; \ 1 \ -1 \ 2; \ -1 \ 1 \ 1]$  and  $b = [5, \ 7, \ 5]$ .

These are the very two arrays that we used in the first section and observed that they are non-degenerate; that is, the  $A$  matrix has a computable determinant, and so the system of equations has a solution.

The solution is derived by multiplying each side of the equation  $Av = b$  by the inverse of  $A$ , giving  $v = \text{inv}(A)*b$ , since  $\text{inv}(A)*A$  is the identity matrix, and this we have already computed; that is,  $(x, \ y, \ z) \Rightarrow (1.0, \ 2.0, \ 4.0)$ .

We can also use LU factorization in solving sets of linear equations. In fact, the routines internally use these very operations in returning their results.

Consider as a second example the following equations:

$$\begin{aligned}x - 2y + 2z &= 5 \\x - y + 2z &= 3 \\-x + y + z &= 6\end{aligned}$$

This has the same matrix of coefficients but a different vector of constants (on the right-hand side), and so we get the following output:

```
julia> A = [1 -2 2; 1 -1 2; -1 1 1];
julia> b = [5; 3; 6];
julia> Alu = lu(A);
# So Ax = b
# Implies x = inv(U)*inv(L)*b
julia> (x,y,z) = inv(Alu.U)*inv(Alu.L)*b;
julia> @show(x,y,z);
x = -5.0
y = -2.0
z = 3.0
# Checking that the answer is correct
julia> x - 2y + 2z
5.0
julia> x - y + 2z
3.0
```

```
julia> -x + y + z
6.0
```

It is worth noting that while it is possible to call specific factorization methods explicitly (such as `lu`), Julia has the `factorize(A)` function which will compute a convenient factorization, including LU, Cholesky, Bunch-Kaufman, and triangular, based upon the type of the input matrix.

This function is now part of the `LinearAlgebra` STDLIB module, so a `using` statement must be issued before getting help on the function:

```
julia> using LinearAlgebra
help?> factorize
Properties of ...                                         Type of Factorization
Positive-definite                                         Cholesky (see cholesky)
Dense Symmetric/Hermitian                                Bunch-Kaufman
                                                       (see bunchkaufman)
Sparse Symmetric/Hermitian                               LDLT   (see ldlt)
Triangular                                                 Triangular
Diagonal                                                 Diagonal
Bidiagonal                                                Bidiagonal
Tridiagonal                                              Tridiagonal
Symmetric real tridiagonal                            LDLT   (see ldlt)
General square                                           LU     (see lu)
General non-square                                     QR     (see qr)
```

For instance, if `factorize` is called on a Hermitian positive-definite matrix, then it will return a Cholesky factorization.

Also, there are ! (bang) versions of functions such as `lu!()`, `qr!()`, and so on that will compute the decompositions in place to conserve memory requirements when dealing with a large number of equations. The reader is referred to Julia's online help for more information on these.

## Eigenvalues and eigenvectors

An eigenvector or characteristic vector of a square matrix  $A$  is a nonzero vector  $v$  that, when the  $A$  matrix multiplies  $v$ , gives the same result as when some scalar multiplies  $v$ . The scalar multiplier is usually denoted by  $\lambda$ .

That is,  $Av = \lambda v$  and is called the eigenvalue or characteristic value  $\sigma_A$  corresponding to  $v$ .

Considering our previous set of three equations, this will yield three `eigvecs` instances (eigenvectors) and corresponding `eigvals` instances (eigenvalues):

```
julia> using LinearAlgebra
julia> A = [1 -2 2; 1 -1 2; -1 1 1];
```

```
# Compute the eigenvalues of A
# (These are complex numbers)
julia> U = eigvals(A)
3-element Vector{ComplexF64}:
-0.2873715369435107 + 1.3499963980036567im
-0.2873715369435107 - 1.3499963980036567im
1.5747430738870216 + 0.0im
# ... and the eigenvectors
julia> V = eigvecs(A)
3x3 Matrix{ComplexF64}:
-0.783249-0.0im      -0.783249+0.0im      0.237883+0.0im
-0.493483-0.303862im -0.493483+0.303862im  0.651777+0.0im
 0.0106833+0.22483im  0.0106833-0.22483im  0.720138+0.0im
```

The eigenvectors are the columns of the V matrix:

```
julia> A*V[:,1] - U[1]*V[:,1]
3-element Vector{ComplexF64}:
 2.220446049250313e-16 + 2.220446049250313e-16im
 -1.1102230246251565e-16 + 0.0im
 -2.7755575615628914e-16 - 1.3877787807814457e-16im
```

That is, all the real and imaginary parts are of the  $10^{-16}$  order, so this is in effect a zero matrix of complex numbers.

### ***Why do we wish to compute eigenvectors?***

Eigenvectors make understanding linear transformations easy, as they are the directions along which a linear transformation acts simply by “stretching/compressing” and/or “flipping,” and they give the factors by which this compression occurs.

They provide another way to affect matrix factorization using **singular value decomposition (SVD)** with **svdfact()**.

They are useful in the study of chained matrices, such as the cat and mouse example we saw in an earlier chapter.

They arise in several studies of a variety of dynamic systems.

We will finish this section by considering a dynamic system given by the following:

$$\begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned}$$

Here,  $x'$  and  $y'$  are the derivatives of  $x$  and  $y$  with respect to time, and  $a, b, c$ , and  $d$  are constants.

This kind of system was first used to describe the growth of the population of two species that affect one another and are termed Lotka-Volterra equations.

We may consider that species  $x$  is a predator of species  $y$ .

The more of  $x$ , the less of  $y$  will be around to reproduce, but if there is less of  $y$  then there is less food for  $x$ , so less of  $x$  will reproduce.

Then, if less of  $x$  is around, this takes the pressure off  $y$ , which increases in number, but then there is more food for  $x$ , so  $x$  increases.

It also arises when you have certain physical phenomena, such as a particle in a moving fluid where the velocity vector depends on the position along the fluid. Solving this system directly is complicated, and we will return to it in the section on DEs. However, suppose that we could do a transform of variables so that instead of working with  $x$  and  $y$ , you could work with  $p$  and  $q$ , which depend linearly on  $x$  and  $y$ .

That is,  $p = \alpha x + y$  and  $w = x + \delta y$ , for some constants  $\alpha$  and  $\delta$ .

The system is transformed into something like this:  $p' = \kappa p$  and  $q' = \lambda q$ .

That is, you can “decouple” the system so that you are now dealing with two independent functions.

Then, solving this problem becomes rather easy:  $p = A \exp(\kappa t)$  and  $q = B \exp(\lambda t)$ .

Then, you can use the formulas for  $z$  and  $w$  to find expressions for  $x$  and  $y$ . This results precisely in finding two linearly independent eigenvectors for the  $[a \ c; b \ d]$  matrix where  $p$  and  $q$  correspond to the eigenvectors and eigenvalues. By taking an expression that “mixes”  $x$  and  $y$  and “decoupling” it into one that acts independently on two different functions, the problem becomes a lot easier.

This can be reduced to a generalized eigenvalue problem by clever use of algebra at the cost of solving a larger system. The orthogonality of the eigenvectors provides a decoupling of the DEs so that the system can be represented as a linear summation of the eigenvectors. The eigenvalue problem of complex structures is often solved using finite element analysis, but it neatly generalizes the solution to scalar-valued vibration problems.

*Note:* We will be looking at a solution of a triple Lotka-Volterra set of equations later in this chapter.

## High-order algebraic equations

While we are here, how do we solve an equation involving the powers of a single dependent variable; for example, (say) the quintic  $3x^5 + 2x^4 + x^3 + 7x + 1 = 0$ ?

In Julia, this is very simple as it can be achieved using a couple of packages to define the polynomial and then to calculate the roots:

```
julia> using Polynomials, Roots
julia> p = Polynomial([1,7,-4,1,2,3])
Polynomial(1 + 7*x - 4*x^2 + x^3 + 2*x^4 + 3*x^5)
```

```
julia> round.(roots(p), digits=4)
5-element Vector{ComplexF64}:
-1.0315 - 1.131im
-1.0315 + 1.131im
-0.1326 + 0.0im
 0.7644 - 0.6992im
 0.7644 + 0.6992im
```

There is one real root and a couple of pairs of complex ones in the form of  $a \pm b \cdot im$ .

Note that the vector specification is in the order from the constant, followed by coefficients for increasing powers of  $x$ .

## Signal processing

Signal processing is the art of analyzing and manipulating signals arising in many fields of engineering. It deals with operations on or analysis of analog as well as digitized signals, representing time-varying or spatially varying physical quantities.

Julia has the functionality for processing signals built into the standard library along with a growing set of packages, and the speed of Julia makes it especially well suited to such analysis.

We can differentiate between 1D signals, such as audio signals, **electrocardiogram (ECG)** signals, variations in pressure and temperature, and so on, and 2D resulting in imagery from video and satellite data streams. In this section, I will mainly focus on the former, but the techniques carry over in a straightforward fashion to the 2D cases.

### Frequency analysis

A signal is simply a measurable quantity that varies in time and/or space. The key insight of signal processing is that a signal in time can be represented by a linear combination of sinusoids at different frequencies.

There exists an operation called the **Fourier transform (FT)**, which takes an  $x(t)$  function of time that is called the time-domain signal and computes the weights of its sinusoidal components. These weights are represented as an  $X(f)$  function of frequency called the frequency-domain signal.

The FT takes a continuous function and produces another continuous function as a function of the frequencies of which it is composed. In digital signal processing, since we operate on signals in discrete time, we use the **discrete FT (DFT)**.

This takes a set of  $N$  samples in time and produces weights at  $N$  different frequencies. Julia's signal processing library, as with most common signal processing software packages, computes DFTs by using a class of algorithms known as **fast FTs (FFTs)**.

### ***Smoothing and filtering***

First, we will generate a signal from three sinusoids and visualize it:

```
julia> using Plots, FFTW
julia> fq = 500.0;
julia> N = 512;
julia> T = 6 / fq;
julia> t = collect(range(0, stop=T, length=N));
julia> x1 = sin.(2π * fq * t);
julia> x2 = cos.(8π * fq * t);
julia> x3 = cos.(16π * fq * t);
julia> x = x1 + 0.4*x2 + 0.2*x3;
```

Now, use the `rfft` function (the real FFT function) since our input signal is composed entirely of real numbers, as opposed to complex numbers. This allows us to optimize by only computing the weights for frequencies from 1 to  $N/2+1$ .

The higher frequencies are simply a mirror image of the lower ones, so they do not contain any extra information. We will need to use the absolute magnitude (modulus) of the output of `rfft()` because the outputs are complex numbers. Right now, we only care about the magnitude and not the phase of the frequency domain signal:

```
julia> X = rfft(x);
julia> sr = N / T;
julia> fd = LinRange(0, sr/2, Int(N/2) + 1)
julia> plot(fd, abs(X)[1:N/8])
```

This transforms the time domain representation of the signal (amplitude versus time) into one in the frequency domain (magnitude versus frequency).

The following figure shows the two representations:

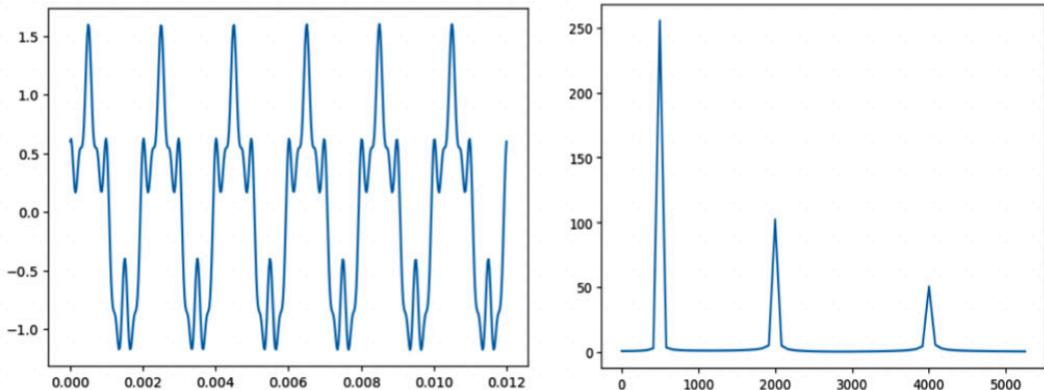


Figure 7.1 – FFT of a sinusoidal function

Now, we can add some high-frequency noise to the signal:

```
julia> ns = 0.1*randn(length(x));
julia> xn = x + ns;
```

Then, use a convolution procedure in the time domain to attempt to remove it. In essence, this is a **moving average (MA)** smoothing technique.

We define a 16-element window and use a uniform distribution, although it might be sensible to use a Gaussian or parabolic one that would weigh the nearest neighbors more appropriately:

```
julia> M = 16;
julia> xm = ones(Float64, M) / M;
```

We then apply a convolution function in the DSP package to `xm` and `xn`:

```
julia> using DSP
julia> xf = conv(xn, xm);
julia> t = [0:length(xf) - 1] / sr
```

It is important that the sum of the weights is 1.

The following figure shows the noisy signal together with the filtered one:

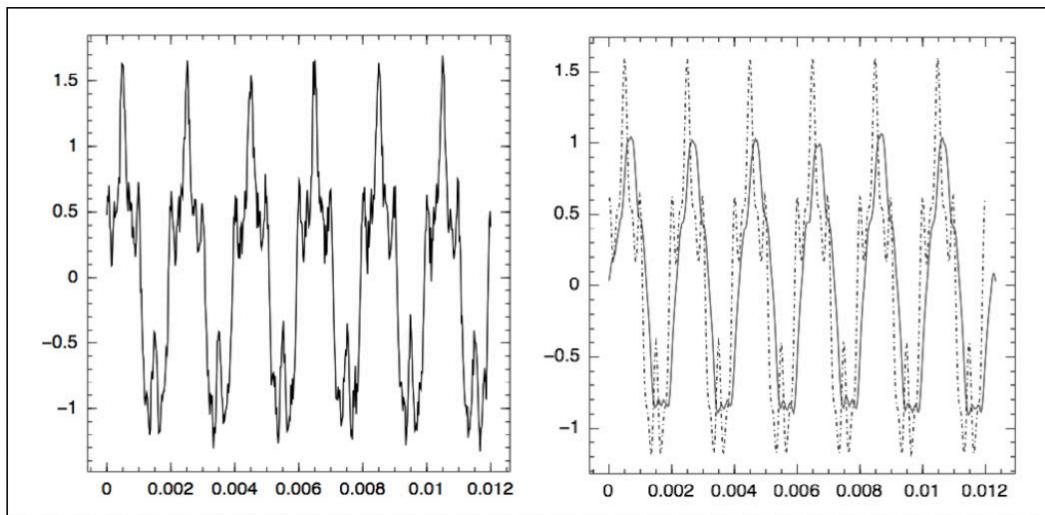


Figure 7.2 – Comparison of noisy and filtered signals of the sinusoid

The main carrier wave is recovered and the noise is eliminated, but given the size of the window chosen, the convolution has a drastic effect on the higher-frequency components.

### **Digital signal filters**

MA filters (convolution) work well for removing noise if the frequency of the noise is much higher than the principal components of a signal.

A common requirement in RF communications is to retain parts of the signal but to filter out the others. The simplest filter of this sort is a low-pass filter. This is a filter that allows sinusoids below a critical frequency to go through unchanged while attenuating signals above the critical frequency. Clearly, this is a case where the processing is done in the frequency domain.

Filters can also be constructed to retain sinusoids above the critical frequency (high pass), or within a specific frequency range (medium band). Julia provides a set of signal processing packages as the DSP.jl group (<https://github.com/JuliaDSP/DSP.jl>), and we will apply some of the routines to filter out noise on the signal we created in the previous section:

```
julia> using DSP
julia> responsetype = Lowpass(0.2);
julia> prototype = Elliptic(4, 0.5, 30);
julia> tf = convert(PolynomialRatio,
                     digitalfilter(responsetype prototype))
```

```
julia> numerator_coefs = coefb(tf);  
julia> denominator_coefs = coefa(tf);
```

This constructs a fourth-order elliptic low-pass filter with normalized cut-off frequency 0.2, 0.5 dB of passband ripple, and 30 dB attenuation in the stopband.

Then, the coefficients of the numerator and denominator of the transfer function will be this:

```
julia> responsetype = Bandpass(10, 40; fs=1000);  
julia> prototype = Butterworth(4);  
julia> xb = filt(digitalfilter(responsetype,prototype),x);  
julia> plot(xb)
```

This code filters the data in the `x` signal, sampled at 1,000 Hz, with a fourth-order Butterworth bandpass filter between 10 and 40 Hz.

The resultant signal is displayed as follows:

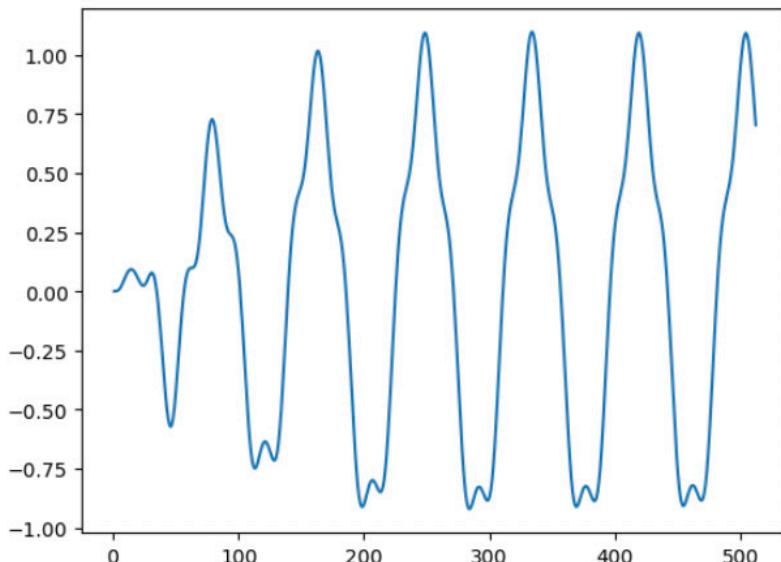


Figure 7.3 – Application of a Butterworth bandpass filter

While being cleaner than convolution, this still affects high frequencies.

Also, while the bandpass filter is infinite in its extent, the one constructed is truncated, and this means that the initial portion of the signal is modulated.

## Image convolutions

Frequency-based methods can be applied to 2D signals, such as those from video and satellite data streams. High-frequency noise in imagery is termed “speckle.” Essentially, due to the orthogonality of the FFT, processing involves applying a series of row-wise and column-wise FFTs independently of each other.

The DSP package has routines to deal with both 1D and 2D cases. Also, the convolution techniques we looked at in the *Frequency analysis* section are often employed in enhancing or transforming images, and we will finish by looking at a simple example using a 3x3 convolution kernel.

The kernel needs to be zero-summed; otherwise, the histogram range of the image is altered. We will look at the *Lena* image that is provided as a 512x512 PGM image:

```
julia> cd(ENV["HOME"] * "/MJ2/DataSources");
julia> img = open(Files/lena.pgm");
julia> magic = chomp(readline(img));

julia> params = chomp(readline(img));
julia> pm = split(params)

# Remember the GSD
julia> try
    global wd = parse(Int64,pm[1]);
    global ht = parse(Int64,pm[2]);
catch
    error("Can't figure out the image dimensions")
end
# Version 1.0 way of defining a byte array
# readbytes!() will read in place
julia> data = Array{UInt8,2}(undef,wd,ht)
julia> readbytes!(img, data, wd*ht);
julia> data = reshape(data,wd,ht);
julia> close(img);
```

The preceding code reads the PGM image and stores the imagery as a byte array in data, reshaping it to be wd by ht.

Now, we define the two 3x3 Gx and Gy kernels:

```
julia> Gx = [1 2 1; 0 0 0; -1 -2 -1];
julia> Gy = [1 0 -1; 2 0 -2; 1 0 -1];
```

The following loops over blocks of the original image applying  $G_x$  and  $G_y$ , constructs the modulus of each convolution, and outputs the transformed image as  $dout$ , again as a PGM image. We need to be a little careful that the imagery is still preserved as a byte array:

```
julia> dout = copy(data);
julia> for i = 2:wd-1
    for j = 2:ht-1
        temp = data[i-1:i+1, j-1:j+1];
        x = sum(Gx.*temp)
        y = sum(Gy.*temp)
        p = Int64(floor(sqrt(x*x + y*y)))
        dout[i,j] = (p < 256) ? UInt8(p) : 0xff
    end
end

# ... and output the result
julia> DS = ENV["HOME"]*"/MJ2/DataSources"
julia> out = open("$DS/Files/lenaX.pgm", "w");
julia> println(out,magic);
julia> println(out,params);
julia> write(out,dout);

julia> close(out);
```

We can display the convoluted image using the `ImageView` package we used previously to view the Julia set:

```
julia> using Images. ImageView
julia> img = load("$DS/Files/lenaX.pgm")
julia> imshow(img)
```



Figure 7.4 – Original and convoluted images of “Lena”

We will leave the discussion of imagery but will be returning to it for a more extensive look in the final section of the next chapter. Now, I will switch to the topic of the solution of DEs

## DEs

DEs are those that have terms that involve the rates of change of variates as well as the variates themselves. They arise naturally in a number of fields, notably dynamics. When the changes are with respect to one dependent variable, most often the systems are called **ordinary DEs (ODEs)**. If more than a single dependent variable is involved, then they are termed **partial DEs (PDEs)**.

Julia has several packages that aid the calculation of differentials of functions and solve systems of DEs; these are grouped together under the *JuliaDiffEq* community group and are now encapsulated as a suite for numerically solving DEs covered by an envelope package, `DifferentialEquations.jl`, whose purpose is to supply efficient Julia implementations of solvers for a wide variety of DEs.

Equations covered by this package include:

- Discrete equations (function maps, discrete stochastic simulations)
- ODEs
- Split and partitioned ODEs (symplectic integrators, implicit-explicit (IMEX) methods)
- Stochastic ODEs (SODEs or SDEs)
- Random DEs (RODEs or RDEs)
- Differential algebraic equations (DAEs)
- Delay DEs (DDEs)
- Mixed discrete and continuous equations (hybrid equations, jump diffusions)
- Stochastic PDEs (SPDEs) and PDEs
- (S)PDEs with both finite difference and finite element methods

We will focus on ODEs here.

A fuller discussion of handling other types of DEs is provided by the SciML project via the web page at <https://github.com/SciML/DifferentialEquations.jl>.

## ODEs

To look first at the simplest of the aforementioned equations (ODEs), we will use the Lotka-Volterra species equations referred to earlier, but to make the model a little more interesting, we will add an intermediate species that is eaten by the predator and itself preys on a third species.

Some examples might be three-species ecosystems, such as owl-snake-mouse, foxes-rabbits-vegetation, and falcon-sparrow-worm.

It is implicit in these models that the predators only eat their prey, such as pandas eating bamboo and koalas eating eucalyptus leaves.

We can write this system as a coupled set of three first-order DEs:

$$\begin{aligned}x' &= a*x - b*x*y \\y' &= -c*y + d*x*y - e*y*z \\z' &= -f*z + g*y*z\end{aligned}$$

where  $x$ ,  $y$  and  $z$  are the three species

and we require  $a,b,c,d,e,f,g > 0$

In the preceding equations,  $a$ ,  $b$ ,  $c$ , and  $d$  are in the two-species Lotka-Volterra equations,  $e$  represents the effect of predation on species  $y$  by species  $z$ ,  $f$  represents the natural death rate of the predator  $z$  in the absence of prey, and  $g$  represents the efficiency and propagation rate of the predator  $z$  in the presence of prey.

If we let all the parameters have a value of one and represent the species as a vector  $u$ , this translates to the following set of linear equations, defined as a Julia function, `ff()`:

```
function ff(d,u,p,t)
    d[1] = u[1] - u[1]*u[2]
    d[2] = -u[2] + u[1]*u[2] -u[2]*u[3]
    d[3] = -u[3] + u[2]*u[3]
end
```

The vector  $d$  is the derivatives of  $u$  and  $t$ , an array corresponding to the timespan.

Given this function, this can be solved once the initial conditions are set and the time range fixed, using the `OrdinaryDiffEq` package (itself part of `DifferentialEquations.jl`):

```
# Setup the initial conditions and the time range
julia> using OrdinaryDiffEq
julia> u0 = [0.5; 1.0; 2.0];
julia> tspan = (0.0,10.0);
# Define the ODE problem, this merely sets up the problem
# which itself is solved by calling the solve routine
julia> prob = ODEProblem(ff,u0,tspan)
ODEProblem with uType Array{Float64,1} and tType Float64. In-place:
truetimespan: (0.0, 10.0)
u0: [0.5, 1.0, 2.0]
```

`OrdinaryDiffEq.jl` contains some good “go-to” choices for ODEs:

- `Auto Tsit5(Rosenbrock23())` handles both stiff and non-stiff equations. This is a good algorithm to use if you know nothing about the equation.
- `BS3()` for fast low-accuracy non-stiff equations.
- `Tsit5()` for standard non-stiff equations. This is the first algorithm to try in most cases.

- `Vern7()` for high-accuracy non-stiff equations.
- `Rodas4()` for stiff equations with Julia-defined types, events, and so on.
- `radau()` for really high-accuracy stiff equations
- (requires additionally installing `ODEInterfaceDiffEq.jl`).

We need to specify which solver to use; a simple choice is `Tsit5()`:

```
julia> u = solve(prob, Tsit5());
```

For those bored with using `PyPlot` for displaying graphs, I will use the `Plots` API, together with the `GR` backend (its default); these plus other visualization examples will be discussed in the next chapter.

The following code uses the `Plots` API to plot the array; the result is shown in *Figure 7.5*:

```
julia> using Plots
julia> styles = [:solid; :dash; :dot]
julia> N = length(styles)
julia> styles = reshape(styles, 1, N)
# styles is now a 1xN Vector
```

Because there is a name clash with the routine `plot`, defined in both `PyPlot` and `GR`, it may be necessary to fully qualify the function to call the appropriate routine :

```
julia> Plots.plot(u, line = (2, styles))
```

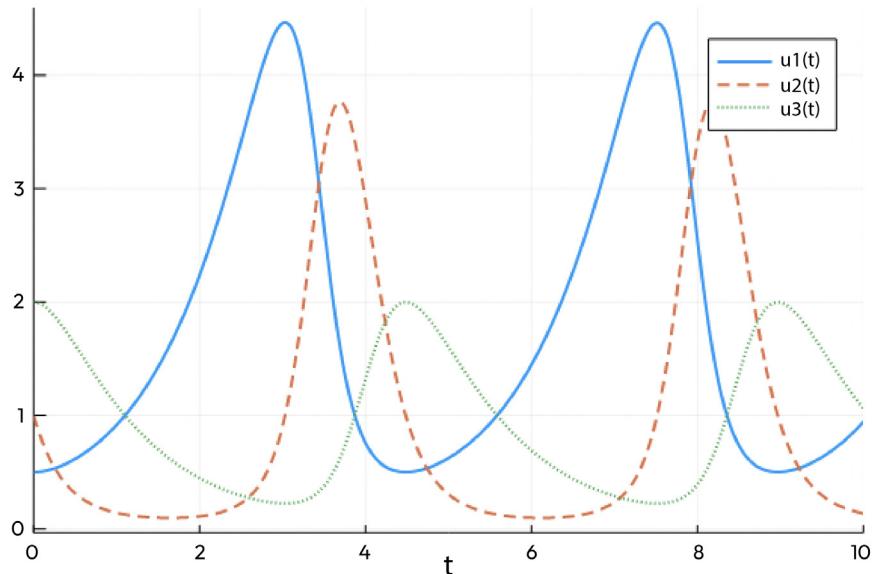


Figure 7.5 – Solution of the Lotka-Volterra problem for three species

This model assumes the  $z$  species does not directly eat  $x$ .

*This might not be true, for example, for the owl-snake-mouse ecosystem, where owls also eat mice; so, in this case, we would add an extra term:  $x' = a^*x - b^*x^*y - h^*xz$ .*

Upon redoing the model as  $d[1] = p[1]^*u[1] - p[2]^*u[1]^*u[2] - p[8]^*u[1]^*u[3]$ , there is an additional constraint that the sum of  $p_5$  and  $p_8$  must be 1.0, to save the first species from overeating.

The solution is given in the Jupyter workbook accompanying the chapter, and we see that apparently, the peak populations of the three species are little altered by the extra term, save that the periodicity is almost doubled.

## Simulating a (real) pendulum

For a second example, I want to consider the solution of a simple pendulum.

This is modeled in the equation  $\theta'' = \frac{-mg \sin(\theta)}{L}$  where  $m$  is the mass of the pendulum bob,  $L$  is its length,  $g$  is the gravitational constant of acceleration, and  $\theta$  is the angular displacement of the pendulum.

The classic (*analytic*) approach is found by approximating  $\sin(\theta)$  by  $\theta$ , resulting in the well-known sinusoidal solution, which holds for small initial displacements of  $\theta$ .

However, what happens if  $\theta$  is not small? That's what we will look at here.

For the solution, we will use one of the routines from the `DifferentialEquations` package:

```
# Rescale time to include 'g/L'
julia> using DifferentialEquations
julia> function pendulum(du, u, t, p)
    du[1] = u[2]
    du[2] = -sin(u[1])
end
julia> u0 = [3.0, 0.0];           # initial state vector
julia> tt = (0.0, 10.0*pi);      # time interval
julia> ps = [1.0];                # parameters
```

Here,  $u$  is a 2-vector equivalent to the displacement  $\theta$  and its angular velocity  $d\theta$ .

The initial position of the bob is high, close to  $\pi$  radians above the vertical, and the velocity at the time of release is 0.

We can solve the twin single ODEs using the `ODEProblem()` routine and plot the solution using the `Plots` package.

We will compute the trajectory for time 0 to  $10\pi$ :

```
julia> prob = ODEProblem(pendulum, u0, tt, ps)
ODEProblem with uType Vector{Float64} and tType Float64. In-place:
true
timespan: (0.0, 31.41592653589793)
u0: 2-element Vector{Float64}
3.0
0.0
julia> using Plots
julia> sol = solve(prob);
julia> plot(sol, layout = (2, 1))
```

One of the nice features when using the `Plots` API is that it can handle the `sol` solution directly. Note that `ODEProblem` does *NOT* create a solution for equal intervals in time, so using other graphics packages requires a little extra work.

The solutions for angular displacement and velocity are shown in *Figure 7.6*:

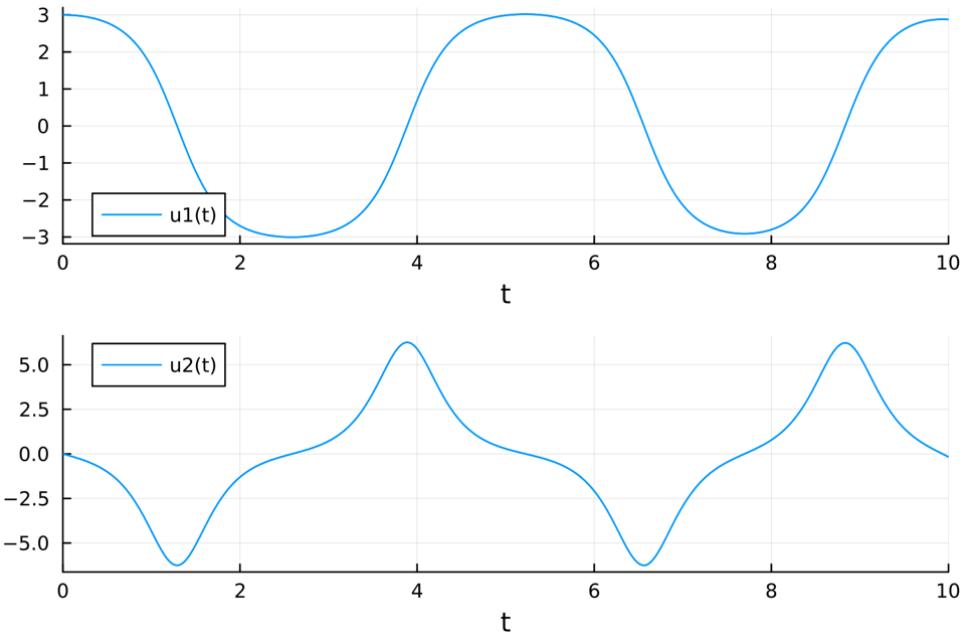


Figure 7.6 – Angular displacement and velocity for the “simple” pendulum

We see that the  $u_1$  displacement is a rather flattened sine wave, and the velocity is markedly different from the classic cosine.

## Catastrophic equations

Non-linear ODEs differ from their linear counterparts in several ways. They may contain functions, such as sine, log, and so on, of the independent variable and/or higher powers of the dependent variable and its derivatives.

The example I will look at now is a non-linear system arising from the temporal development of a chemical reaction, which is one displaying a set of bifurcating solutions characteristic of catastrophe theory.

Consider the reaction will generate heat by the  $e^{-E/RT}$  Arrhenius function and lose heat at the boundary proportional to the  $(T - T_0)$  temperature gradient.

It is possible to write  $e^{-E/RT} \sim e^{-E/RT_0} (T - T_0)$ , which means at a low temperature, behavior is proportional to the exponential value of the temperature difference:

$$\Theta = T - T_0.$$

That is,  $\frac{d\theta}{dt} = e^{-\theta} - a\theta$ ,

where the time  $t$  is scaled by the Arrhenius term  $e^{-E/RT_0}$  and  $a$  is a constant (viz., the Newtonian cooling coefficient).

Although the ODE package is capable of handling non-linear systems, in this case, I will look at a solution that utilizes an alternative `Sundials` package.

`Sundials` is a wrapper around a C library, which should be installed when adding the package in Julia; it provides:

- **CVODES:** For integration and sensitivity analysis of ODEs.

CVODES treats stiff and non-stiff ODE systems such as  $y' = f(t,y,p)$ ,  $y(t_0) = y_0(p)$ , where  $p$  is a set of parameters.

- **IDAS:** For integration and sensitivity analysis of DAEs. IDAS treats DAE systems of the form  $F(t,y,y';p) = 0$ ,  $y(t_0) = y_0(p)$ ,  $y'(t_0) = y_0'(p)$ .

- **KINSOL:** For the solution of non-linear algebraic systems.

KINSOL treats fixed point non-linear systems; that is, of the form  $F(u) = 0$ .

In addition to the preceding heat equation, we will add an extra equation representing the depletion of fuel.

By redefining the time scales, it is possible to simplify the equations and write them in terms of temperature difference,  $x1$ , and reactant concentration,  $x2$ , as follows:

```
x1 = x2 * exp(x1) - a*x1
x2 = -b*x2 * exp(x1)
```

Note we have added an extra term to allow for the fact that fuel is limited, so the temperature will eventually return even after a catastrophic explosion:

```
julia> function exotherm(t, x, dx; a=1, b=1)
    p = x[2] * exp(x[1])
    dx[1] = p - a*x[1]
    dx[2] = -b*p
    return(dx)
end
```

The `exotherm` function models the twin equations with two parameters: `a` corresponding to the rate of cooling, and `b` corresponding to the depletion of fuel.

We will keep the value of `b` fixed at 0.1 but compute a set of six trajectories for `a` in [0.8,1.0,1.2,1.4,1.6,1.8].

The following is one of these solutions for `a` = 0.8:

```
julia> using Sundials
julia> t = collect(range(0.0; stop=5.0, length=1001));
julia> fexo(t,x,dx) = exotherm(t, x, dx, a=0.8, b=0.1);
julia> x1 = Sundials.cvode(fexo, [0.0,1.0],t)
1001×2 Matrix{Float64}:
 0.0      1.0
 0.0050013  0.999499
 0.0100053  0.998995
 0.015012   0.99849
 0.0200217  0.997982
 0.0250345  0.997471
  .....
  .....
  .....
 0.549299   0.000256713
 0.547083   0.000258691
julia> using Plots
julia> plot(x1[:,1])
```

*Figure 7.7* shows the complete set of six solutions; the one we computed previously is `y1`.

Notice that `y1`, `y2`, and `y3` all exhibit runaway conditions, whereas those of `y4`, `y5`, and `y6` do not. This is because the heat loss coefficient is insufficient to dissipate the heat being generated, and if it were not for fuel depletion, it would continue to rise without limit.

This is termed a bifurcation point; we see that this happens in the interval  $a \sim [1.2, 1.4]$  when  $b = 0.6$ .

As the value of  $b$  decreases, the bifurcation point shifts upward, requiring increasing values for  $a$  to balance the heat generation with that being dissipated. It is relatively easy to show that the limiting case for  $b = 0$  is  $a = \exp(1) \sim 2.71828\dots$ :

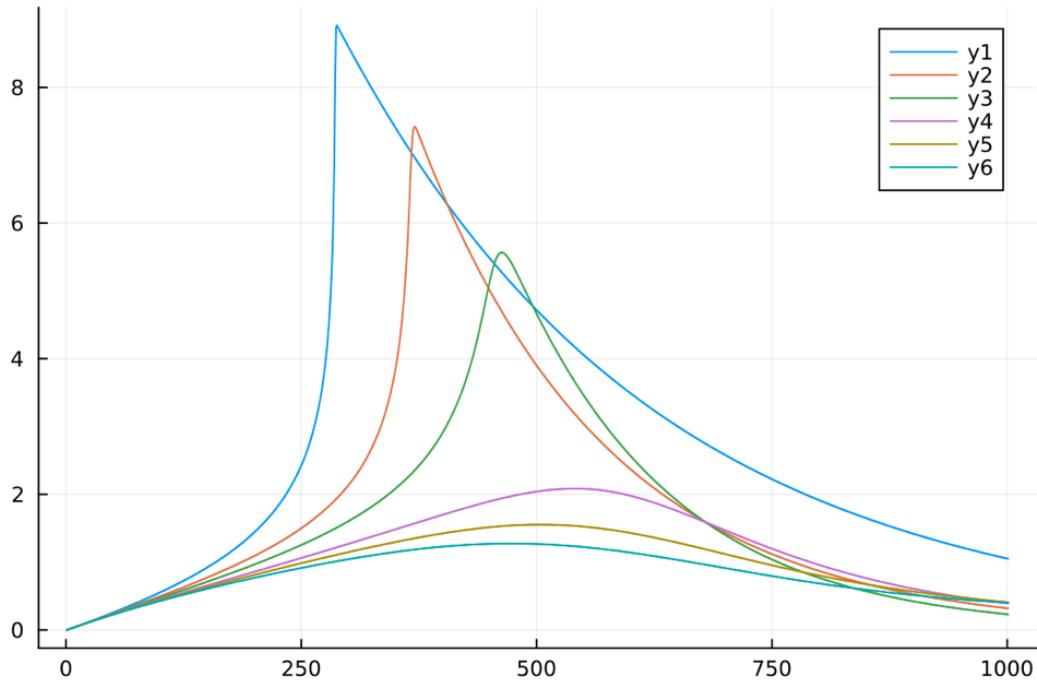


Figure 7.7 – Trajectories for exothermic reaction with fuel depletion

Catastrophe theory was somewhat eclipsed in the 1960s-70s, but they discovered that quite simple sets of DEs could display exotic solutions that are extremely sensitive to changes in parameters or initial conditions.

This was christened as chaos theory, and we will turn to this next.

## A touch of chaos

One example of a chaotic system is the infamous gravitational three-body equation that defeated Newton's and later mathematicians' attempts at a closed analytic solution.

A second is the double pendulum (rather than the simple one discussed previously). This comprises one pendulum with another pendulum attached to its end. It has a strong sensitivity to initial conditions, which in some instances can result in producing chaotic behaviors.

To illustrate the power of ODE solvers, we will conclude this section by modeling the strange attractor chaotic system first derived by Edward Lorenz, who coined the term “*the Butterfly effect*.” Lorenz was simulating a simple model of the weather with only three dependent variables and found that if he restarted the calculation at different points in the simulation, he got widely different long-time solutions. This was, in fact, because his outputted values, as printed, were not of the same precision as the actual ones currently in the simulation.

The following code defines the problem, then solves it, using CVODE\_Adams (from the Sundials package), and displays the solution via the Plots API to show a 3D visualization, all in 10 lines of code:

```
# Parameter(p = 28.0) is chosen so the solution is chaotic.
julia> function lorenz(du,u,p,t)
    du[1] = 10.0(u[2]-u[1])
    du[2] = u[1]*(28.0-u[3]) - u[2]
    du[3] = u[1]*u[2] - (8/3)*u[3]
end
julia> u0 = [1.0;0.0;0.0];
julia> tspan = (0.0,100.0);
julia> prob = ODEProblem(lorenz,u0,tspan);
julia> sol = solve(prob,CVODE_Adams());
julia> Plots.plot(sol,vars=(1,2,3))
```

Again, the Plots package is able to handle the solution *as-is*, this time as a 3D plot, as shown in Figure 7.8:

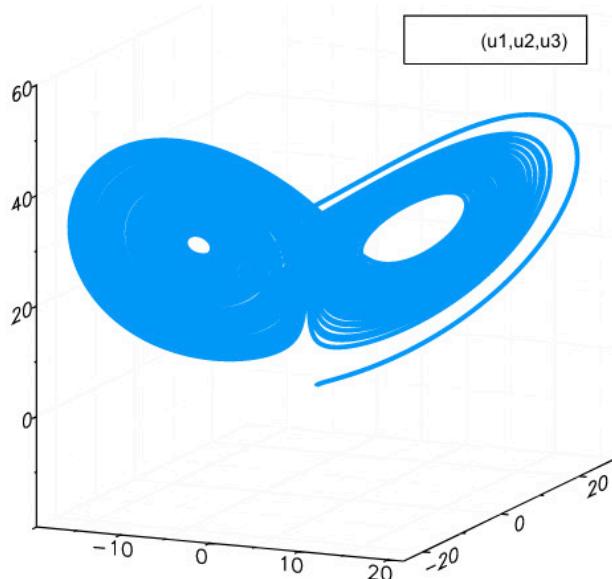


Figure 7.8 – The “famous” Lorenz strange attractor

## Stochastic DEs

The DE framework can be used to solve more than just ODEs.

The example chosen here is a **stochastic DE (SDE)** corresponding to the Wiener process, named after Norbert Wiener, who looked at the mathematical properties of one-dimensional Brownian motion. Brownian motion arises in many fields, including the behavior of particulates in fluids, quantum mechanics via the Feynman–Kac formulae, and the mathematical theory of finance, the Black–Scholes option pricing model.

Recall that we looked at an example of this when computing Asian stock options using Black–Scholes, in *Chapter 1*.

Here, we will discuss the application of the `DifferentialEquations` package to SDEs rather than just plain ODEs.

An SDE is a DE in which one or more of the terms is a random process, and so its variation is modeled by using a statistical distribution – often, but not always, Gaussian.

Consider the following pair of equations but with some random Gaussian noise applied; this is a Wiener process, and the solution can be found by running the code that follows the equations:

```
julia> using DifferentialEquations
julia> f(du,u,p,t) = (du .= u)
julia> g(du,u,p,t) = (du .= u)
# Create 8 initial values for u
julia> u0 = rand(8);
# The simplest Weiner process
julia> W = WienerProcess(0.0,0.0,0.0);
# Define the problem using W and solve the SDE

julia> prob = SDEProblem(f,g,u0,(0.0,1.0),noise=W)
SDEProblem with uType Vector{Float64} and tType Float64. In-place:
true
timespan: (0.0, 1.0)
u0: 8-element Vector{Float64}:
 0.14952217845370508
 0.44628615689631357
 0.7978404723313609
 0.018547049757968836
 0.22879411013724504
 0.5607109295584563
 0.5874916617204066
 0.3930951243934949

julia> sol = solve(prob,SRIW1());
# Plot the solutions using the Plots API and GR
```

```
julia> using Plots; gr()
julia> Plots.plot(sol, linewidth=2)
```

The results over the interval [0.0, 1.0] for 8 separate trajectories are shown in *Figure 7.9*:

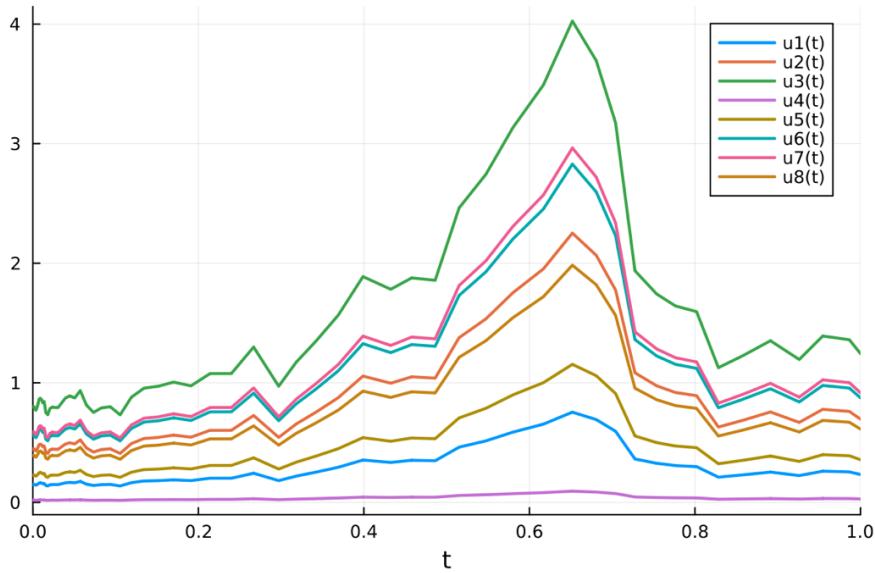


Figure 7.9 – Solutions of the simple Wiener process (Brownian motion)

The workbook also contains an example of modeling jump equations.

These are differential systems with sudden discontinuous breaks imposed to model sudden rises and falls (usually falls) in the stock market.

## Calculus

The **Calculus** package provides tools for working with the basic calculus operations of differentiation and integration. It can be used to produce approximate derivatives by several forms of finite differencing or to produce exact derivatives using **symbolic differentiation (SD)**.

### Differentiation

There are a few basic approaches to using the package, some of which we will examine in this section.

We use finite differencing to evaluate a derivative at a specific point and higher-order functions to create new functions that evaluate derivatives' SD to produce exact derivatives for simple functions:

```
julia> using Calculus
julia> f(x)=sin(x)*cos(x);
```

```
julia> derivative(f,1.0) -0.4161468365471423
# Check since d(f) => cos*cos - sin*sin
julia> cos(1.0)^2 - sin(1.0)^2 -0.4161468365471423
# Possible to curry the function
julia> df = derivative(f)
julia> df(1.0)
-0.4161468365471423
# Also defined is the 2nd derivative
julia> d2f = second_derivative(f)
julia> d2f(1.0)
-1.8185953905296441
```

`Calculus.jl` has some useful 2D functions; that is, that map  $\mathbb{R}^2 \Rightarrow \mathbb{R}$ :

```
#=
We need to be careful of name clashes by qualifying the functions.
=#
julia> h(x) = (1+x[1])*exp(x[1])*sin(x[2])*cos(x[2]);
julia> gd=Calculus.gradient(h);
julia> gd([1.0,1.0])
2-element Array{Float64,1}:
 3.7075900080760276 -2.262408767426671
julia> hs = Calculus.hessian(h);
julia> hs([1.0,1.0])
2x2 Array{Float64,2}:
 4.94345 - 3.39361
 -3.39361 - 9.88691
```

For scalar functions, we can use the '`'` operator to calculate derivatives as well:

```
julia> f'(1.0)
-0.41614683653632545
```

This operator can be used arbitrarily many times, but note that the approximation worsens with each higher-order derivative calculated:

```
julia> f''(1.0)
-1.8185953905296441
julia> f'''(1.0)
1.7473390557101791
julia> f''''(1.0)
5505.591834126032
```

It is possible to output the symbolic version of the derivative using the differentiated routine:

```
julia> differentiate("sin(x)*cos(x)", :x)
:(1 * cos(x)) * cos(x) + sin(x) * (1 * -(sin(x)))
```

Although not entirely perfect, this can be somewhat simplified:

```
julia> simplify(differentiate("sin(x)*cos(x)", :x))
:(cos(x) * cos(x) + sin(x) * -(sin(x)))
```

These techniques work with more than just a single variable:

```
julia> simplify(differentiate("x*exp(-x)*sin(y)", [:x, :y]))
2-element Array{Any,1}:
 :(1*exp(-x)*sin(y) + x*(-1*exp(-x)) * sin(y) + x*exp(-x)*0)
 :(0*exp(-x)*sin(y) + x*0*sin(y) + x*exp(-x)*(1*cos(y)))
```

Here, we get a 2D array corresponding to the partial derivatives, and clearly, terms with a zero multiplier can be ignored.

## Automatic differentiation

**Automatic differentiation (AD)** has become important recently as it is at the heart of many ML techniques, some of which, such as `Zygote.jl`, we will meet later in *Chapter 10*.

Interestingly, AD is different from symbolic methods such as those contained in `SymJulia` or numerous packages listed by the *JuliaDiff* group in <https://juliadiff.org>, and to illustrate this, I will consider an example especially to compute the square root of a real number:

```
julia> struct D <: Number
        d1::Float64
        d2::Float64
    end
julia> import Base: +, /, convert, promote_rule
julia> +(x::D, y::D) = D(x.d1+y.d1, x.d2+y.d2);

julia> /(x::D, y::D) =
        D(x.d1/y.d1, (y.d1*x.d2 - x.d1*y.d2)/y.d1^2);
julia> convert(::Type{D}, x::Real) = D(x, zero(x))
julia> promote_rule(::Type{D}, ::Type{<:Number}) = D
promote_rule (generic function with 128 methods)
```

Create an ordered pair with (d1=17, d2=1):

```
julia> da = D(17,1)
D(17.0, 1.0)
```

Use a method to compute roots by the algorithm by Hero of Alexandria (~ 60 C.E.), although it was probably known to Babylonian and Indian mathematicians much earlier than that:

```
julia> function hero(x; k::Integer = 10)
    @assert k > 0
    t = (1+x)/2
    for i = 2:k
        t = (t + x/t)/2
    end
    return t
end
```

Because of the magic of multiple displace, we can call the function with the D () structure since the algorithm only uses addition and division operations, both of which have been defined:

```
julia> db = hero(da)
D(4.123105625617661, 0.12126781251816648)
```

The first term (db . d1) contains the estimate of the square, and we can confirm that this is correct:

```
julia> db.d1^2
17.0
```

So, what about db . d2? It is equal to 0.5/sqrt(17), which is the first derivative of  $x^{0.5}$  for  $x \rightarrow 17$ .

How does this work? The Hero algorithm is actually a special case of Newton's method for solving general algebraic equations, 1,600 years before Newton!:

$$f(x) \sim x^2 - S = 0$$

$$x_{(n+1)} = x_n + \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - S}{2x_n} = \frac{1}{2} \left( x_n + \frac{S}{x_n} \right)$$

Let's decode it without using a "dual" number and return the results as a tuple:

```
julia> using Printf
julia> function dhero(x; k = 10)
    t = (1+x)/2
    dt = 1
    @printf "%3d : %.5f : %.5f\n" 1 t dt
    for i = 2:k;
        t = (t+x/t)/2;
        dt = (dt + (t - x*dt)/t^2)/2;
        @printf "%3d : %.5f : %.5f\n" i t dt
    end
    (t,dt)
end
```

```
julia> dhero(17, k=5)
1 : 9.00000 : 1.00000
2 : 5.44444 : 0.30508
3 : 4.28345 : 0.12793
4 : 4.12611 : 0.12127
5 : 4.12311 : 0.12127
(4.123106716962795, 0.12126781251943676)
julia> sqrt(17)
4.123105625617661
```

To see how this actually happens, display Hero's first four terms using the SymPy package:

```
julia> using SymPy
julia> x = symbols("x");
julia> for i = 1:4
    display(simplify(hero(x, k=i)))
end

$$\frac{x+1}{2}$$


$$\frac{4x + (x + 1)^2}{4(x + 1)}$$


$$\frac{x^4 + 28x^3 + 70x^2 + 28x + 1}{8(x^3 + 7x^2 + 7x + 1)}$$


$$\frac{x^8 + 120x^7 + 1820x^6 + 8008x^5 + 12870x^4 + 8008x^3 + 1820x^2 + 120x + 1}{16(x^7 + 35x^6 + 273x^5 + 715x^4 + 715x^3 + 273x^2 + 35x + 1)}$$

```

It gets a little complex after 4 with polynomials of the order O(18) – try upping it to 5 or more to see what I mean!

To evaluate these polynomials, we will define a simple function for the purpose:

```
julia> function poly(x,a)
    k = length(a)
    @assert k > 0
    if k == 1 return a end
    p = 0
    for i = 1:k
        p = p + a[i]*x^(i-1)
    end
    return p
end
```

The function takes a scalar number plus an array of coefficients in order from lowest (that is, *the constant term*) to highest.

Evaluate the previous symbolic output and check that this agrees with the results from the preceding `dhero()` function:

```
julia> poly(17, [0.5,0.5])
9.0
julia> poly(17, [0.25,1.5,0.25])/poly(17, [1,1])
5.444444444444445
julia> poly(17, [1,28,70,28,1])/(8*poly(17, [1,7,7,1]))
4.283446712018141
julia> poly(17, [1,120,1820,800,12870,8008,1820,120,1])/
       (16*poly(17, [1,35,273,715,715,273,35,1]))
4.124809325664573
```

The calculation seems to converge quickly, so let's just evaluate the first 4 iterations for values in the interval  $[0,20]$  and compare it with the analytic function.

The results are shown in *Figure 7.10*:

```
julia> using Plots; gr()
Plots.GRBackend()
julia> i = 0.0:0.1:20.0;
julia> plot([x->hero(x,k=i) for i=1:4],i,
           label=["Iteration $j" for i=1:1,j=1:4])
julia> plot!(sqrt, i, c="black", label="sqrt",
           title = "How to be a Hero")
```

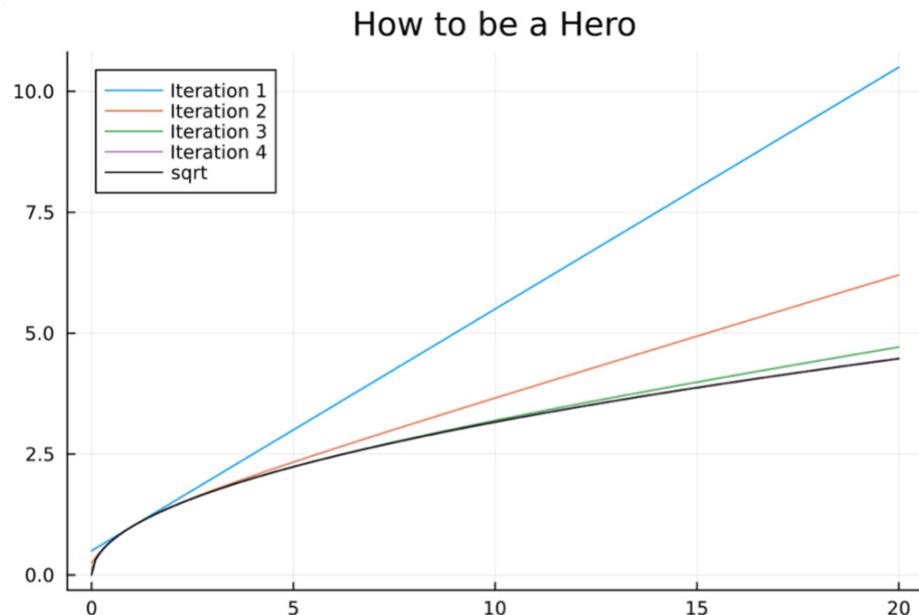


Figure 7.10 – Hero(ic) solutions for  $\sqrt{x}$  in interval  $[0,20]$

Try this with a `plotly()` backend to `Plots` and get an interactive graph in your browser.

The techniques discussed here form the basis for AD, which is a principal method in optimization problems and ML.

A good place to start is to look at forward difference methods using packages listed by *JuliaDiff*, such as <https://github.com/JuliaDiff/ForwardDiff.jl>.

## Quadratures

Integration (that is, quadratures) is no longer covered in `Calculus.jl`, and so in this section, I will briefly mention two separate packages: `QuadGK.jl` and `HCubature.jl`.

To illustrate the use of `QuadGK`, let us use a simple function,  $\sin(x) * (1.0 + \cos(x))$ , and integrate it over the interval  $[0.0, 1.0]$ :

```
julia> using QuadGK
julia> f(x) = sin(x) * (1.0 + cos(x))
julia> quadgk(f, 0.0, 1.0)
(0.813734403268646, 2.220446049250313e-16)
```

The function returns a tuple whose first component is the value of the quadrature.

An alternate package (`HCubature.jl`) has been implemented and written by Steven Johnson (of `PyCall`, `PyPlot`, `IJulia`, and so on), and this will compute multidimensional quadratures; however (clearly), it can also be used for 1D integration, providing the same result as `QuadGK`:

```
julia> using HCubature
julia> hquadrature(f, 0.0, 1.0)
(0.813734403268646, 2.220446049250313e-16)
```

The power comes when applying to a 2D (or higher) case and the initial conditions provided as vectors.

So, for the function  $2x^*e^{-x}*\sin(x)*\cos^*(x)$ , we can evaluate the quadrature as this:

```
julia> h(u) = 2.0*u[1]*exp(-u[1])*sin(u[2])*cos(u[2])
julia> hcubature(h, [0, 0], [1, 1])
(0.18710211142604422, 2.598018441709888e-9)
```

In the next section, I am going to turn to a brief discussion of Julia's optimization techniques using the `JuMP` package.

## Optimization

Mathematical optimization problems arise in the fields of linear programming, ML, resource allocation, production planning, and so on.

One well-known allocation problem is that of a traveling salesman who has to make a series of calls and wishes to compute the optimal route between calls. The problem is not tractable but clearly can be solved exhaustively; however, by clustering and tree pruning, the number of tests can be markedly reduced.

The generalized aim is to formulate the minimization of some  $f(x)$  function for all values of  $x$  over a certain interval, subject to a set of  $g_i(x)$  restrictions.

The problems of local maxima are also included by redefining the domain of  $x$ . It is possible to identify three cases:

1. No solution exists.
2. Only a single minimum (or maximum) exists.

In this case, the problem is said to be convex and is relatively insensitive to the choice of starting value for  $x$ .

3. The  $f(x)$  function has multiple extremals.

For this, the solution returned will depend particularly on the initial value of  $x$ .

Approaches to solving mathematical optimization may be purely algebraic or involve the use of derivatives. The former is typically exhaustive with some pruning, while the latter is quicker by utilizing hill-climbing-type algorithms.

Optimization is supported in Julia by a community group called *JuliaOpt*, and we will briefly introduce a couple of modules: JuMP and Optim.

## JuMP

As an example of using JuMP, we are going to maximize the function  $5x + 3y$  subject to the constraint  $3x+5y < 7$  using the Clp solver:

```
julia> using JuMP, Clp
julia> m = Model(Clp.Optimizer)
julia> @variable(m, 0 <= x <= 5 );
julia> @variable(m, 0 <= y <= 10 );
julia> @objective(m, Max, 5x + 3y );
julia> @constraint(m, 2x + 5y <= 7.0 );
```

We defined a model and associated it with a specific `Clp.Optimizer` optimization method routine.

For the optimizer selected, (in addition to JuMP), Clp must be imported.

The use of `@variable`, `@objective`, and `@constraint` macros define the starting domain ( $x,y$ ) for model  $m$  and the objective function and constraints in a clear manner.

Solving this is (now) a matter of calling the `optimize!()` routine in JuMP:

```
julia> JuMP.optimize!(m)
Coin0506I Presolve 1 (0) rows, 2 (0) columns
and 2 (0) elements
Clp0006I 0  Obj 0 Dual inf 15.5 (2)
Clp0006I 1  Obj 17.5
Clp0000I Optimal - objective value 17.5
Clp0032I Optimal objective 17.5 - 1 iterations time 0.002
```

The output is diagnostic from Clp but does contain the optimal solution.

This is also available in the model `m` (which `optimize!()` has modified), together with the values of the variable at the optimal solution:

```
julia> rdd(x,d=4) = round(x,digits=d)
julia> println("x = ", rdd(JuMP.value(x)), " and y = ", rdd(JuMP.
value(y)))
x = 3.5 and y = 0.0
julia> rdd(JuMP.objective_value(m))
17.5
```

### ***Knapsack problem***

This is a problem in combinatorial optimization; given a set of items, each with a weight and a price value, to determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit, and the total value is as large as possible.

It derives its name from the problem faced by someone who is constrained by carrying a knapsack to a maximum weight and must fill it with the most valuable items.

The knapsack problem has been studied for more than a century, with early works dating as far back as 1897; nowadays, it often arises in resource allocation where there are financial constraints. It is studied in fields such as combinatorics, computer science, complexity theory, cryptography, and applied mathematics.

We will solve the problem using JuMP, LinearAlgebra, and the GLPK solver with the following model:

```
julia> using JuMP, LinearAlgebra, GLPK
julia> N = 10;
julia> m = Model(GLPK.Optimizer)
# Prices vector of size N
julia> p = [92; 57; 68; 60; 43; 67; 84; 87; 72; 49];
# Weights vector of size N
julia> w = [23; 31; 44; 53; 38; 63; 85; 89; 82; 29];
# Weight constraint
```

```
julia> C = 170;
# Define array to hold the results
julia> @variable(m, x[1:N], Bin);
# Add the objective and the constraint(s)
julia> @objective(m, Max, dot(p, x));
julia> @constraint(m, dot(w, x) <= C);
```

There are 10 items with the prices and weights given in the arrays  $p$  and  $w$ .

The objective is to maximize prices by a choice of values in the  $x$  vector, subject to the constraint that any sack can only carry weights up to a total of  $C$  ( $\leq 170$ ).

Output the final state of the model  $m$ :

```
julia> m
A JuMP Model
Maximization problem with:
Variables: 10
Objective function type: AffExpr
`AffExpr`-in-`MathOptInterface.LessThan{Float64}`:
1 constraint
`VariableRef`-in-`MathOptInterface.ZeroOne`:
10 constraints
Model mode: AUTOMATIC
CachingOptimizer state: ATTACHED_OPTIMIZER
Solver name: GLPK
Names registered in the model: x
```

The solution takes a similar form to the one shown previously, except for the use of the GLPK optimizer:

```
julia> JuMP.optimize!(m)
julia> JuMP.objective_value(m)
309.0
```

The bin array holds which items will go into the knapsack, so we can use these to evaluate the total weight and profit:

```
julia> totalw = totalp = 0.0
julia> y = zeros(N)
julia> for i in 1:N
    y[i] = JuMP.value(x[i])
    totalw += y[i]*w[i]
    totalp += y[i]*p[i]
end
julia> y'
```

```
1x10 adjoint(::Vector{Float64}) with eltype Float64:
 1.0  1.0  1.0  0.0  1.0  0.0  0.0  0.0  0.0  1.0
julia> @show (totalw, totalp);
(totalw, totalp) = (165.0, 309.0)
```

We see that items 1, 2, 3, 5, and 10 will go into the bag with a total weight of 165, which is close to but below the constraint of 170.

The total price of the items is 309, agreeing with `objective_value()` as output earlier.

### ***The travelling salesman problem***

The classic example in this category is known as the **travelling salesman problem (TSP)** and like the aforementioned Knapsack problem has no known analytic solution, so must be solved by exhaustive optimization methods.

Basically, the question gives a list of towns and the distances between each pair of cities, starting at one specific city to derive the shortest possible route so that the salesman visits each town once and then returns to the city of origin.

Refer to [https://en.wikipedia.org/wiki/Travelling\\_salesman\\_problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem) on *Wikipedia* for a detailed discussion of the problem.

The solution to the TSP is much more complex than the knapsack problem, but there is an excellent solution in Julia by Daniel Schermer, to be found at [https://jump.dev/JuMP.jl/stable/tutorials/algorithms/tsp\\_lazy\\_constraints/](https://jump.dev/JuMP.jl/stable/tutorials/algorithms/tsp_lazy_constraints/).

An alternative approach in optimization problems is to use the `Optim` package, which is briefly described next.

### ***Optim***

`Optim` is a native package with which calculations are coded in Julia without the need for separate solvers or third-party libraries.

The main call is to the `optimize()` function, which requires at least a function definition and vectors for the starting values.

Optionally, a value for the solution method can be supplied with one of the following:

- `bfgs`
- `cg`
- `gradient_descent`
- `momentum_gradient_descent`
- `l_bfgs`

- nelder\_mead
- newton

As an example, consider the Rosenbrock function, defined as  $f(x, y) = (a-x)^2 + b^*(y-x^2)^2$ .

This is a non-convex function and is often used as a performance test problem for optimization algorithms. The global minimum is inside a long, narrow, parabolic-shaped flat valley to find the valley is trivial; however, converging to the global minimum is much more difficult.

The Rosenbrock function  $f(x, y) = (a-x)^2 + b^*(y-x^2)$  has a global minimum at  $(a, a^2)$ , where the function's value is  $f(a, a^2) = 0$

Usually, the  $(a,b)$  parameters are chosen as  $(1, 100)$ .

Define the Rosenbrock function as previously and solve the problem of looking for a local minimum by using Optim and starting at  $(0.0,0.0)$ , as follows:

```
# Use the BFGS method
# Hard code a = 1 and b = 100 into the function definition
julia> using Optim
julia> rosenbrock(x) = (1.0-x[1])^2 + 100.0*(x[2]-x[1]^2)^2
julia> result = Optim.optimize(rosenbrock, zeros(2), BFGS())
* Status: success
* Candidate solution
    Final objective value:      5.471433e-17
* Found with
    Algorithm:      BFGS
* Convergence measures
    |x - x'|          = 3.47e-07 ≤ 0.0e+00
    |x - x'| / |x'|   = 3.47e-07 ≤ 0.0e+00
    |f(x) - f(x')|   = 6.59e-14 ≤ 0.0e+00
    |f(x) - f(x')| / |f(x')| = 1.20e+03 ≤ 0.0e+00
    |g(x)|           = 2.33e-09 ≤ 1.0e-08
* Work counters
    Seconds run:    0  (vs limit Inf)
    Iterations:     16
    f(x) calls:     53
    ∇f(x) calls:   53
```

The BFGS method is chosen by default, although this can be overwritten.

We find that the algorithm did indeed converge in 16 iterations with a final objective value of  $O(10^{-17})$  and a tolerance of under  $10^{-8}$ .

### **Convex.jl**

There is one other optimization package worthy of mention; viz., `Convex.jl` (formerly `CVX.jl`), which can be found at <https://github.com/jump-dev/Convex.jl>.

At the time of writing, this is listed as in maintenance mode, so will not be reviewed here, but it does have an extensive set of examples at <https://jump.dev/Convex.jl/stable> for interested readers.

## **Stochastic simulations**

Problems encountered so far are completely determined by the models and will produce the same solutions repeatedly, even the chaotic strange attractor, given the same parametrization and initial conditions.

Some models have terms that occur randomly, and these are called stochastics.

We have already seen examples earlier of the price of a volatile stock. While the price increased roughly with the underlying price of money, fluctuations were considered to exist on a day-to-day process sampled from a Gaussian process. Time-series analysis is often used to reduce the effect of such fluctuations and reveal underlying trends, but there are certain systems where stochastics are paramount.

Typical examples are models of queueing systems that might occur in services at banks, or checkouts in supermarkets. I will discuss a particular case of a bank teller later in this section.

Simulations are often dealt with using a framework that attempts to hide the details of the coding as part of the model definition. This has similarities with the approach we saw with the `JuMP` package in optimization problems.

The approach is not a new one; in fact, it was introduced by IBM for simulation problems with **General Purpose Simulation System (GPSS)** in 1961, some of the modern-day inheritors being `jDisco` and `SimPy`.

## **SimJulia**

The `SimJulia` package is similar to the Python-based `SimPy` module, although as it is a native implementation, it requires neither a wrapper nor the use of `PyCall`. It is both a discrete event and a continuous time-process-simulation framework.

In discrete event models, time is advanced in steps, and individual events fire according to their schedules/triggers. For continuous events, the change is per step and computed according to derivative as well as variable values. Clearly, being able to handle discrete events is a prerequisite for stochastic simulations, but the ability to handle continuous events is a useful addition.

The classic example is that of a queue for service in a bank, post office, or grocery shop, and we will illustrate the use of `SimJulia` with such a model.

### **Bank teller example**

Consider an example of modeling a bank service with the following assumptions:

- Customers wait in a single queue arriving at random intervals
- There are a number of resources (tellers) who service the next customer in the queue
- Customers may decide to wait or leave depending on the length of the queue

All three may involve stochastic variates, and these will need to be generated from a probability distribution or will be based on actual empirical measurement.

The actual distributions of arrival rates will not necessarily be Gaussian since we cannot have negative waiting and service times. Arrivals may differ in a city situation where there is a peak around lunchtime and a “rush” toward closing.

Service times may also be long-tailed as some transactions may be more complex than the “norm” and require considerably more resources.

We will be interested in determining the mean time to serve customers and may be interested in the effects of increasing the number of tellers to match demand, but also balancing against the need to minimize the idle time of tellers.

The principle aim of such a simulation would be to decide whether it is desirable to increase the resource (tellers) to meet demand, with the obvious trade-off between the cost of extra tellers balanced against the extra revenue generated.

We will need the `Distributions` package to provide density functions to sample against, as we have noted that uniform or normal distributions are inappropriate.

We will assume in this example that the arrival times and service times follow an exponential (Poisson) distribution, although, in a real simulation, we noted that the modeling of these would need to be more exact. Also, we will assume if the queue length is more than a given maximum, the customer does not wait.

First, we need a function to represent the entire customer experience from arriving, through being served, to leaving, and output the relevant time steps and waiting times.

We will use a package called `ConcurrentSim` (formerly `SimJulia`), which is a discrete event process-oriented simulation framework, similar to the `SimPy` Python module and its successors of historic simulation approaches provided by Simula and GPSS:

```
julia> using ConcurrentSim, ResumableFunctions  
julia> using Printf, Random, Distributions
```

Define the model as follows:

```
julia> NUM_CUSTOMERS = 15 # number of customers to generate
julia> NUM_TELLERS = 2      # number of servers
julia> μ = 0.4             # service rate
julia> λ = 0.9              # arrival rate
julia> arrival_dist = Exponential(1/λ) # arrival times
julia> service_dist = Exponential(1/μ) # service times
```

Setting a different random number seed will generate different results per run.

For repeatable results, this could be a specific integer value rather than using the clock:

```
julia> iseed = ccall((:clock,"libc"),Int32,())
julia> Random.seed!(iseed);
```

We need to mode the visit, and this must be a resumable task as it may be suspended if no resource (that is, a bank teller) is available to service it immediately.

The following function contains a number of `@printf` statements for exemplary purposes; these can be omitted or else triggered with the addition of a Boolean `DEBUG` constant:

The points of task suspension/resumption are indicated by using the `@yield` macro:

```
julia> queue_length = 0;
julia> queue_stack = Array{Integer,1}(undef,0);
julia> @resumable function visit(env::Environment,
                                teller::Resource,
                                id::Integer,
                                time_arrvl::Float64,
                                dist_serve::Distribution)
# customer arrives
# queue_length is a global variable
    global queue_length
    @yield timeout(env, time_arrvl)
    @printf
    "Customer %2d %10s : %.3f\n" id "arrives" now(env)
    if queue_length > 0
        push!(queue_stack,id)
        println(
            "CHECK: Length of the queue is $queue_length")
    end
    queue_length += 1
# customer starts to be served
    @yield request(teller)
    queue_length -= 1
```

```

    @printf
    "Customer %2d %10s : %.3f\n" id "being served" now(env)
# teller is busy
    @yield timeout(env, rand(dist_serve))
# customer leaves
    @yield release(teller)
    @printf
    "Customer %2d %10s : %.3f\n" id "leaves" now(env)
end

```

Armed with this functional model, we just need to initialize a simulation and run it:

```

# initialize simulation <: environment
julia> sim = Simulation()
#initialize service resources
julia> service = Resource(sim, NUM_TELLERS)
# initialize customers and set arrival time
# customers arrive randomly based on Poisson distribution
julia> arrival_time = 0.0
julia> for i = 1:NUM_CUSTOMERS
    arrival_time += rand(arrival_dist)
    @process visit(sim,service,i,arrival_time,service_dist)
end

```

Note that the visits are scheduled by another macro in SimJulia: `@process`.

We now run the simulation:

```

julia> run(sim)
Customer 1 arrives : 2.115
Customer 1 being served : 2.115
Customer 1 leaves : 2.463
Customer 2 arrives : 2.909
Customer 2 being served : 2.909
Customer 3 arrives : 3.199
Customer 3 being served : 3.199
Customer 4 arrives : 3.590
Customer 2 leaves : 4.269
Customer 4 being served : 4.269
Customer 4 leaves : 4.441
Customer 5 arrives : 4.478
Customer 5 being served : 4.478
Customer 6 arrives : 5.357
Customer 7 arrives : 5.553
CHECK: Length of the queue is 1

```

```
Customer 8 arrives : 5.854
CHECK: Length of the queue is 2
Customer 5 leaves : 5.924
Customer 6 being served : 5.924
Customer 3 leaves : 6.320
Customer 7 being served : 6.320
Customer 9 arrives : 6.491
CHECK: Length of the queue is 1
Customer 10 arrives : 6.504
CHECK: Length of the queue is 2
Customer 11 arrives : 6.563
CHECK: Length of the queue is 3
Customer 12 arrives : 6.595
CHECK: Length of the queue is 4
Customer 6 leaves : 7.576
Customer 8 being served : 7.576
Customer 8 leaves : 8.486
Customer 9 being served : 8.486
Customer 13 arrives : 8.928
CHECK: Length of the queue is 3
Customer 14 arrives : 8.980
CHECK: Length of the queue is 4
Customer 15 arrives : 9.114
CHECK: Length of the queue is 5
Customer 7 leaves : 11.223
Customer 10 being served : 11.223
Customer 9 leaves : 11.576
Customer 11 being served : 11.576
Customer 11 leaves : 12.604
Customer 12 being served : 12.604
Customer 12 leaves : 13.776
Customer 13 being served : 13.776
Customer 13 leaves : 14.617
Customer 14 being served : 14.617
Customer 10 leaves : 16.256
Customer 15 being served : 16.256
Customer 14 leaves : 17.441
Customer 15 leaves : 20.032
```

Notice in this simulation that for the parametrization chosen, the queue length gets as high as 5, and so a third service checkout (NUM\_TELLERS = 3) would seem to be required to ensure that customers do not choose to wait.

In practice, the model should incorporate some means to allow for situations in which customers choose to leave without being served.

## Summary

This chapter covered a diverse range of topics arising from the discipline of scientific computing, with a certain amount of cherry-picking on my part. Julia is now especially blessed with several packages that can be applied to scientific problems, so the reader is encouraged to look at the various community groups for additional information. We began by looking at classical linear algebra problems, the solutions of which are provided by routines from within the Julia BASE and STDLIB systems.

For the remaining sections, we turned to a variety of packages and applied them to examples from signal processing, optimization, and the solution of ordinary and stochastic DEs and touched upon the support Julia provides for the differentiation and integration of functions.

Finally, we looked at the solution of problems such as those that have a random (stochastic) component and saw how this can be simulated by various packages within Julia.

In the next chapter, we will have some light relief and look in greater detail at the question of the production of graphics and other forms of data visualization and will see that Julia's various approaches are especially rich and diverse.



# 8

## Visualization

Julia has no built-in graphics commands. This means that it is not possible to create some datasets and issue a plot command without first installing and loading a package.

One reason for this is that Julia needs to build a variety of different **operating systems (OSs)** from source; any libraries that are shipped, such as OpenBLAS and LibUV, must be in source form and not interfere with the building process.

Graphics engines have a variety of different backends, such as Gtk, Qt, and others. While specialist packages may be restricted in terms of their OS support, the overall Julia system may not.

Initially, the inclusion of built-in graphics was seen as a long-term goal and one that would be added in future releases; however, it seems that this is not the case anymore. As we will see, Julia is much better for this.

With the introduction of the Plots API, which we have seen frequently in the previous chapters, it is possible to use a uniform syntax for a variety of backends (PyPlot, GR, PlotlyJS, and others), all of which here.

The topics in this chapter are presented under five main headings:

- Visualization using text packages
- Basic graphics (“golden oldies”)
- The Plots API
- Display frameworks
- Image processing and raster graphics

There are a few packages that do not conform to the Plots API, and we will look at them as well; this includes the Winston and Gadfly packages.

An additional point to note is that the Julia method of importing symbols into its main namespace via the `using` command means that most graphics packages, which tend to have functions such as `plot()` and `display()`, do not produce a name clash.

Of course, it is possible to use `import` and fully qualify any function call. This is an approach I will adopt in this chapter to avoid such clashing. Normally, when you are working with a single package, a simple `using` statement can be employed.

Earlier in this book, we covered some of the important modules that are used for creating graphics. In this chapter, we will cover some of these again, as well as introduce a few more of the most widely used. All the packages discussed in this chapter are now very sophisticated, so the examples provided only skim the surface of their capabilities. You are encouraged to review all the accompanying documentation for any individual package.

## Textual visualization

Displays that use text refer to packages that do not need a graphics engine (such as QT). The graphics are generated in one of two ways:

- Using normal printable characters, either ASCII and/or Unicode
- By writing textual markup, normally based on LaTeX, though via Postscript or even directly to a PDF is possible

We learned how to implement the first of these methods previously, but it will be useful to briefly mention it again here.

### Simple inline displays

When we provided an overview of Julia in *Chapter 1*, we briefly saw that it is possible to create some quite sophisticated text graphics (that is, using printable characters) using the `UnicodePlots` package. It is the successor of early packages such as `ASCIIPLOTS` and `TextPlots` and provides a wider variety of available graphic types to display.

We saw examples of a line plot and a histogram in *Chapter 1*. As a quick refresher, I'll create a horizontal bar plot of the populations of the inner London Boroughs according to the 2021 census.

The data is contained in the `inner-london-boros.txt` file as comma-separated pairs of values. This could be processed using a CSV file or `DataFrames` package but as an alternative, we will just read the file line by line and populate a couple of vectors on the fly:

```
import UnicodePlots
const ui = UnicodePlots
X = Vector{String}();
Y = Vector{Int64}();
DS = ENV["HOME"]*"/MJ2";
for s in eachline("inner-london-boros.txt")
```

```
(x,y) = split(s, ", ")
push!(X,x)
push!(Y, parse(Int64,y))
end
julia> ui.barplot(X,Y,
    xlabel="Inner London Boroughs Population")
```

The resulting output is shown in *Figure 8.1*:

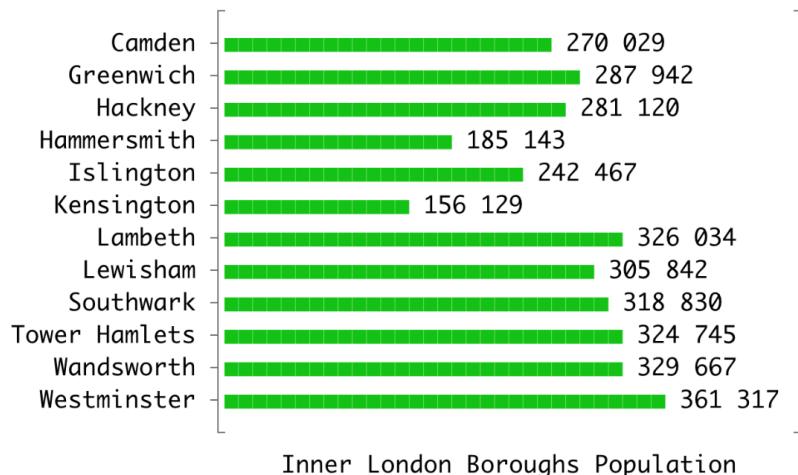


Figure 8.1 – Bar plot of the population of inner London boroughs (2021 census)

As its name suggests (`UnicodePlots`), it can use Unicode characters in addition to simple ASCII ones.

It can create a wider variety of graphs, some of which are as follows:

- Scatter plot
- Line plot
- Staircase plot
- Bar plot (horizontal)
- Histogram (horizontal)
- Box plot (horizontal)
- Sparsity pattern density plot

For a more detailed discussion of these graphs, plus a few extra ones, please refer to the package documentation (<https://github.com/JuliaPlots/UnicodePlots.jl>).

## Luxor

Most graphic packages are raster style, where pixels are set (*colored*) within a canvas. We will look at some examples of these in the final section of this chapter.

However, Luxor is an alternative for drawing static vector graphics. The advantage of vector-based graphics is that they can easily be rescaled by just expanding (or reducing) the dimensions of the viewpoint.

It is a high-level interface to the `Cairo.jl` module (*hence the name*) and provides more simple drawing functions for working with shapes, polygons, clipping masks, PNG images, turtle graphics, animations, and shapefiles.

Additionally, once a drawing is completed, it can easily be saved in PDF, PNG, SVG, or EPS files.

The package contains a few macros to test the installation, such as `@svg` and `@png`:

```
julia> using Luxor
julia> @png juliacircles()
```

As a more exciting example, let's use Luxor to display a drawing based on Sierpinski triangles.

This is a fractal shape that was described by Waclaw Sierpinski in 1915 and is a self-similar structure that occurs at different levels of iterations, or magnifications.

It is constructed from an equilateral triangle by repeatedly removing triangular subsets using the following prescription:

1. Start with an equilateral triangle.
2. Subdivide it into four smaller congruent equilateral triangles and remove the central triangle.
3. Repeat *Step 2* with each of the remaining smaller triangles forever.

The result of the applying algorithm five-fold is shown in *Figure 8.2*:



Figure 8.2 – Sierpinski triangles generated from the preceding algorithm

To develop the code in Julia, we must use/import the packages we need. Note the output; when the code is executed, it will be in color. This is true for a variety of the figures in this chapter:

```
julia> using Random, Colors  
julia> import Luxor  
julia> const lx = Luxor
```

Again, we are importing the graphics package and aliasing it as `lx` to avoid any possible name clashes later in this chapter.

Now, let's define a function to create a triangle and use this to build up the Sierpinski triangle:

```
julia> function triangle(points, degree)  
    lx.sethue(cols[degree])  
    lx.poly(points, :fill)  
end  
julia> function sierpinski(points, degree)  
    triangle(points, degree)  
    if degree > 1  
        p1, p2, p3 = points  
        sierpinski([p1, lx.midpoint(p1, p2),  
                   lx.midpoint(p1, p3)], degree-1)  
        sierpinski([p2, lx.midpoint(p1, p2),  
                   lx.midpoint(p2, p3)], degree-1)  
        sierpinski([p3, lx.midpoint(p3, p2),  
                   lx.midpoint(p1, p3)], degree-1)  
    end  
end
```

Now, we can draw the graphic and preview the result:

```
julia> function draw(n)  
    lx.circle(lx.O, 100, :clip)  
    points = lx.ngon(lx.O, 150, 3, -pi/2, vertices=true)  
    sierpinski(points, n)  
end  
julia> lx.Drawing(400, 250)  
julia> lx.background("white")  
julia> lx.origin()  
julia> depth = 8  
julia> cols = distinguishable_colors(depth)  
julia> draw(depth)  
julia> lx.finish()  
julia> lx.preview()
```

The output is shown in *Figure 8.3*:

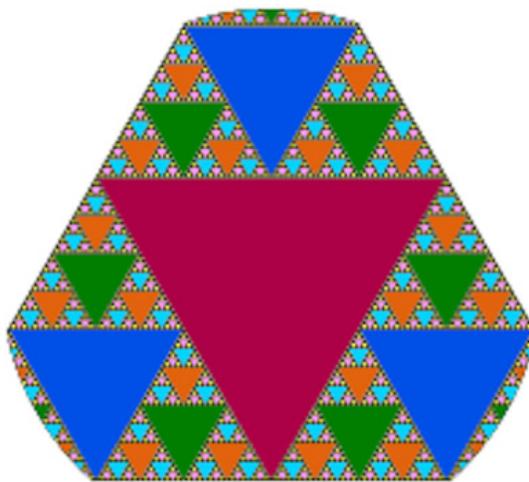


Figure 8.3 – Sierpinski triangles as generated by the Luxor package

The triangles at each level have been given a different color and the outer bounding triangle is clipped.

Luxor incorporates some simple “turtle graphics” functions. We will look at these next.

## Turtle graphics

The routines to control the *turtle* all begin with a capital letter: `Forward`, `Turn`, `Circle`, `Orientation`, `Rectangle`, `Pendown`, `Penup`, `Pencolor`, `Penwidth`, `Reposition`, and so on. The angles are specified in degrees.

Here is an example of using these commands to create a simple drawing:

```
julia> import Luxor
julia> lx = Luxor

julia> lx.Drawing(600, 400, "turtles.png")
julia> lx.origin();
julia> lx.background("midnightblue");

julia> tur = lx.Turtle();
julia> lx.Pencolor(tur, "cyan");
julia> lx.Penwidth(tur, 1.5);
julia> n = 5;
julia> for i in 1:400
```

```
global n
lx.Forward(tur, n)
lx.Turn(tur, 89.5)
lx.HueShift(tur)
n += 0.75
end
julia> lx.fontsize(20)
julia> lx.finish()
```

The result is created on the disk as the rainbow spiral by altering the hue of the “pen” each time the turtle is turned. The output is written to the `turtles.png` file, a copy of which is shown in *Figure 8.4*. It would be a sin not to see it in its full glory:

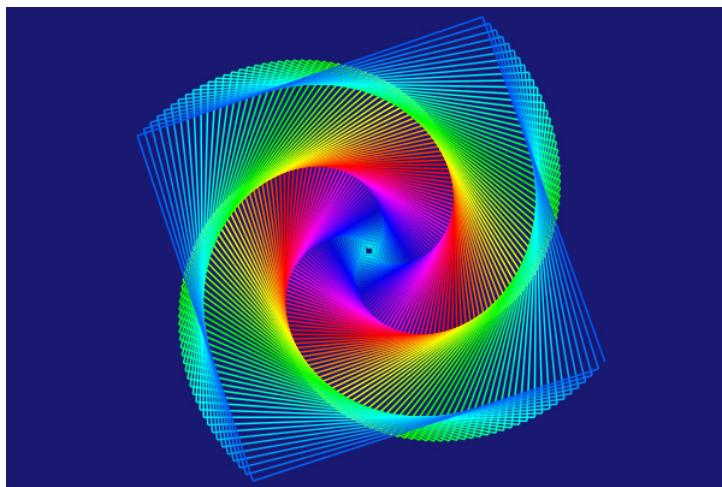


Figure 8.4 – Rainbow spiral created by the “turtle” commands in Luxor

The final thing we will consider here is PGFPlots, which uses LaTeX to produce displays.

## PGFPlots

The PGFPlots package is maintained by JuliaText (<https://github.com/JuliaTeX>). The group has a few other interesting packages, especially `BibTeX.jl` for those with an interest in bibliographic citations.

PGFPlots integrates well with IJulia to output SVG images to the notebook.

The user supplies axis labels, legend entries, and the plot coordinates for one or more plots, while the package applies axis scaling, computes any logarithms, and supplies axis ticks before drawing the plots.

The TEX library supports line, scatter, bar, area, histogram, mesh, and surface plots, but at the time of writing, not all of these have been implemented in the Julia package.

As with all graphic engine type packages, certain additional executables need to be present for `PGFPlots.jl` to work, as follows:

- `Pdf2svg`: This is required by `TikzPictures`. Its installation varies by OS.
- The `Pgfplots` library: This is installed using a LaTeX package manager such as TeX Live or MiKTeX.

The latter has a source forge web page (<https://pgfplots.sourceforge.net>) that acts as an excellent reference regarding what can be achieved. It also provides some additional links.

The following code demonstrates drawing some of the curves we covered in earlier chapters:

```
julia> using PGFPlots
julia> p = Axis([
    Plots.Linear(x -> sin.(3x).*exp(-0.3x), (0, 8),
        legendentry = L"\sin(3x)*\exp(-0.3x)" ),
    Plots.Linear(x -> sqrt.(x) ./ (1+x.^2), (0, 8),
        legendentry = L"\sqrt{2x}/(1+x^2)" ) ])
```

Saving this to an SVG file requires us to install the `pdf2svg` utility:

```
julia> save("fig-08-05.svg", p);
```

This figure can be displayed in a web browser, as shown in *Figure 8.5*:

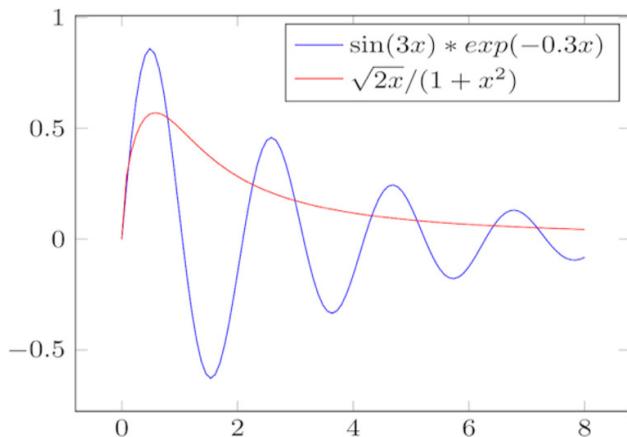


Figure 8.5 – Simple curves generated by PGFPlots

It is very easy to make histograms with another type of the `Plots.Histogram` style:

```
julia> fq = randn(10000);
julia> p = Axis(Plots.Histogram(fq, bins=100), ymin=0)
julia> save("histogram.svg", p);
```

I have included a copy in the `Files` folder of MJ2/DataSources.

**Note**

Here, `Plots` is a submodule of PGFPlots and should not be confused with the `Plots` API, which is the subject of a later section.

It is possible to use the `tikzCode()` routine to show the LaTeX code generated by the preceding code:

```
julia> print(tikzCode(p)
\begin{axis}[
    ymin = {0}
]
\addplot+[
    mark = {none},
    ybar interval, fill=blue!10, draw = blue
] coordinates {
    (-3.456651394038948, 2.0)
    (-3.3326094791467336, 3.0)
    (-3.2085675642545195, 7.0)
    . . . . .
    . . . . .
    . . . . .
    (3.985863499493914, 1.0)
    (4.109905414386128, 1.0)
};
\end{axis}
```

Leaving text packages behind, we will turn to some packages that were quickly written to fill the need for graphics support in the early versions of Julia. These are what I describe as “golden oldies” but are still around and flourishing today.

## Basic graphic packages

These packages are, confusingly, supported by two Julia groups, namely `JuliaGraphics` (<https://github.com/JuliaGraphics>), and `JuliaPlots` (<https://github.com/JuliaPlots>), so these are both great sources of reference.

First, we will shift to more familiar territory by discussing the graphics implemented through Python modules. We will cover ones we have met on various occasions throughout this book already – that is, `PyPlot` and its cousin, `PythonPlot`.

## PyPlot and PythonPlot

PyPlot is a part of the work of Steven Johnson of MIT, which arose from the previous development of the PyCall module. We have used it extensively in the previous chapters and will take a little time to discuss it further here.

Note that PyPlot is one of the graphics packages that can be used as a backend for the Plots API, with the others being GR and PlotlyJS. We will cover these later in this chapter.

PyPlot provides an interface to the `matplotlib` plotting library from Python; therefore, to use it, Python and `matplotlib` must be installed. If this is successful, it will work either by creating an independent window (via Python) or embedding it in an IJulia workbook.

I found that the easiest way to install both Python and `matplotlib` is using the Anaconda distribution from continuum.io.

This works on all three common platforms – that is, Windows, Mac OS X, and Linux.

For a full discussion, and any problems relating to the installation, please refer to GitHub's online documentation of `PyPlot.jl`.

The latter package, PythonPlot, again from Professor Johnson, uses a more recent module called `PythonCall.jl` to interface with Python's `matplotlib` with much less overhead than the older `PyCall.jl`. It is based on a fork of the PyPlot package and aims to function in the main as a drop-in replacement for the original.

However, there are a few caveats that need to be observed when switching from PyPlot to PythonPlot. Some of the major points are listed in the module's README file, but one worth noting is that the `show`, `close`, `step`, and `fill` functions are renamed to `plotshow`, `plotclose`, `plotstep`, and `plotfill`, respectively.

For other differences, refer to <https://github.com/JuliaPy/PythonPlot.jl>.

For our purposes, I will continue to use the older PyPlot module.

As a first example, I have picked one from the earliest pieces of PyPlot documentation – that of a sinusoidally modulated sinusoid since it produces a beautiful graphic.

The following code creates the code, displays it via a native Python window (from the REPL or VS Code) or inline graphic in a Jupyter notebook, and then writes the disk as an SVG file:

```
julia> import PyPlot
julia> const py = PyPlot
julia> x = collect(range(0.0, stop=2pi, length=1000))
julia> y = sin.(3*x + 4*cos.(2*x));
julia> py.title("A sinusoidally modulated sinusoid");
julia> py.plot(x, y, color="red", linewidth=2.0,
```

```
linestyle="--") ;  
julia> py.savefig("sinusoid.svg");
```

The resulting plot is shown in *Figure 8.6*:

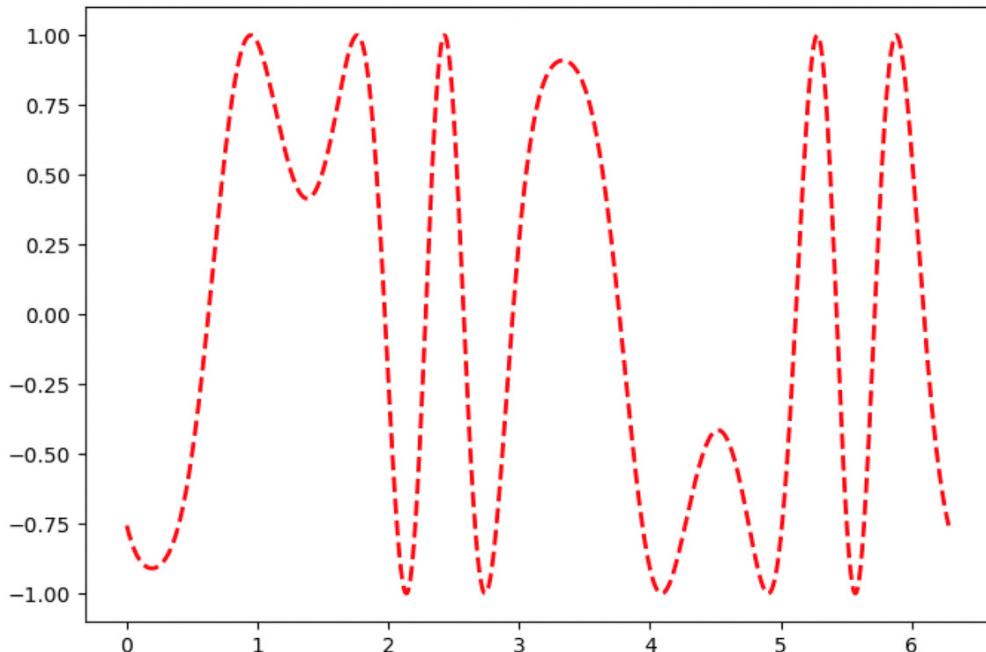


Figure 8.6 – A sinusoidally modulated sinusoid

The PyPlot package also imports functions from Matplotlib's mplot3d toolkit.

Unlike `matplotlib`, you can create 3D plots directly without creating an `Axes3d` first object, simply by calling `bar3D`, `contour3D`, `contourf3D`, `plot3D`, `scatter3D`, and so on.

Furthermore, PyPlot exports *MATLAB-like* synonyms such as `surf` for `plot_surface` and `mesh` for `plot_wireframe`.

The following code shows how to create a really simple 3D surface:

```
julia> y = collect(range(0, stop=3π, length=250))  
julia> py.surf(y, y, y.*sin.(y).*cos.(y).*exp.(-0.4y))
```

Here's the output:

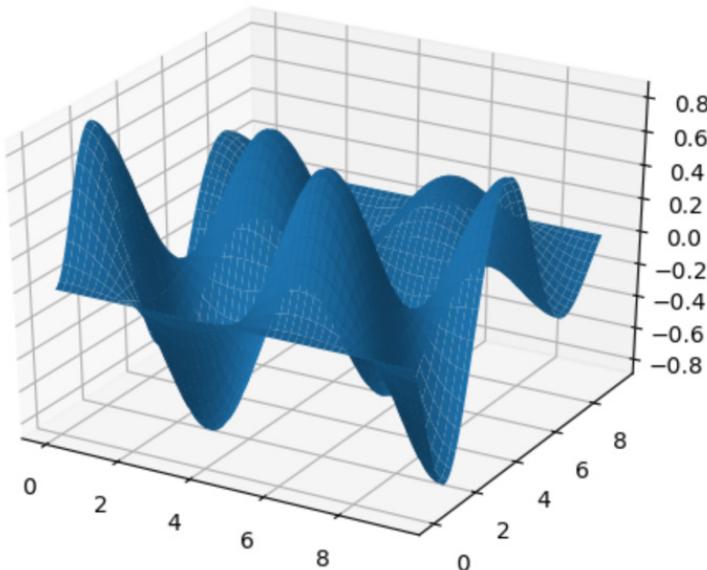


Figure 8.7 – Visual display of a 3D surface

As a final example, let's create a more substantial display with axes, titles, and annotations using the XKCD comic mode from Python's Matplotlib.

The module includes an `xkcd()` routine that, when called, switches seamlessly to XKCD mode, at which point all aspects of the display are affected:

```
julia> py.xkcd()
julia> x = collect(range(1, length=101, stop=10));
julia> y = sin.(3x + cos.(5x))
julia> py.title("XKCD fun")
julia> py.plot(x,y)
```

The `comic` graph is shown in *Figure 8.8*:

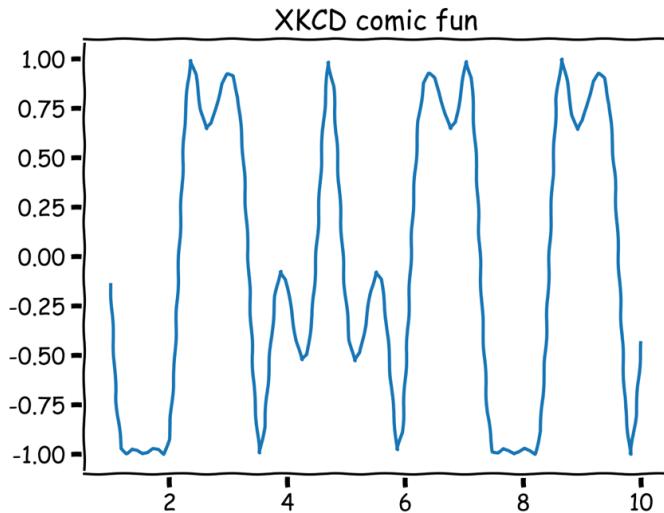


Figure 8.8 – Graph of  $\sin(3x) + \cos(5x)$  in XKCD mode

Since PyPlot and PythonPlot are wrappers around Python’s Matplotlib, the number of routines, parameters, and other aspects are very extensive.

The Julia documentation suggests that as a reference source, you should use [https://matplotlib.org/2.0.2/api/pyplot\\_api.html](https://matplotlib.org/2.0.2/api/pyplot_api.html).

## Winston

Winston is a 2D plotting package that has been available since the earliest days of Julia. It has fallen out of favor in recent years and does not conform to the Plots API, which we will cover later in this chapter.

One of the reasons for this is the difficulty of installing backend support on certain platforms. Nevertheless, it is compliant with version 1.0 and is a particular favorite of mine, so I won’t apologize for discussing it here.

The typical usage we have already seen is implemented via the `plot()` function:

```
julia> import Winston;
julia> const wn = Winston
julia> t = collect(range(0, stop=3pi, length=1000));
```

Let’s define three functions and create arrays based on the `t` variate and display them:

```
julia> f(x::Array) = 8x .* exp.(-0.3x) .* sin.(3x);
julia> g(x::Array) = 0.1x.* (2pi .- x).* (3pi .- x);
```

```
julia> h(x::Array) = 10.0 ./ (1 .+ x.*x).^0.5;
julia> y1 = f(t); y2 = g(t); y3 = h(t);
julia> wn.plot(t,y1,»b»,t,y2,»r»,t,y3,»k»)
```

Here's the output:

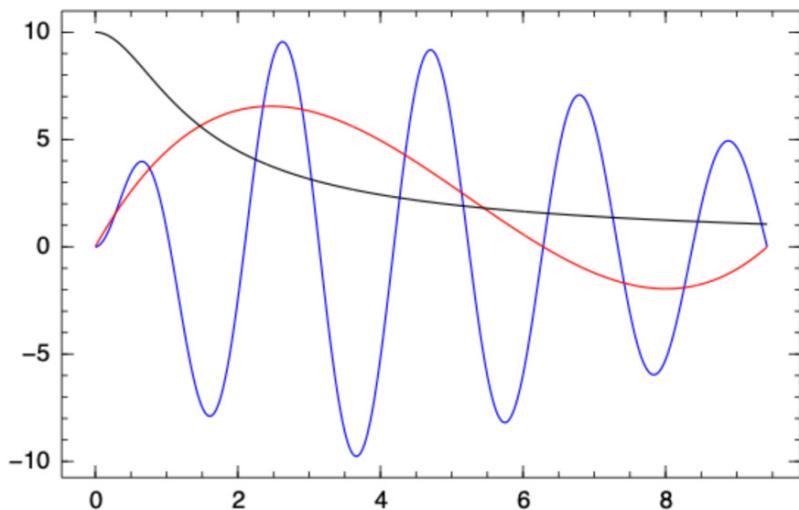


Figure 8.9 – Plots of three curves generated by Winston

We can also use Winston to create log-log and semi-log plots.

As an example, we will use the first curve from the preceding set of three functions and plot it against the logarithm of time.:

```
# Plot y1 against log(t)
julia> wn.semilogx(t,y1)
julia> wn.title("log(t) vs 10x * exp(-0.3x) * sin(3x)")
```

**Note:**

The text style uses  $\backslash\pi$  to plot  $\pi$ .

Here's the output:

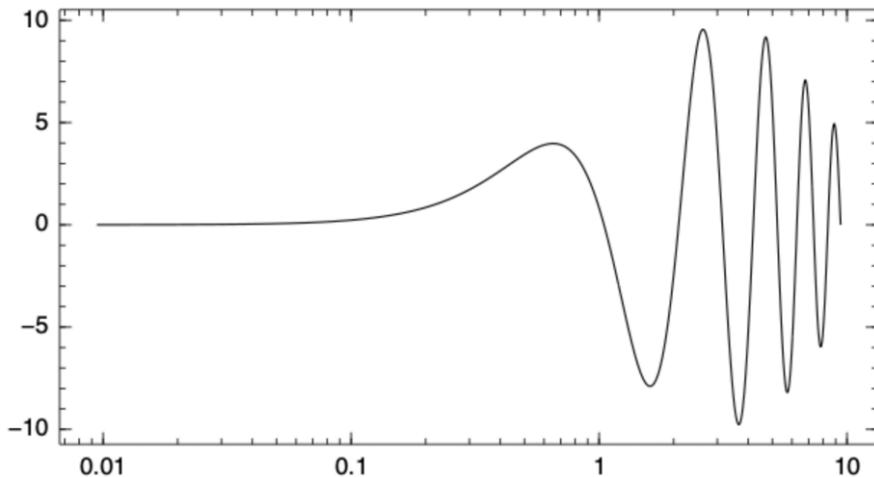


Figure 8.10 – Plot of  $10x \cdot \exp(-0.3x) \cdot \sin(3x)$  against  $\log(t)$

To conclude this brief introduction to Winston, let's look at a more complex example using a framed plot. Such a plot allows various parameters of the graphic, as well as curves, titles, and so on, to be set before they're displayed.

Let's create a framed plot and a linear relationship between two variables, dithering each by applying a random Gaussian variate:

```
julia> p = wn.FramedPlot(aspect_ratio=1,
           xrange=(-10,110), yrange=(-10,110));
julia> n = 21;
julia> x = collect(range(0.0, length=n, stop=100.0));
julia> yA = 10.0*randn(n) .+ 40.0;
julia> yB = x .+ 5.0*randn(n);
```

Now, let's set labels and symbol styles for the plot:

```
julia> a = wn.Points(x, yA, kind="circle");
julia> wn.setattr(a,label="'a' points");
julia> b = wn.Points(x, yB);
julia> wn.setattr(b,label="'b' points");
julia> wn.style(b, kind="filled circle");
```

At this point, we can plot a line that “fits” through the yB points and add a legend at the top LHS part of the graph:

```
julia> s = wn.Slope(1, (0,0), kind="dotted");
julia> wn setattr(s, label="slope");
julia> lg = wn.Legend(.1, .9, Any[a,b,s] );
julia> wn.add(p, s, a, b, lg);
```

Finally, we can display the completed graphic:

```
julia> wn.display(p)
```

Here's the output:

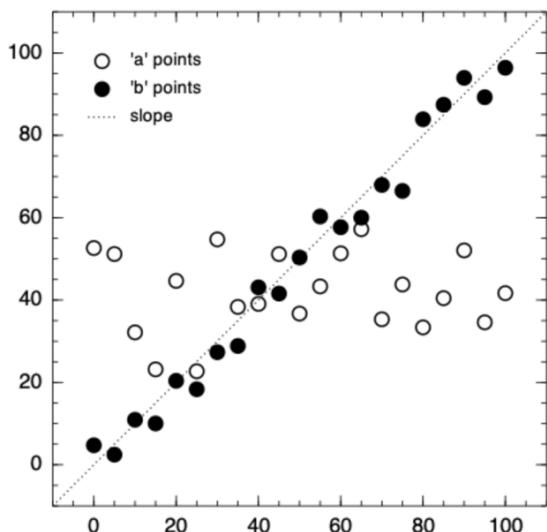


Figure 8.11 – Graphic using a Winston framed plot

This plot can be saved to disk as a PNG file using `wn.savefig(p, "WF-Plot.png")`.

## Gadfly

Gadfly is a large and complex package and provides great flexibility regarding the range and breadth of the visualizations that are possible in Julia. It is equivalent to the ggplot2 R module and is based on *The Grammar of Graphics*, a seminal work by Leland Wilkinson.

Together with Winston and PyPlot, Gadfly is one of the earliest visualization packages that has stood the longevity test.

The package, which was originally implemented in Julia by Daniel C. Jones, is a heavyweight that uses a large set of modules. It takes a long time to compile and create displays but this is compensated by the variety of visualizations that can be produced. Gadfly has a website (<http://gadflyjl.org/stable/>) that demonstrates a great many of these.

Once created, Gadfly can render the graphics to publication quality, outputting them in SVG, PNG, Postscript, and PDF format.

The following code creates a scatter diagram of random (x,y) pairs of numbers displayed in the positive quadrant between [0.0:1.0, 0:0:1.0] but filters so that only points in the positive arc are displayed. You may recall that this was a method we used to estimate a value for  $\pi$  in *Chapter 1* since for the unit arc, the area will be equivalent to  $\pi/4$ :

```
julia> using Gadfly, Cairo, Fontconfig
julia> gd = Gadfly;
julia> X = Float64[]; Y = Float64[];
julia> N = 1000;
julia> for i in 1:N
        x = rand()
        y = rand()
        if (x*x + y*y) < 1
            push!(X,x)
            push!(Y,y)
        end
    end
julia> dd = gd.plot(x = X, y = Y)
julia> draw.PNG("random-pts.png", 15cm, 12cm) , dd
```

Note that Gadfly works with Jupyter (via IJulia) to display the output in a notebook rather than in a separate window.

To save the graphic as a PNG file, we need to include Cairo and Fontconfig.

The result is shown in *Figure 8.12*.

The number of points that are selected is the size of the  $X$  and  $Y$  arrays and can be found using the `length()` method or `lastindex()` as an alternative:

```
julia> K = lastindex(X); # => 782
julia> println("Estimate of π is ", round(4*K/N, digits=3))
Estimate of π is 3.144
```

Here's the output:

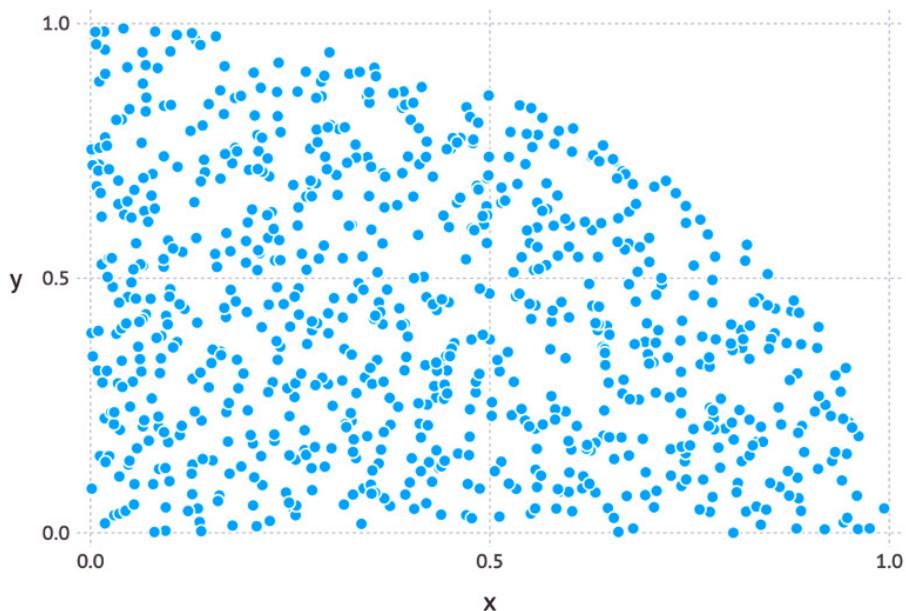


Figure 8.12 – Points in the upper quadrant of the unit circle that contribute to the estimate of Pi

Gadfly has a tight integration with Julia's DataFrames, which is especially useful.

To illustrate this, let's look at the GCSE result set we investigated in *Chapter 6, Working with Data*.

Recall that this is available as part of the RDatasets suite of source data:

```
julia> using Gadfly, RDatasets, DataFrames;
julia> mlmf = dataset("mlmRev", "Gcsemv")
julia> df = mlmf[completescases(mlmf), : ]
1523×5 DataFrame
Row | School  Student  Gender  Written  Course
    | Cat...   Cat...   Cat...   Float64?  Float64?
1   | 20920    27      F        39.0     76.8
2   | 20920    31      F        36.0     87.9
3   | 20920    42      M        16.0     44.4
4   | 20920    101     F        49.0     89.8
. . . . .
. . . . .
```

After extracting the data, we need to operate with values that do not have any missing values. We can use the `completescases()` routine to create a subset of the original data.

To view the data values for the exam and coursework results and also differentiate between boys and girls, we can do the following:

```
julia> plot(df, x="Course", y="Written", color="Gender")
```

Note that Gadfly produces the legend for the gender categorization automatically:



Figure 8.13 – Gender categorization

### ***Plotting functions***

As an example of a function-type invocation, the following code shows what can be produced in a single call – Perl and APL programmers would be proud of it!

```
#=
Take note of the extensive use of Julia's broadcasting style.
=#
julia> plot((x,y) ->
    x .* exp.(-(x - floor.(x))).^2 .- y.^2,
           -8.0, 8.0, -2.0, 2.0)
```

The result is displayed in *Figure 8.14*:

$$x \cdot \exp(-(x - \lfloor x \rfloor)^2) - y^2$$

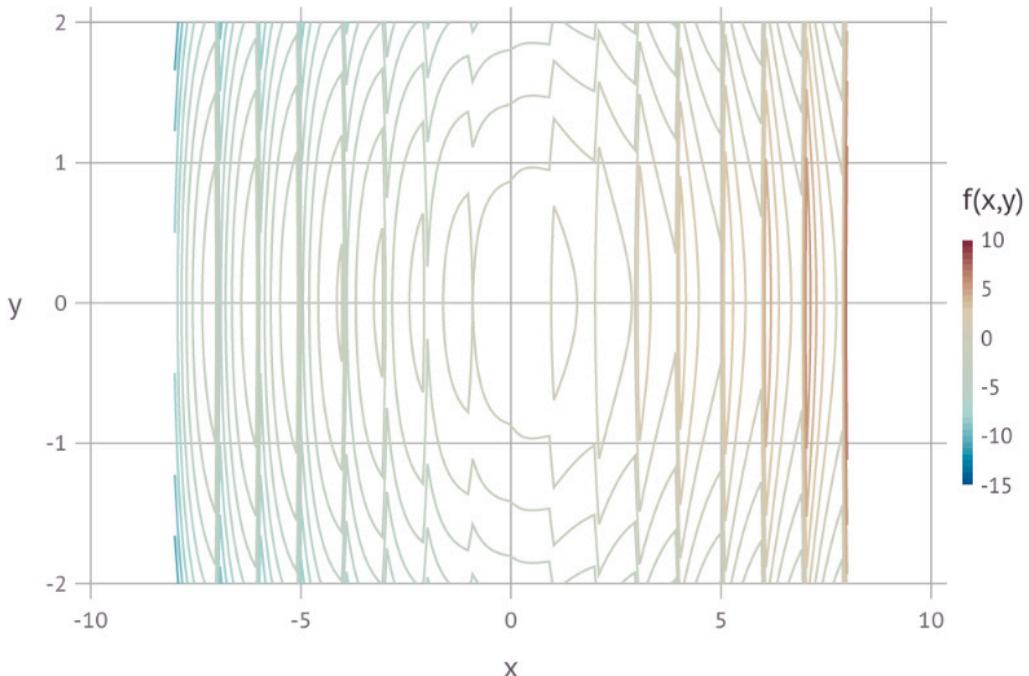


Figure 8.14 – 2D plot of  $x \cdot \exp(-(x - \lfloor x \rfloor)^2) - y^2$

Finally, looking at a different type of invocation, we can use the following function:

```
f(x) -> x*(5-x)*sin(5x) with x ∈ [0:5]
```

This is shown as a line graph in *Figure 8.15*. It's superimposing the data points, which are changed slightly due to the use of a randomly distributed normal variate.

Gadfly produces multiline plots using the `layer()` routine and uses the concept of themes to overwrite the color schemes.

Let's consider the line plot of the preceding function uniform variate (*in red*) together with a scatter plot of the dithered points.

The colors can be set using the `gd.Theme()` routine:

```
julia> x = collect(0.0:0.02:5.0);
julia> n = length(x); # => Will be 251 points
julia> y1 = [3*y*(5-y)*sin(5*y) for y in x];
```

```
julia> y2 = Vector{Float64}(undef,n);
julia> [y2[i] = y1[i] + randn() for i in 1:n];
julia> import Gadfly
julia> const gd = Gadfly
julia> gd.plot(
    gd.layer(x=x,y=y1,gd.Geom.line,
              gd.Theme(default_color=gd.colorant"red")),
    gd.layer(x=x, y=y2, gd.Geom.point,
              gd.Theme(default_color=gd.colorant"blue")))
```

The plot can be seen in the following figure:

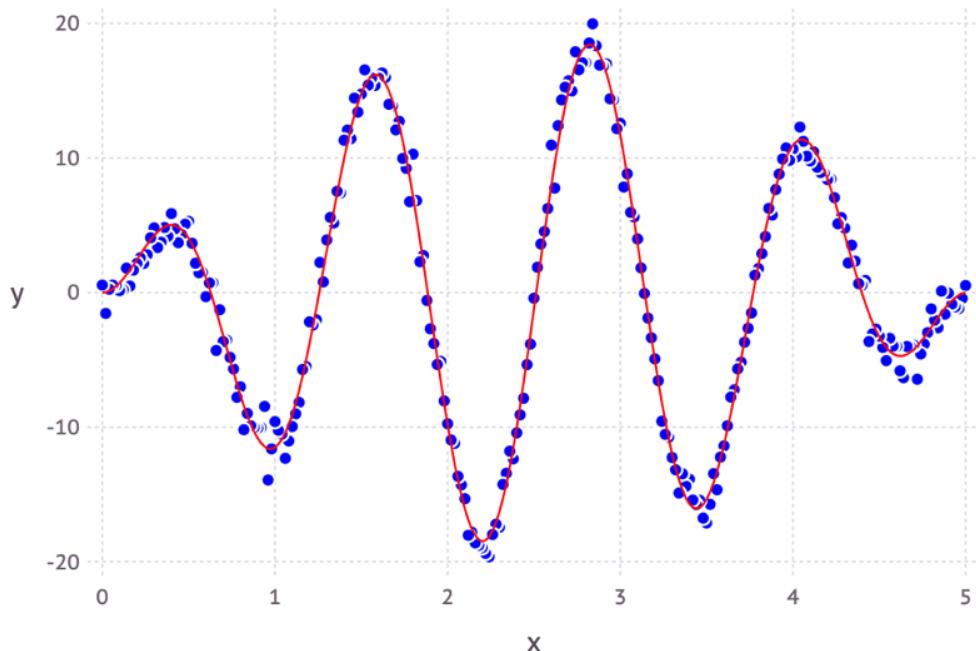


Figure 8.15 – Plot of  $x \cdot (5-x) \cdot \sin(5x)$  with slightly “jittered” points

In the next section, we will look at the `Compose` package. This was designed as a vector graphics system that functions as a principal component of `Gadfly`. However, as we will see, it can be used in its own right to create some complex patterns.

### **Compose**

Unlike most vector graphics libraries, `Compose` is thoroughly declarative. The graphics are defined by using a tree structure, assembling various primitives, and then letting the module decide how to draw them.

The primitives can be classified as context, form, and property; the assembly operation is then achieved via the `compose()` function:

- `context`: An internal node
- `form`: A leaf node that defines some geometry, such as a line or a polygon
- `property`: A leaf node that modifies how its parent's subtree is drawn, such as a fill color, font family, or line width
- `compose(a, b)`: This returns a new tree rooted at `a`, with `b` attached as a child

A typical invocation has a distinctly LISP-like feel. We can use the following code to build a complex drawing based on the Sierpinski fractal; we covered this previously as an example of using Luxor. Here, the overall shape of an equilateral triangle is subdivided *recursively* into smaller equilateral triangles:

```
julia> using Compose
julia> function sierpinski(n)
    if n == 0
        compose(context(), polygon([(1,1), (0,1), (1/2, 0)]));
    else t = sierpinski(n - 1);
        compose( context(), (context( 1/4, 0, 1/2, 1/2), t),
                  (context( 0, 1/2, 1/2, 1/2), t),
                  (context( 1/2, 1/2, 1/2, 1/2), t));
    end
end
```

The triangle is composed using the `polygon()` routine and built up recursively:

```
julia> ctxt = compose(sierpinski(1),
    linewidth(0.2mm), fill(nothing), stroke("black"));
julia> draw(SVG("sierp1.svg", 10cm, 8.66cm), ctxt);
julia> ctxt = compose(sierpinski(3),
    linewidth(0.2mm), fill(nothing), stroke("black"));
julia> draw(SVG("sierp3.svg", 10cm, 8.66cm), ctxt);
julia> ctxt = compose(sierpinski(5),
    linewidth(0.2mm), fill(nothing), stroke("black"));
julia> draw(SVG("sierp5.svg", 10cm, 8.66cm), ctxt);
```

---

The following figure shows a montage of three separate invocations for  $n = 1, 3, 5$ :



Figure 8.16 – Sierpinski fractal built up using the Compose package

### **Gaston**

Gaston is another early Julia plotting package that provides an interface to `gnuplot`, which is a separate visualization system available on Linux, Mac OS X, and, with a little work, Windows too.

Gaston has not had any major changes recently but due to the nature of the package, it doesn't need them – it has been tested by me with Julia LTS (1.6) and is also stable (1.9.x) on Linux and Mac OS X; it should work on Windows as well.

We won't discuss it in detail in this chapter, but for GNU enthusiasts, details of the package can be found at <https://mbaz.github.io/Gaston.jl/stable>.

A related package called `Gnuplot.jl` is referenced in the Gaston documentation; as its name suggests, it also provides another interface to `gnuplot`.

## **The Plots API**

`Plots.jl` is a visualization interface and toolset. It was the brain-child of Tom Breloff and is maintained by several outstanding contributors comprising the JuliaPlots community group.

We have covered the Plots package on several occasions in the preceding chapters, so this section will discuss some of its more fundamental philosophy and features.

The API sits above other backends, such as GR and PyPlot, connecting commands with implementation. If one backend does not support the desired features or make the right trade-offs, it is possible to switch to another backend with one command.

There is no need to change the code and no need to learn a new syntax.

Another backend is Spencer Lyon's PlotlyJS. This is essentially an “offline” version of the older Plotly module. I will cover how to deal with both of these later in this chapter.

Some of the initial aims of the package were listed by Breloff as follows:

- **Powerful:** Complex visualizations are easy to create
- **Intuitive:** Commands “just work”
- **Concise:** More efficient development and analysis
- **Flexible:** Produces plots from your favorite package
- **Consistent:** No need to commit to one graphics package
- **Lightweight:** Very few dependencies

In addition to the GitHub sources, they provide an extensive set of online documentation with many examples of dynamic, interactive, and 3D visualizations.

In this section, I will just concentrate on some of the simpler main features of the API.

Part of the power of Plots lies in the many combinations of allowed input data. For example, writing `plot(x = 1:10, y = rand(10))` will work as expected as it will simply translate into a call of `plot(1:10, rand(10))`.

Alternatively, it is possible to use `plot(rand(10))`. Here, the single input will be mapped to the `:y` keyword, and a missing value for `:x` will default to a unit range of 1:10.

Also, passing a  $(n \times m)$  matrix of values will create  $m$  series, each with  $n$  data points while following a consistent rule – vectors apply to a series, while matrices apply to many series.

This is demonstrated in the following example:

```
julia> using Plots

# Use GR as backend
julia> gr()
Plots.GRBackend()
# There will be 25 data points in 3 separate series
julia> xs = 0 : 2π/25 : 2π;
# Define a sine, cosine and a scaled function(by x)
julia> data = [sin.(xs) cos.(xs)
               0.5.*xs.*sin.(xs).*cos.(xs)];
julia> size(data)
(25,3)
# We put labels in a vector: one applies to each series
julia> labels = ["Sine" "Cosine" "Scaled function"];
# Marker shapes are to be applied to the data points
julia> markershapes = [:diamond :circle :star5];
#=
Marker colors in a matrix: applies to series and data points
```

```
=#
julia> markercolors = [:orange :red :blue];
# Now create the display
julia> p = plot(xs, data, label = labels,
               shape = markershapes,
               markercolor = markercolors,
               markersize = 5)
```

The backend that's used here is GR. When issued from the REPL, it creates a separate window using (*on my Mac*) the GKS QTterm.

Notice the effect of specifying the labels, marker shapes, and color attributes:

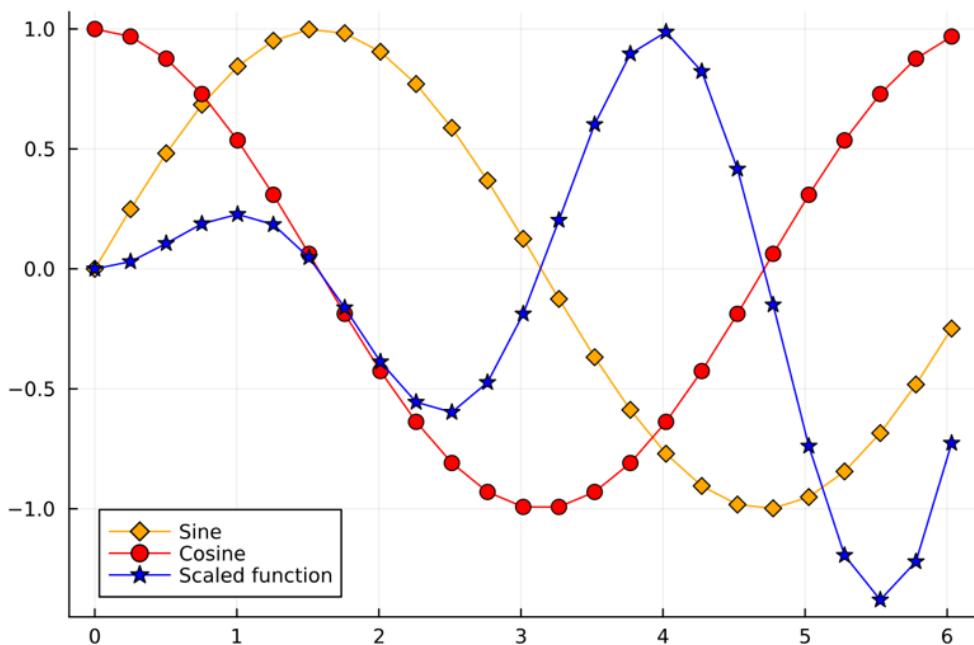


Figure 8.17 – Graph of  $\sin(x)$ ,  $\cos(x)$  and  $0.5 \cdot x \cdot \cos(x) \cdot \sin(x)$  using the Plots API

## Creating multiple plots using layouts

There are many methods for doing this; the following code highlights a simple method, which is to define a layout that will split a series. The layout command takes in a 2-tuple that builds a grid of plots and will automatically split a series for each plot:

```
julia> yy; = 2.0 * randn(100,3)
julia> plot(yy, layout = grid(3, 1, heights=[0.3,0.3,0.3]))
```

Here's the output:

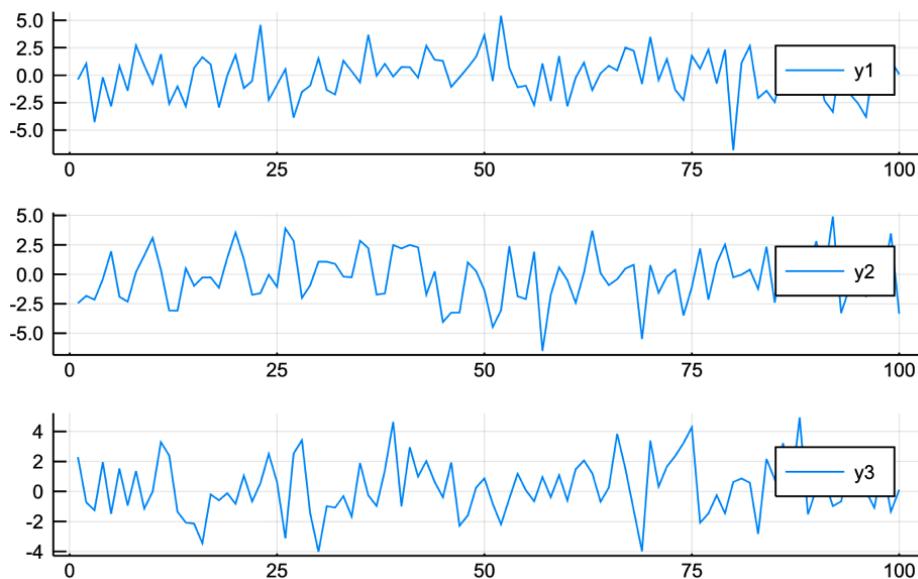


Figure 8.18 – 3x1 vertical grid of random number plots

`yy` is a matrix of three plots shown on a grid of three plots and are labeled as `y1`, `y2`, and `y3`.

A full discussion of many of the advanced features of layouts is provided in the online documentation (<https://docs.juliaplots.org/latest/layouts>).

## Recipes

Recipes are a way of defining visualizations in your packages and code, without having to depend on Plots. They are a way of defining visualizations by utilizing the `@recipe` macro from `RecipesBase`.

`RecipesBase` is a package that allows users to create advanced plotting logic without Plots.

Recipes have given rise to several frameworks. One such framework is StatsPlot. It will be discussed in the next section; some others will be covered in the following section.

There are four main types of recipes in Plots:

- User recipes
- Type recipes
- Plot recipes
- Series recipes

**Note**

The recipe type is determined by the dispatch signature.

The following is an example of a simple recipe:

```
julia> mutable struct MyRecipe end
julia> @recipe
function f(::MyRecipe, n::Integer = 10; add_marker = false)
    linecolor --> :blue
    seriestype := :path
    markershape --> (add_marker ? :circle : :none)
    delete!(plotattributes, :add_marker)
    rand(n)
end
```

`MyRecipe` is an empty mutable structure that is used for the dispatch signature.

The recipe aims to create a random set of points.

Here's an explanation of how the `@recipe` macro works:

1. The `f(args...; kw...)` signature is converted by `@recipe` into a definition of `apply_recipe(plotattributes::KW, args...)`, where `plotattributes` is an attribute dictionary of the type alias of `KW Dict{Symbol, Any}`.
2. The `-->` operator turns the `→ :blue` line color into `get!(plotattributes, :linecolor, :blue)`, setting the attribute only when it doesn't already exist.
3. The `:=` operator turns the `:= :path` series type into `plotattributes[:seriestype] = :path`, forcing that attribute value.
4. `markershape` checks the `add_marker` custom keyword but only if `markershape` was not already set.
5. The macro then returns the data to be plotted via a call to `rand()`.

We need to instantiate the empty structure and define each of the four separate plots:

```
julia> mt = MyRecipe();

julia> plot(plot(mt, 25, linecolor = :black),
           plot(mt, 100, linecolor = :red),
           plot(mt, marker = (:star5,5)),
           plot(mt, add_marker = true) )
```

All these are line plots of a set of random numbers.

The effect of using these recipes is shown in *Figure 8.19*:

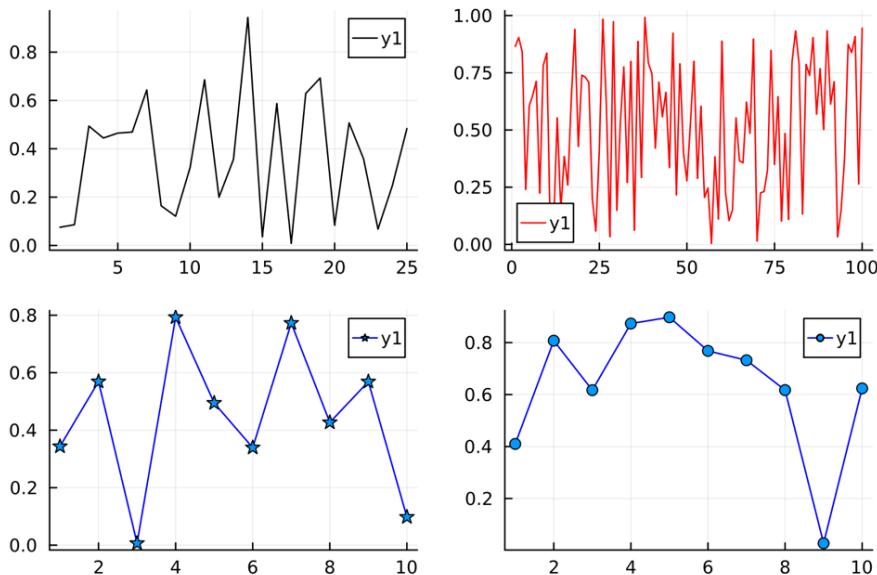


Figure 8.19 – 2x2 grid of plots using “MyRecipe”

Plot 1 has 25 points while plot 2 has 100 points, whereas plots 3 and 4 default to 10 points in the recipe. The difference between the latter two is the marker symbol and that a different set of random numbers are generated.

After “using” the Plot API, it is necessary to follow this with an additional call to a “so-called” backend. We will discuss this briefly next.

## Backends

Although not all graphic packages obey the Plots API, because of the liberal reorganization of parameters, they are quite extensive.

These are termed backends and the three of the most used are as follows:

- GR
- PlotlyJS
- PythonPlot

While working with Plots, different backends provide slight variations in functionality, so some parts of the API may not be available.

## GR

Although a `using` statement for the backend should *not* be used for a backend, the package needs to be installed before one is used.

One of the best choices is GR, which is very quick and works on all platforms. To be used on Mac OS X, GKSTerm must be installed.

Like all backends, GR can be used as a standalone graphics package. We won't discuss it here, so you are encouraged to look at the extensive online documentation to see what can be achieved.

## PlotlyJS

These are treated as separate backends, though they share much of the code and use the Plotly JavaScript API.

`plotly()` is the only dependency-free plotting option as the required JavaScript is bundled with Plots. It can create inline plots in IJulia or open standalone browser windows when run from the Julia REPL.

However, `plotlyjs()` is seen as the preferred option and taps into the greater functionality of Spencer Lyon's `PlotlyJS.jl` – for example, inline IJulia plots can be updated from any cell.

From the Julia REPL (*rather than a notebook*), `plotlyjs()` uses `Blink.jl` and Electron to plot within a standalone GUI window. It also supports more output formats than Plotly, such as such as Postscript and PDFs.

We will look at PlotlyJs and Plotly in a little more detail in the next section on frameworks.

## PythonPlot

The PythonPlot package should be familiar to us by now as it has been used for many of the examples in this book and integrates well with Jupyter notebooks. It also conforms to the Plots API and is a good choice for a backend.

### **Note that PyPlot is deprecated in v1.9.x.**

PythonPlot has a great wealth of functionality that's inherited from Python's `matplotlib`; this is well supported by the API. We have seen that it can create 2D and 3D displays and will work with the REPL as well as Jupyter and VS Code.

The downside is that Python needs to be installed (including Matplotlib) but it has been remarked that this is likely to be the case if you're using Jupyter and that a distribution such as Continuum's Anaconda setup should handle everything necessary between Julia and Python.

There can be some setup problems with Python support but these are discussed in the Julia documentation.

The three backends we've listed here are the most popular ones but there are a few other alternatives.

We looked at two of these when we described some standalone packages – that is, UnicodePlots and PGFPlots(X).

Which backend you choose will depend on which attribute is important:

- **Features:** PythonPlot, Plotly(JS), GR
- **Speed:** GR, InspectDR
- **Interactivity:** Plotly(JS), PyPlot, InspectDR
- **Design:** Plotly(JS), PGFPlots/PGFPlotsX
- **REPL plots:** UnicodePlots
- **3D plots:** PythonPlot, GR, Plotly(JS)

One of the backends on this list that you will likely be unfamiliar with is InspectDR.

Some of Julia's plotting options may be too slow, especially when you're dealing with large sets of data, typically of the order of 100K or more. This makes their use in simulation and/or responsive cases somewhat problematic. For that reason, InspectDR has been designed to be fast, even in such situations.

Imaging is built upon the Cairo library, using Julia/GTK+ for the display and higher-level functions such as generating and manipulating “widgets.”

The details on the web are a little scant but boast an impressive set of features. A few of the major ones are listed here:

- Publication-quality output
- Included as a “backend” of `Plots.jl`
- Relatively short load times/time to plot
- Designed to handle larger datasets
- Responsive even with moderate (say >200k points) datasets

Likewise, the documentation and examples are much less extensive than we would get with visualization frameworks such as Makie, StatsPlots, and the Images(.jl) suite, which we will meet later.

However, GitHub does provide a set of “demo” samples, together with a few more detailed examples that can be found via the following two references:

- <https://github.com/ma-laforge/InspectDR.jl/tree/master/sample>
- <https://github.com/ma-laforge/InspectDR.jl/tree/master/Blink>

Furthermore, since it can be used as a backend to the Plots API, a good place to start might be to switch out GR, or PlotlyJS, in favor of InspectDR in some of the Plots API examples by calling `inspectdr()`.

Examples of using InspectDR haven't been included here but one is available in the accompanying code for this chapter, both as a REPL/VS Code script and also a Jupyter notebook.

### Saving plots with HDF5

In *Chapter 7*, we saw that **Hierarchical Data Format (HDF)** is a set of file formats (HDF4, HDF5) that are designed to store and organize large amounts of data and that Julia has support for the latter; it is also used as a special case for its internal JLD data format.

Using HDF5 as a backend is unusual since it does *not* create any graphics; rather, it can be used to save, and later retrieve, a visualization:

```
julia> using Plots;
julia> hdf5()
Plots.HDF5Backend()
```

Create a plot, `p`, using the preceding simple Plots example. This needs to be done *after* the backend is specified.

This can be saved to disk in HDF5 format. Note that the user is (currently) issued a warning, which can be ignored.

Let's create a display, as we did previously, but with only the first couple of plots by using the data from the recipe (`mt`) in the previous example:

```
julia> p = plot(plot(mt, 25, linecolor = :black),
                  plot(mt, 100, linecolor = :red));
    | Warning: HDF5 interface does not support `display()` function.
    | Use `Plots.hdf5plot_write(::String)` method to write to .HDF5 "plot"
file instead.
└ @ Plots ~/julia/packages/Plots/UQI78/src/backends/hdf5.jl:193
julia> Plots.hdf5plot_write(p, "plotsave.hdf5")
```

At a later stage/session, it is possible to specify an additional conventional backend, retrieve the plot, and display it:

```
# Switch the backend to GR
julia> gr()
Plots.GRBackend()

julia> p = Plots.hdf5plot_read("plotsave.hdf5")
```

In the next section, we will turn to more comprehensive visualization packages that are capable of creating quite complex displays more easily.

## Visualization frameworks

A graphics framework provides a high-level interface to create complex visualizations as easily as possible. Ideally, we would like to acquire the data, specify a minimum of parametrization to identify the layout, labels, and so on, and pass these to a routine to produce the overall display with little (or no) knowledge of the underlying plotting methods.

### Plotly/PlotlyJS

Plotly is a data analysis and graphing web application that can create precise and beautiful charts. It is based on D3 and, as such, incorporates a high degree of interaction such as hover text, panning, and zoom controls, as well as real-time data streaming options.

We will start by looking at PlotlyJS. It does not interact with the Plotly web API; instead, it uses the underlying JavaScript library to construct graphics using all local resources. This means that neither a Plotly account nor an internet connection is needed to use this package. However, plots can be uploaded into the cloud, as we will see later.

The routines, and their syntax, reflect their Plotly heritage; you are advised to turn to the documentation for `Plotly.jl` as they can be used in both heritages.

Visual displays created by PlotlyJS are web-based, incorporating a degree of interactivity without any additional coding.

Since the underlying displays for Jupiter and Pluto notebooks are web-based, this presents no difficulty. PlotlyJS is normally used as a backend of choice for Plots, especially in circumstances where a degree of interactivity is required.

#### *Example – a box plot of stock prices*

A box and whisker plot uses boxes and lines to depict the distributions of one or more groups of numeric data. The box limits indicate the range of the central 50% of the data, with a central line marking the median value.

Lines extend from each box to capture the range of the remaining data – that is, maximum to minimum (whiskers) – with dots placed past the line edges to indicate outliers. These normally lie outside three standard deviations.

The example I have chosen uses the stock market values for Google, Apple, Amazon, and Microsoft for 2018 to 2019.

All stocks have been normalized to start with their closing values on 01-01-2019:

```
julia> cd(ENV["HOME"] * "/MJ2") ;  
julia> df = CSV.read("DataSources/CSV/stocks.csv", DataFrame) ;
```

```

julia> trace1 = box(;x = df[!, :GOOG], name="Google");
julia> trace2 = box(;x = df[!, :AAPL], name="Apple");
julia> trace3 = box(;x = df[!, :AMZN], name="Amazon");
julia> trace4 = box(;x = df[!, :MSFT], name="Microsoft");
julia> data = [trace1, trace2, trace3, trace4];
julia> layout = Layout(
    title="Selected Stock Values for 2018-2019",
    xaxis=attr(title="Normalized to 1.0 on 1st January 2018",
    zeroline=false));
julia> plot(data, layout)

```

Here's the output:

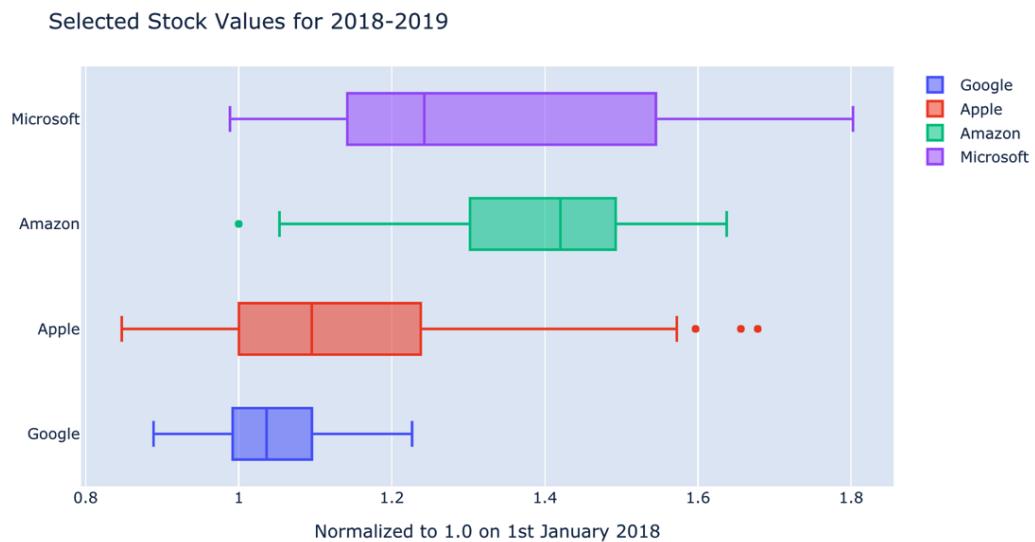


Figure 8.20 – Box and whiskers plot for four company stock prices for 2018 to 2019

PlotlyJS is very useful in conjunction with web-based workbooks such as Jupyter or Pluto since a large-scale degree of interaction can be achieved. We will not delve into this topic in detail in this book; however, I will return to it in the final chapter when we review some miscellaneous topics when developing Julia.

### *A contour plot in the clouds*

Originally, access to Plotly was done via a REST API to <http://plot.ly>, but a variety of programming languages can now access the API, including Julia.

To use Plotly, you will need to sign up for an account via <http://plot.ly> and provide a unique username and email address.

Upon registering, an API key will be generated and emailed together with a confirmation link. All plots are stored under this account and can be viewed and managed online, as well as embedded in web pages.

On registering with Plotly, or (say) switching computing platforms, it is important to create a hidden `.credentials` file in the user account in a `.plotly` directory.

This can be done manually; however, a simple routine is available:

```
julia> Plotly.set_credentials_file(  
Dict("username"=>"sherrinm", "api_key"=>"<<API key>>") )
```

The API key is (currently) a 20-character ASCII string that's returned on registration by Plotly and can be viewed, if forgotten, by logging into <https://plot.ly>.

Every piece of code requires a call to the `signin()` routine:

```
julia> using Plotly  
julia> Plotly.signin("sherrinm", "<<Plotly API key>>")
```

On successful execution, `signin()` returns a `PlotlyAccount` data object, and an online graph is created under that account by formulating and executing a response function.

Note that Plotly graphics are stored by number, not name. The visualization is uploaded as a data stream that is interpreted and rendered by the JavaScript engine.

The response function posts the data to Plotly, creates the plot, and generates a URL for it as a reply.

Here is a contour plot of some sinusoids with a randomly generated component:

```
Julia> using Plotly  
julia> N = 100;  
julia> X = collect(range(-2*pi, stop=2*pi, length=N));  
julia> Y = collect(range(-2*pi, stop=2*pi, length=N));  
julia> Z = zeros(N, N);  
julia> for i = 1:N, j = 1:N  
r2 = (X[i]^2 + Y[j]^2)  
Z[i,j] = sin(X[i]) * cos(Y[j]) * sin(r2)/log(r2+1)  
end  
julia> data = contour(z=Z , x=>X, y=Y);
```

Now, we need to create a `resp` response object and post it to the cloud:

```
julia> resp = Plotly.plot(data);  
julia> plot_url = Plotly.post(resp);  
RemotePlot(URI("https://chart-studio.plotly.com/~sherrinm/120/#/"))
```

The plot can be viewed from my Plot.ly account as has been made public: <https://chart-studio.plotly.com/~sherrinm/120/#/>.

Again, note that the Plotly graphs, charts, and more are saved by number, not by name.

*Figure 8.21* shows the results that can be retrieved by using the `Plotly.download_plot()` routine and referencing the preceding URL:

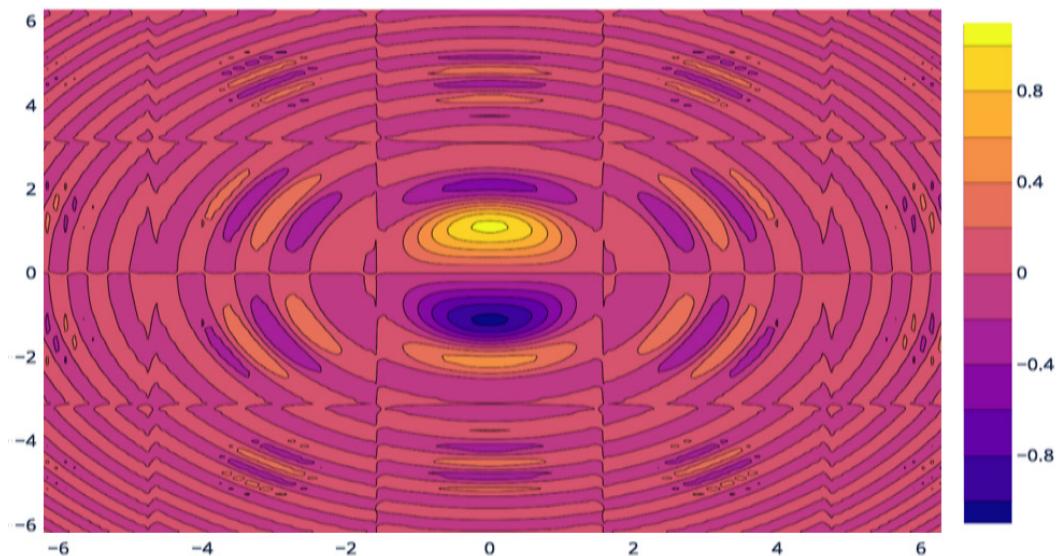


Figure 8.21 – Contour plot set up and then downloaded from Plotly

Uploading to the cloud defaults to public rather than private access, but the status of the diagram can be changed via the user's online Plotly account.

## StatsPlots

To some extent, StatsPlots can be viewed as a stand-in for the Plots API, a little less weighty but one that implements many statistical methods. As such, it was another approach proposed by Tom Breloff. It is now maintained by the JuliaStats group.

It can deal directly with DataFrames and naturally encompasses the concepts of the Distributions packages. It is usually possible to load a dataset, apply a statistical procedure to the dataset, and display the results all within a single call.

In *Chapter 6*, we looked at the dataset for GSCE results in a series of schools in the UK, and the differences between marks for written (exam) versus coursework and between male (boys) and female (girls) students. Let's start by loading it from the RDatasets package:

```
julia> using StatsPlots, RDatasets
julia> mlmf = dataset("mlmRev", "Gcsemv");
```

The dataset contains missing values both in the written (exam) and coursework marks. To apply statistical procedures, these will need to be removed (as we did previously).

We can use this step to differentiate based on gender:

```
julia> wF = collect(skipmissing(
    mlmf[mlmf.Gender .== "F", :Written]));
julia> wM = collect(skipmissing(
    mlmf[mlmf.Gender .== "M", :Written]));
julia> cF = collect(skipmissing(
    mlmf[mlmf.Gender .== "F", :Course]));
julia> cM = collect(skipmissing(
    mlmf[mlmf.Gender .== "M", :Course]));
```

Now, by using the `@df` macro from StatsPlots, we can pass columns within an array and then call the `density` function to display the spectral density of the four reduced datasets.

First, we can set up the legend as a legend – I've replaced `Written` with `Exam` since in this case, it is a tautology:

```
julia> labs = ["Exam (Girls)" "Exam (Boys)"
               "Course (Girls)" "Course (Boys)"];
julia> @df mlmf density([wF, wM, cF, cM],
                        labels=labels, legend = :topleft)
```

The resultant plot can be seen in *Figure 8.22*:

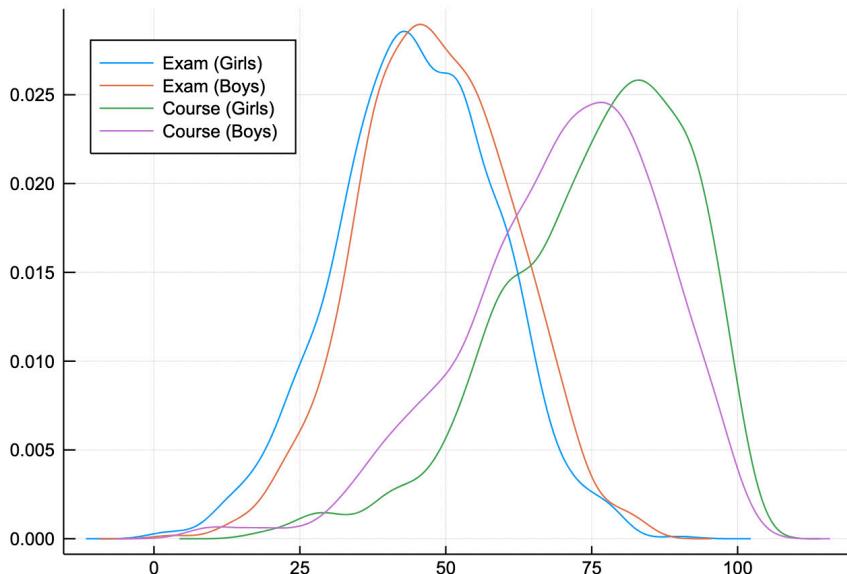


Figure 8.22– Special density plot for coursework and written work separated by gender

Recall that spectral density is the analog of a frequency histogram for continuous data distributions. There is little difference to be seen based on gender; however, there is a marked discrepancy when looking at examination and coursework marks.

That being the case, let's examine the latter and use StatsPlots to view the differences. First, we will need data without any missing values in either of the two sets of marks:

```
julia> mlmfX = mlmf[completecases(mlmf), :]
1523×5 DataFrame
```

This reduces the original dataset from 1,905 to 1,523 values, which is still a reasonable number to work with. However, it could be argued that missing values were due to examinations missed and/or coursework not presented, so these values should be set to zero.

To visualize the dataset, we can plot it against schools:

```
julia> @df mlmfX plot(:School, [:Written :Course])
```

The output can be seen in the following figure:

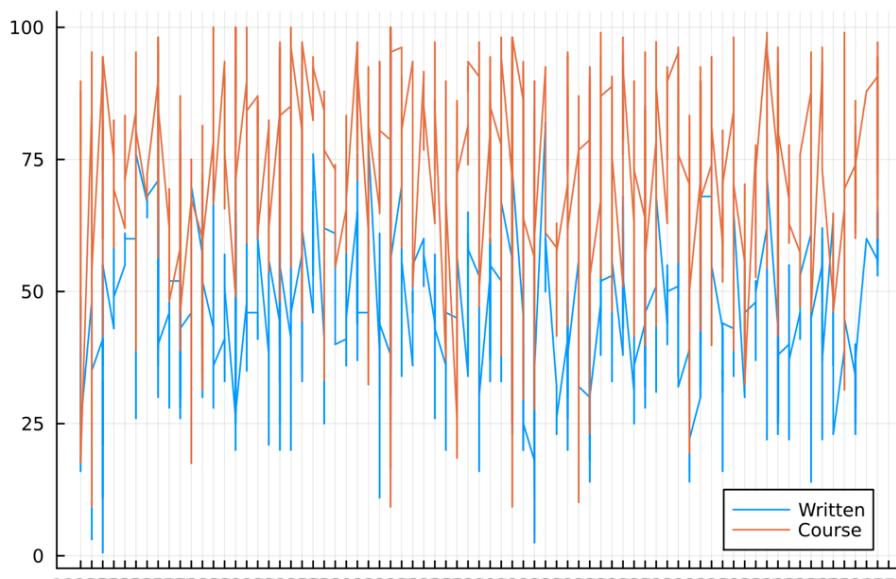


Figure 8.23 – Written (exam) by coursework marks in selected schools for UK GSCE

It's a little messy but it confirms what can be seen by eying the data – that the exam marks are much lower in the vast majority of cases.

*A color view is easier, as can be seen in the code accompanying this chapter.*

So, let's look at a correlation plot of the two datasets combined with a dot plot of each. For brevity, I have combined both plots in *Figure 8.24*:

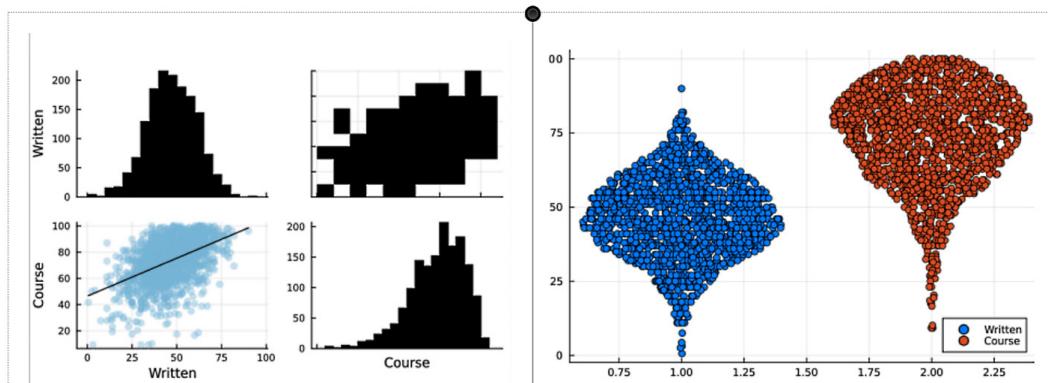


Figure 8.24 – Correlation and dot plot for written versus coursework marks

These were generated with the following code:

```
julia> @df mlmfX corrplot([:Written :Course])
julia> @df mlmfX dotplot([:Written :Course])
```

The interesting part of the correlation plot (an LHS of 8.24) is the lower quadrant, which shows the regression line through a very thick set of data, with a few (mainly low) outliers. The other quadrants in this plot are histograms of the data and the correlation.

The dot plot on the RHS is similar to a violin plot, which StatsPlots can also generate, but it's much less fun.

One last thing we can do before wrapping up is use the Queryverse to filter out the coursework marks for all those who did well in the exams, say with over a 60% mark:

```
julia> using Query
mlmfX |> @filter(_.Written > 60) |> @df scatter(:Course)
```

This is displayed as a scatter diagram in *Figure 8.25*:

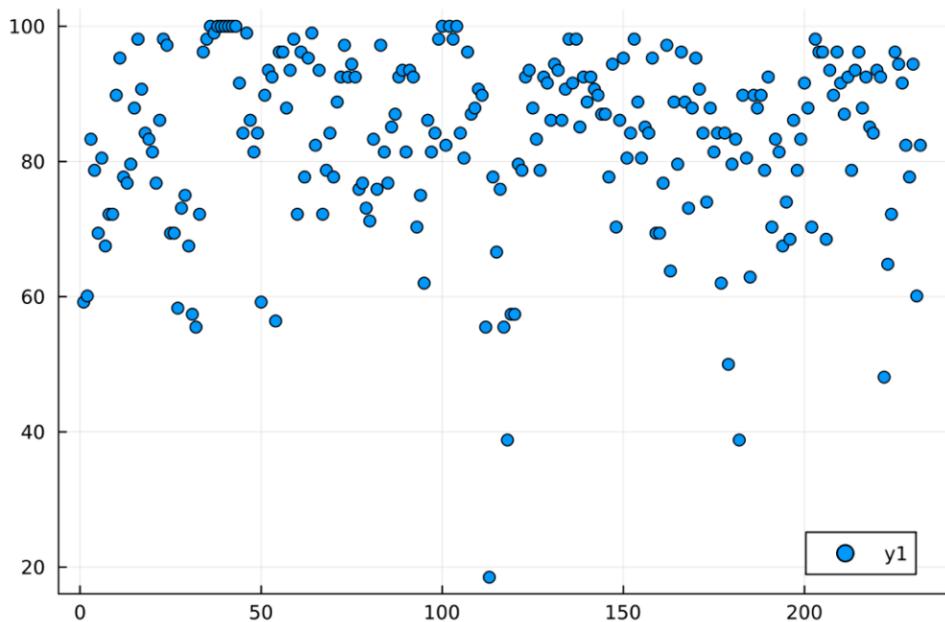


Figure 8.25 – The coursework mark for students is 60% in their examinations

As can be seen, despite a few, and one notable, exception, we can see students who had good marks in their exams and performed very well in their coursework.

I am going to leave StatsPlots now and turn to the behemoth that is Makie.

## Makie

Makie is a high-level plotting interface for GLVisualize, with a focus on interactivity and speed. It uses the GPU to provide its displays but can operate via the CPU alone, albeit somewhat slowly.

Makie is the frontend package that defines all plotting functions. These are reexported by a variety of backends, so they do not have to be specifically installed or imported.

Four backends concretely implement all abstract rendering capabilities defined in Makie:

- **GLMakie.jl**: A GPU-powered interactive 2D and 3D plotting backend in standalone GLFW.jl windows
- **CairoMakie.jl**: A Cairo.jl-based, non-interactive 2D (*and some 3D*) backend typically for publication-quality vector graphics
- **WGLMakie.jl**: A WebGL-based interactive 2D and 3D plotting backend that runs within browsers
- **RPRMakie.jl**: A backend that uses RadeoProRender for ray tracing Makie scenes

Each backend uses the `activate!()` function, which optionally takes keyword arguments that control various aspects of the backend.

For example, to activate the GLMakie backend and set it up to produce windows with a custom title and no anti-aliasing, we can use the following code:

```
julia> using GLMakie
julia> GLMakie.activate!(title="Plot title", fxa=false);
```

A bewildering variety of examples have been created and can be viewed on the Makie website: <http://makie.juliaplots.org>.

We will just look at a couple here using GLMakie's backend.

First, we'll look at an obligatory "Hello World" plot – that is, a simple line plot. This can be seen in *Figure 8.26*:

```
# Define two target functions ...
julia> using Makie
julia> GLMakie.activate!(title="Plot title", fxa=false);
julia> f1(u) = sin.(u) ./ (1 .+ u)
julia> f2(u) = u.*exp.(-0.5u) .* cos.(u)

#= ... and create some values for a specific range = 0:4pi, length =
80)for both functions  =
julia> h = pi/20;
julia> x = collect(0:h:4pi)
julia> y1 = f1(x);
julia> y2 = f2(x);
```

Makie uses the concept of a scene and a canvas to create the visualization. Lines are plotted between points using the `lines()` routine and individual points using `scatter()`:

```
= Create the scene and the line plot of y1, overlaying it with the
data points =
julia> scene = lines(x, y1, color = :blue)
julia> scatter!(scene, x, y1, color = :red, markersize = 0.1)

#= Do the same for y2, using a different colour and marker, then
display the finished "scene". =
julia> lines!(scene, x, y2, color = :black)
julia> scatter!(scene, x, y2, color = :green,
               marker = :utriangle, markersize = 0.1)
julia> scene
```

Notice the conventional use of `!-style` functions, which refer to an existing scene as the first parameter and overlay onto it:

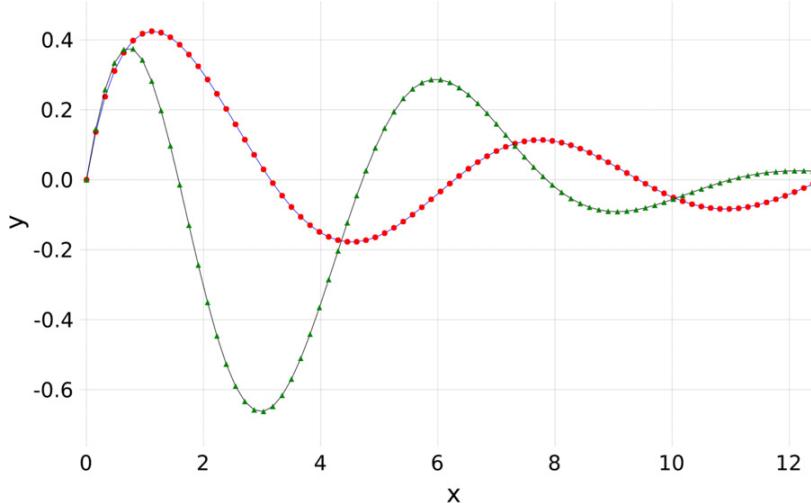


Figure 8.26 – Two simple line curves with associated points using Makie

Secondly, we are returning to the Iris dataset from `RDatasets`, which we used to create a correlation, and Andrews plots via `StatsPlots`.

A scatter plot of the data is a well-known example from the earliest days of Gadfly and has been reproduced here. To recreate the display, we will require a little more coding to navigate the `DataFrame`:

```
julia> using RDatasets, DataFrames
julia> iris = dataset("datasets", "iris")
```

We can create an empty scene using the `Scene()` constructor and build up the scatter diagram. The resultant plot is shown in *Figure 8.27*.

To categorize the different species, we will iterate over `iris[:Species]` and get the corresponding `SepalWidth` and `SepalLength` values, adding the data points using `scatter!()`:

```
julia> scene = Scene();
julia> colors = [:red, :green, :blue];
julia> i = 1;      # A color incrementer
julia> for sp in unique(iris[!, :Species])
    idx = iris[!, :Species] .== sp
    sel = iris[idx, [:SepalWidth, :SepalLength]]
    scatter!(scene, sel[:, 1], sel[:, 2], color = colors[i], limits =
Rect(1.5, 4.0, 3.0, 4.0))
    global i = i + 1
```

```

end;
# Add the axes and label then show the scene;
julia> axis = scene[Axis]
julia> axis[:names][:axisnames] = ("Sepal width",
                                    "Sepal length")
julia> scene

```

Here's the output:

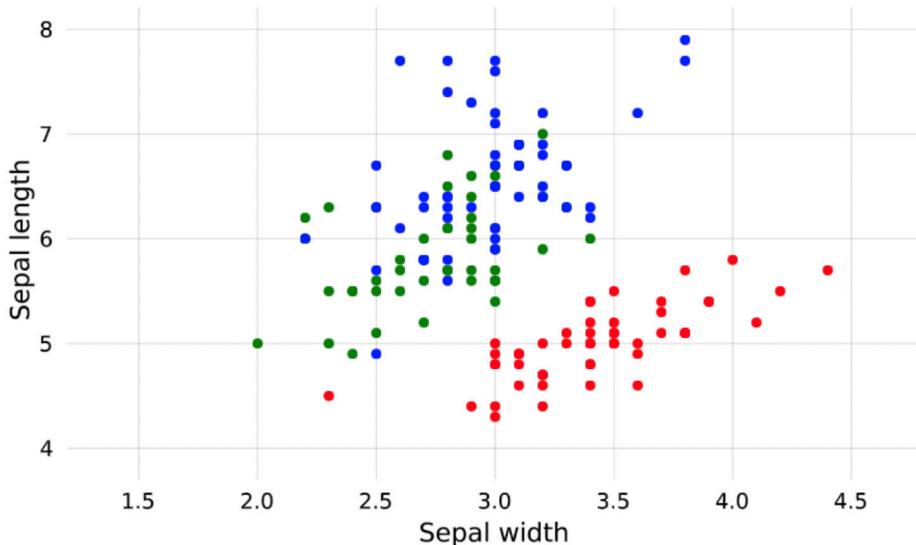


Figure 8.27 – Sepal length versus width for different species in the Fisher Iris dataset

In the final section of this chapter, we will leave the world of creating graphic displays and briefly discuss the alternative topic of manipulating raster visualizations.

#### Image processing of raster displays

Working with images and colormaps at the pixel level is often referred to as raster graphics, as opposed to vector graphics or using textual markup.

Since graphics packages such as Plots PythonPlot, Winston, and others eventually translate their plots into rasters to create a display, the majority of the packages we've discussed so far can work with images directly – that is, without an explicit framework.

This is true at least as far as loading, displaying, and saving images go, and since cropping in Julia can be achieved by matrix slicing, that is also quite easy to do.

So, before we skim the upper reaches of the `Images.jl` package and its associates, we will look at this first.

## Basic image processing

Some simple procedures can be done when processing images, as we saw when we discussed convolution methods in *Chapter 6* to perform edge detection on the “Lena” image.

For two of these, cropping and resizing, I will use image data from the `TestImages` package. Refer to <https://testimages.juliaimages.org/stable/imagelist> for a list of those currently available.

So far, we have displayed images using the `ImageView` package and the `imview()` routine, but this can also be done using the `Plots` API or `Makie`. We will display one of each here.

Cropping is straightforward since it can be done easily using array slicing, which is fast and efficient in Julia.

We will use an image of a fighter jet from `TestImages`, cropping it to display only the plane:

```
julia> using TestImages
julia> jet = testimage("jetplane");
julia> size(jet)
(512, 512)
```

Notice that the type of image is not usually specified in the `testimage()` routine. They are normally as TIFF or PNG and once loaded, the internal representation in Julia is completely equivalent, regardless of its original format, as we will see later.

To get the portion containing the plane, we will extract 200 pixels on the first index, leaving the second as-is:

```
julia> jetC = jet[121:320, :];
```

To display the resulting image, which can be seen in *Figure 8.28*, we will use the `image()` function from `GLMakie`:

```
julia> using GLMakie
julia> image(rotr90(jetC), axis = (aspect=DataAspect(),
    title="Leaving on a Jetplane ?"))
```

The `image()` routine is unusual as it places the origin at the top left, displaying the image on its side. So, a reverse rotation of 90° is required to orient it properly. This can be done using the `rotr90()` function:



Figure 8.28 – A cropped image of the fighter jet showing only the plane

### *Resizing*

When resizing, there are two aspects to consider – scaling up and scaling down.

In a sense, the former is trivial when scaling up by an integer amount as it only requires a broadcast multiplication over the image array. So, for example, when doubling the size of an image, a single pixel would become four. This is usually acceptable by eye although a convolution of nearest neighbors should be made, say with a 3x3 kernel to interpolate each pixel value.

Scaling down is trickier and I have included some code for this. I have used a test image of the Earth taken by the Apollo 17 mission.

This is a large image, some 3,000 x 3,000 pixels, and also in color. After getting the image, I want to convert it into grayscale and reduce it to a size of 500 x 500:

```
julia> using Colors, TestImages
julia> a17 = testimage("earth_apollo17");
julia> typeof(a17[1,1])
RGB{FixedPointNumbers.N0f8}
```

We need the `Colors` package to apply the `Gray` function.

Notice that the image type changes from `RGB{FixedPointNumbers.N0f8}` to `RGB{FixedPointNumbers.N0f8}`:

```
julia> a17g = Gray.(a17);
julia> typeof(a17g[1,1])
Gray{FixedPointNumbers.N0f8}
```

The reduction will be by a sixth in both dimensions, though we could use a different integer scale-down factor for both the width and the height:

```
julia> k = 6; kr = 1/k*k;
julia> sz = size(a17g);
julia> (m,n) = [convert(Integer,floor(sz[i]/k))
               for i = 1:2];
```

In this case, the resizing will be done by averaging over 36 pixels, so we'll define a zeroed array to hold the final values:

```
julia> a17r = zeros(Gray{Float32},m,n);
julia> for i in 1:(m-1)
           for j in 1:(n-1)
               for h in 1:k
                   a17r[i,j] += kr*a17g[6*i + h, 6*j + h]
               end
           end
       end
```

A couple of points to note are that the `a17r` matrix is set to the `Gray{Float32}` type, not `Gray{FixedPointNumbers.NOf8}`; this is because scaling the sum of the pixels by a real number, (`kr`), will implicitly change the data type:

```
julia> sf = 1.0/maximum(a17r);
julia> a17s = sf .* a17r;
```

Interestingly, the `ImagesView.imshow()` routine, which we covered earlier in this book, would display `a17r` correctly because it implicitly scales the pixel values in the range of 0.0 to 1.0.

This is an extra step that we had to do manually to use `Plots.plot()` to display the result – hence rescaling by the `sf` factor to produce the `a17s` image:

```
julia> using Plots; gr()
Plots.GRBackend()
julia> plot(a17)
julia> plot(a17s)
```

The original image and the resized one are shown side by side in *Figure 8.29*, with no noticeable difference to the eye, except for the change of increment ticks in the *X* and *Y* axes:

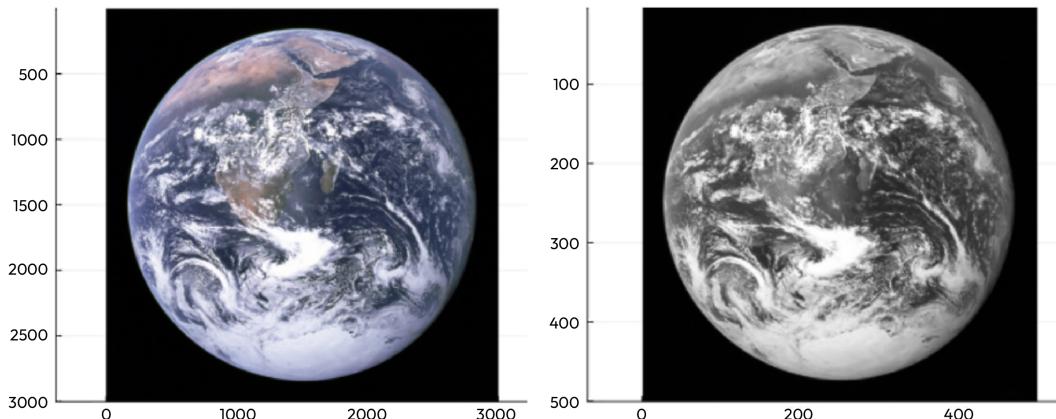


Figure 8.29 – The Earth viewed from Apollo 17 together with the resized grayscale image

There is so little apparent difference that I have chosen to show the color image (`a17`) on the LH side rather than the grayscale one (`a17g`).

Resizing is a function that can be more simply performed by using the `Images.jl` suite of routines. This is what we will turn to next.

## The `Images(jl)` family

The set of packages from Tim Holy (and now others) provides the most comprehensive support for manipulating images in Julia.

The main package is `Images.jl`. It started as a single relatively comprehensive package. Originally, it utilized programs from the `ImageMagick` suite, but now, it has been greatly expanded and implemented as a purely native set of modules.

As with the frameworks we have just introduced, it's possible to devote an entire chapter, probably much more, to this topic.

For serious image-processing enthusiasts, it is the *go-to* source. Here, we will look at a couple of examples and see how easy it is to implement an analysis using `Images`.

The standard paradigm is very straightforward:

1. Acquire and load an image.
2. Call one, or more, functions to do the processing.
3. Display and/or save the result.

Acquisition (*Step 1*) is usually done via the `load()` routine from a package such as `TestImages` or via the web, which we will discuss in *Chapter 11*, and is displayed (*Step 3*) invariably via the `ImageView` package.

#### Note

There is an interesting package that can be used to display images when working with the REPL that can be found at <https://github.com/JuliaImages/ImageInTerminal.jl>; it works with a certain class of terminal command shells, all of which are discussed in its GitHub repository.

Now, we will concentrate on the intermediate tasks in *Step 2*, with a couple of examples *both* involving bridges.

### A tale of two bridges

The `TB1.jpg` image, provided in this chapter, is a color view of London's Tower Bridge.

The following code turns it into a grayscale image, blurs it with a convolution using a 4x4 Gaussian kernel, and displays it, together with the grayscale as a one-row mosaic:

```
julia> using Images, ImageView, Colors
julia> using ImageFiltering, MosaicViews
julia> DS = ENV["HOME"] * "MJ2/DataSources";
julia> img = load("$DS/Files/TB1.jpg")
julia> typeof(img)
Matrix{RGB{N0f8}}
julia> imgB = imfilter(img, Kernel.gaussian(4))
julia> imgX = mosaicview(img, imgB, nrow=1, npad=20,
                           fillvalue = Gray{N0f8}(1.0))
julia> imshow(imgX);
```

This has been done using the `ImageFiltering` package. Additionally, we have used `MosaicViews` to create the display of both images, as shown in *Figure 8.30*. The `Colors` package is needed to apply the `Gray()` routine:

#### Note:

There is no need to include `FileIO.jl` as this is imported implicitly by `Images.jl`.



Figure 8.30 – London’s Tower Bridge with a blurred image, using a Gaussian 4x4 filter

For those interested, I have included a Pluto workbook with the code accompanying this chapter to implement the same process as `towerbridge.pluto.jl`.

The second example is a bridge that is not in London. It is a nighttime image of a bridge, probably over the East River in New York City, hence the filename I have chosen.

I wish to perform edge detection but because of the low contrast, the technique we applied to *Lena* in *Chapter 6* will not suffice. Fortunately, there is a package we can use for `ImageEdgeDetection` that thresholds the image to a black-on-white image before applying an edge detection kernel:

```
julia> using ImageEdgeDetection, ImageTransformations
julia> img = load("../Files/NYB2.png")
julia> typeof(img)
Matrix{RGB{N0f8}}
```

Use `ImageTransformations.jl` to do a 50% resize. The package can do other transformations of course, such as rotations and warping.

Note that even though the image is a grayscale one, the image type for a PNG is RGBA, so we still have to broadcast the `Gray()` function to the image before using the `detect_edges()` routine:

```
julia> imgR = resize(img, ratio=0.5);
julia> imgE = detect_edges(Gray.(imgR),
                           Canny(spatial_scale=1.2))
julia> imgX = mosaicview(imgR, imgE, nrow=1, npad=20,
                           fillvalue = Gray{N0f8}(1.0))
julia> imshow(imgX);
```

The `Canny()` method uses a multi-stage algorithm to detect a wide range of edges in images and was developed by John Candy in the 1980s.

The `spatial_scale` parameter is selected to determine the level of edge detection.

The code then creates a mosaic view, as per the previous example. This is shown in *Figure 8.31*:



Figure 8.31 – Nighttime view of an NYC bridge and subsequent edge detection

The actual algorithm is quite complex and there is an excellent discussion of it on Wikipedia: [https://en.wikipedia.org/wiki/Canny\\_edge\\_detector](https://en.wikipedia.org/wiki/Canny_edge_detector).

The `Images` package has a whole raft of other methods and can work with colored images, colormaps, pseudocolor, and more. This will not be discussed here, so you are encouraged to refer to the `JuliaImages` group documentation on GitHub: <https://github.com/JuliaImages>.

## Summary

This chapter has presented a wide variety of options for producing visualizations that are now available to Julia programmers.

First, we looked at some of the popular “golden oldies” packages, such as `UnicodePlots`, `Winston`, `Gadfly`, `PyPlot`, and `PGFPlots`. After this, we introduced the `Plots` API, together with some of the newer backends, such as `GR` and `PlotlyJS`.

Then, we looked at how the `Plotly` cloud-based system can be utilized in Julia to generate, manipulate, and store data visualizations online. The `Plots` API makes provisioning graphic frameworks such as `StatsPlots` and `Makie` possible; we discussed these briefly.

Finally, we looked at how raster graphics can be processed and displayed.

In the next chapter, we will return to the subject of accessing data by looking at the various ways with which we can interact with SQL and NoSQL databases in Julia, delving into the `Queryverse` and introducing one further graphics package: `Vega Lite`.



# 9

# Database Access

In *Chapter 6, Working with Data*, we looked at working with data that is stored in disk files: we looked at plain text files and also datasets taking the form of R data files, CSV, HDF5, and so on.

In this chapter, we will consider data stored in databases but will exclude the “big data” data repositories as networking and remote working are the subjects of the next chapter.

There are several databases of various hues, and as it will not be possible to deal with all that Julia currently embraces, we will pick one specific example for each category as they arise.

Specifically, we will look at the following topics:

- Relational databases and SQL queries
- NoSQL databases
- The REpresentational State Transfer (REST) API
- The Queryverse

This chapter doesn’t intend to delve into the workings of the various databases or to set up any complex queries, nor to go in-depth into any of the available packages in Julia.

All we can achieve in this space is to look at the connection and interfacing techniques of a few common examples. However, before we get to them, I will provide some brief comments on databases.

## Database preliminaries

A few years ago, discussing databases was simple. We spoke of relational databases such as Oracle, MySQL, and SQL Server and that was pretty much it. There were some exceptions, such as databases involved in **Lightweight Directory Access Protocol (LDAP)** and some configuration databases based on XML, when the whole world was relational. Even products that were not purely relational, such as Microsoft Access, tried to present a relational face to the user.

We usually use the terms *relational* and *SQL database* interchangeably, although the latter more correctly refers to the method of querying the database than to its architecture. What changed was the explosion of large databases from sources such as Google, Facebook, and Twitter, which could not be accommodated by scaling up the existing SQL databases of the time, and indeed such databases still cannot.

Relational databases consist of a set of tables, normally linked with a main (termed primary) key and related to each other by a set of common fields via a schema. A schema is a kind of blueprint that defines what the tables consist of and how they are to be joined together.

NoSQL data stores are often termed to be schemaless, meaning it is possible to add different datasets with some fields not being present and other ones occurring. Data stores comprising text often fall into this category as different instances of text frequency throw up varying metadata and textual structures.

Possibly the biggest difference between SQL and NoSQL databases is the way they scale to support increasing demand. To support more concurrent usage, SQL databases scale vertically, whereas NoSQL databases can be scaled horizontally – that is, they can be distributed over several machines. In the era of big data, this is responsible for their new-found popularity.

## Interfacing to databases

Database access is usually done via a separate task called a **database management system (DBMS)** that manages simultaneous requests to the underlying data. Querying data records presents no problems, but adding, modifying, and deleting records impose “locking” restrictions on the operations.

There is a class of simple databases that are termed “file-based” that is aimed at a single user. A well-known example of this is SQLite; we will discuss the Julia support package in detail later.

With SQL databases, one other function of the DBMS is to impose consistency and ensure that updates are transactional safe. This is known by the acronym **Atomicity, Consistency, Isolation, and Durability (ACID)**. This means that the same query will produce the same results and that a transaction such as transferring money will not result in the funds disappearing and not reaching the intended recipient.

Although it may seem that all databases should operate in this manner, the results returned by querying search engines, for example, do not always need to produce the same result set. So, we often hear that NoSQL data stores are governed by the **CAP theorem** – that is, **consistency, availability**, and **partition tolerance**. We are speaking of data stores spread over several physical servers, and the rule is that to achieve consistency, we need to sacrifice 24/7 availability or partition tolerance; this is sometimes called the two out of three rule.

In considering how we might interface with a particular database system, we can (roughly) identify four mechanisms:

- A DBMS will be bundled as a set of query and maintenance utilities that communicate with the running database via a shared library that exposes a set of routines in an **application program interface (API)** to the user.

- A second common method of accessing a database is via an intermediate abstract layer, which itself communicates with the database API via a driver that is specific for each database. An early system was **Open Database Connectivity (ODBC)**, developed by Microsoft, and subsequently, **Java Database Connectivity (JDBC)**, for working with Java databases.
- Another approach can be used for any case where there is a Python module for a specific database system. Routines in the Python module are called from Julia, which are used to handle the interchange of data types between Julia and Python.
- Access to a database by sending messages to the database, to be interpreted and acted on by the DBMS. This is typified by usage over the internet using HTTP requests, typically GET or POST.

We will look at examples of all of these in this chapter.

However, I will not be dealing with datasets composed of XML-based filesystems in detail. We discussed the general principles when we looked at working with files and you are encouraged to re-read that chapter.

## Relational databases

The primary difference between relational and non-relational databases is the way data is stored. Relational databases were not the first architectures to be implemented; those based on single (and then multiple) indices and values preceded them. As we will see later, they are making something of a comeback with the constraints of handling large datasets.

Relational data is tabular by nature and hence stored in tables with rows and columns. Tables can be related to one another and cooperate in data storage as well as swift retrieval.

Data storage in relational databases aims for higher normalization, breaking up the data into the smallest possible logical tables (related) to prevent duplication and gain tighter space utilization.

While normalization of data leads to cleaner data management, it often adds a little complexity, especially to data management, where a single operation may have to span numerous related tables. Since the databases are on a single server and partition tolerance is not an option, in terms of the CAP theorem, they are consistent and available.

## Building and loading

Before we look at some of the approaches to handling relational data in Julia, let's create a simple script that generates a SQL load file.

The dataset we will use comprises a set of “quotes,” of which there are numerous examples online. We will find this data useful later in this book, so we will create a database here.

There are only three (text) fields separated by tabs (rather than commas) and separate records per line, as shown here:

```
category<TAB>author<TAB>quote
```

Some examples are as follows:

```
Classics<TAB>Aristophanes <TAB>You can't teach a crab to walk  
straight  
Words of Wisdom<TAB>Voltaire<TAB>Common sense is not so common
```

For the `etj.jl` script, we will assume that a simple command line will be as follows:

```
etj.jl quodata.tsv [ > loader.sql ]      # the output is to be piped to  
a database
```

We want to create a table for the quotes and another for the categories.

This is not normalized as there may be duplication in authors, but this denormalization saves a table join.

The downside is that we can extract quotes by category more easily than by author, which will require us to define a foreign index (with a corresponding database maintenance penalty) or an exhaustive search.

The following SQL file (`build.sql`) will create the two tables we require:

```
create table categories (  
    id integer not null,  
    category varchar(40) not null,  
    primary key(id)  
) ;  
create table authors (  
    id integer not null,  
    category author(40) not null,  
    primary key(id)  
) ;  
create table quotes (  
    id integer not null,  
    cid integer not null,  
    aid integer,  
    qtext varchar(250) not null,  
    primary key(id)  
) ;
```

In building the load file, we need to handle single quotes ('), which are used in SQL for text delimiters. The usual convention is to double them up (''), but some loaders also accept escaped backslashing (\').

Our script's command line is quite simple to check, so we will work with it directly. In the case of more complex command lines, Julia has an ArgParse package that has similarities with the Python argparse module, but it also has some important differences.

The command-line script can be implemented like so:

```
#! env julia
using DelimitedFiles
nargs = length(ARGS);
if nargs == 1
    tsvfile = ARGS[1];
else
    println("usage: etl.jl tsvfile");
    exit();
end
```

A one-liner is needed to double up single quotes:

```
escticks(s) = replace(s, "'" => "''");
```

Now, we can read all the files into a matrix (using `DelimitedFiles`):

```
# The first dimension is number of lines
qq = readdlm(tsvfile, '\t')
n = size(qq)[1];
```

Then, we can store all the categories in a dictionary:

```
j = 0;
cats = Dict{String, Int64}();
```

Finally, we can loop through to load up the quotes table:

```
for i = 1:n
    cat = qq[i,1];
    if haskey(cats,cat)
        jd = cats[cat];
    else
        global j = j + 1;
        jd = j;
        cats[cat] = jd;
    end
    sql = "insert into quotes values(i,jd,";
    if (length(qq[i,2]) > 0)
        sql *= string("'", escticks(qq[i,2]), "'");
    else
        sql *= string("null,");
    end
    sql *= string("'", escticks(qq[i,3]), "'");
```

```
    println(sql);
end
```

At this point, we can dump the categories:

```
for cat = keys(cats)
    jd = cats[cat];
    println("insert into categories(jd, 'cat') ;");
end
```

The file is read using the `DelimitedFiles` package, so we can specify the field separator as a tab. Because the quotes table will have an index in the categories table, we will store all new categories in a hash (that is, `Dict()`).

So, each category is checked to see if it is in the hash using `haskey()`; otherwise, the category index is bumped up and the new category is stored.

At present, the category has to be designated as global. Hopefully, this will not be forever. This same approach can be used for authors, thus making relational normalization better, but authors' names are sometimes presented differently, so we can search the database with an exhaustive query.

Any input lines containing single quotes (') will need to have these doubled up using the one-liner `escticks()` function.

After writing the SQL statements for adding records to the quotes table, the same for the categories table can be created from the hash.

Here, we are assuming that the categories do *not* contain single quotes; otherwise, we would need to apply `escticks()` here as well.

This produces an intermediate SQL load file that can be used with most standard loaders that can output to `STDOUT` and be piped into the loader. If the input file argument was also optional, it could be part of a Unix command chain.

The merit of using this approach is that it can be used to load any relational database with a SQL interface, and it is easy to debug if the syntax is incorrect or we have failed to accommodate some particular aspect of the data (UTF-8, dealing with special characters, and so on).

As a downside, we must drop out of Julia to complete the load process. However, we can deal with this by either spawning the (specific) database load command line as a task or inserting the entries in the database on a line-by-line basis via the particular Julia database package we are using.

## Interfacing with a database

By a native interface, in terms of interfacing with databases, I am referring to the case under which a package makes calls to an underlying shared library API rather than requiring some meta broker such as ODBC or JDBC or using calls to (*say*) a Python, or similar, module.

This is a little different from the notion of native versus wrapper packages, which was propounded previously when we discussed Julia modules themselves.

As a first example, we will look at a database that does not require a separate database server and without the necessity of a separate DBMS: SQLite.

### SQLite

SQLite can be built from source but there are also a variety of precompiled binaries that can be downloaded from <https://www.sqlite.org/download.html>.

#### Note

The SQLite executable is usually designated as `sqlite3` to differentiate it from the prior version. Here, the command-line prompt is `sqlite>`.

As a dataset, we will work with the "queries" tables, for which we created a build and load file in the previous section. I have included a build and load file with the files associated with this chapter – that is, `qbuild.sql` and `qloader.sql`.

To start SQLite, assuming it is on the execution path, we must type `sqlite3 [dbfile]`.

If the database file (`dbfile`) does not exist, it will be created; otherwise, SQLite will open the file. If no filename is given, SQLite will work with an in-memory database that can be later saved to disk.

SQLite has several options that can be listed by typing `sqlite3 --help`.

The SQLite interpreter accepts direct SQL commands terminated by `;`.

In addition to this, instructions to SQLite can be applied using a special set of commands prefixed by a dot `(.)`. These can be listed by running the `.help` command.

These commands can usually be abbreviated, so long as that form is unique. For example, `.read` can be shortened to `.re`; however, this does not apply to `.help` as this clashes with the command headers.

#### Note

This command is case-sensitive and must be written in lowercase.

So, we can use the following sequence of instructions to create a database from our quotes build/load scripts, *assuming we are in the DataSources/Quotes folder*:

```
$> cd $HOME/MJ2/DataSources/Quotes
$> sqlite3 quotes.db
  SQLite version 3.32.3 2020-06-18 14:16:19
Enter ".help" for usage hints.
sqlite> .read "qbuild.sql"
sqlite> .read "gloader.sql"
-- Check there are 3 tables and that the quotes table has 36 entries
sqlite> .ta
authors      categories   quotes
sqlite> select count(*) from quotes;
36
sqlite> .ex
```

The command line assumes that the quotes.db file does not exist, so you may have to delete it manually or use the qdrop.sql script to delete any existing entries. Note that it is not necessary to save the file under a different name.

To interface from this database in Julia, we will use the `SQLite.jl` package:

```
julia> using SQLite.jl
julia> dbfile =
ENV["HOME"] * "/MJ2/DataSources/Quotes/quotes.db";
julia> db = SQLite.DB(dbfile);
```

First, let's look at the database tables and their schema:

```
julia> SQLite.tables(db)
3-element Vector{SQLite.DBTable}:
SQLite.DBTable("categories", Tables.Schema:
:id      Union{Missing, Int64}
:catname Union{Missing, String})
SQLite.DBTable("authors", Tables.Schema:
:id      Union{Missing, Int64}
:autname Union{Missing, String})
SQLite.DBTable("quotes", Tables.Schema:
:id      Union{Missing, Int64}
:cid    Union{Missing, Int64}
:aid    Union{Missing, Int64}
:quotext Union{Missing, String})
```

---

Interface to this database and run a more sophisticated query that uses all three tables and outputs the result as a DataFrame, limiting it to the first four rows for the Science category:

```
julia> using DataFrames
julia> sql = """select a.autname, q.quotext from quotes q
           join authors a on a.id = q.aid
           join categories c on c.id = q.cid
           where c.catname = 'Science' limit 4""";
```

Run the SQL query using the DBInterface.execute() routine:

```
julia> df = DataFrame(DBInterface.execute(db,sql))
4×2 DataFrame
Row | autname          quotetext
    |
1   | Noelie Altito   The shortest distance between tw...
2   | Shaw`s Principle Build a system that even a fool ...
3   | Fingle`s Creed  Science is true. Don't be misle...
4   | Carl Zwanzig    Duct tape is like the force. It...
```

In addition to running queries, the SQLite module can be used for executing **database manipulation language (DML)** statements such as `insert`, `update`, and `delete` and to create and drop tables using SQLite methods rather than via SQL statements.

Finally, advanced SQL experts should note that it is possible to specify parametrized SQL statements using the `SQLite.Stmt()` routine to construct and prepare them and `SQLite.bind!()` to bind values to parameters.

You should refer to the three tables that make up the Quotes database as we will be covering this in the context of a few other database examples later in this chapter. However, next, we'll look at one of the most popular databases: MySQL.

## MySQL

MySQL is probably the most utilized open source database, with web-based packages such as Wordpress, Joomla, and others almost exclusively using it as their backend datastore.

This is *not* the case in the Enterprise, where Oracle – and on Windows servers, SQL Server – are the most widely used.

While it is possible to download and install MySQL directly, bundles such as Xampp (Linux), Mamp (macOS), and Wamp (Windows) can be downloaded for free and will install a MySQL DBMS. It will also download an Apache web server and PHP interpreter, plus a phpMyAdmin web application that can be used to create and maintain MySQL databases and more.

**Note**

Much of the work in this chapter and in *Chapter 11* on networking computing has been done on macOS using Mamp Pro.

Following on from Oracle purchasing MySQL, as part of its previous sale to Sun Systems, the original developers, led by Monty Widenius, recreated an alternative clone, MariaDB, from what remained open source. From a user viewpoint, MariaDB acts entirely as MySQL, in particular concerning the interface API.

Some of the major systems open source providers, such as Redhat, Ubuntu, and others, are now bundling MariaDB, rather than MySQL, which was part of their distros. In this book, when we refer to MySQL, we are referring to both MySQL and MariaDB.

In this case, I am going to present three different ways to interface with MySQL as in some instances, the techniques can be carried over to different database systems:

- Open DataBase Connectivity (ODBC)
- Native interfaces
- PyCall

In this section, I'll provide a more interesting dataset (Chinook) if you wish to study this in more detail.

The Chinook database dates back to 2012 but is still available. A good reference to it can be found on GitHub (<https://github.com/lerocha/chinook-database>).

Chinook is an open source equivalent to the Northwinds dataset that accompanied Microsoft Office with a similar schema and dataset. (*Chinook are winds of the North American plains*).

The Chinook database schema is shown in *Figure 9.1*:

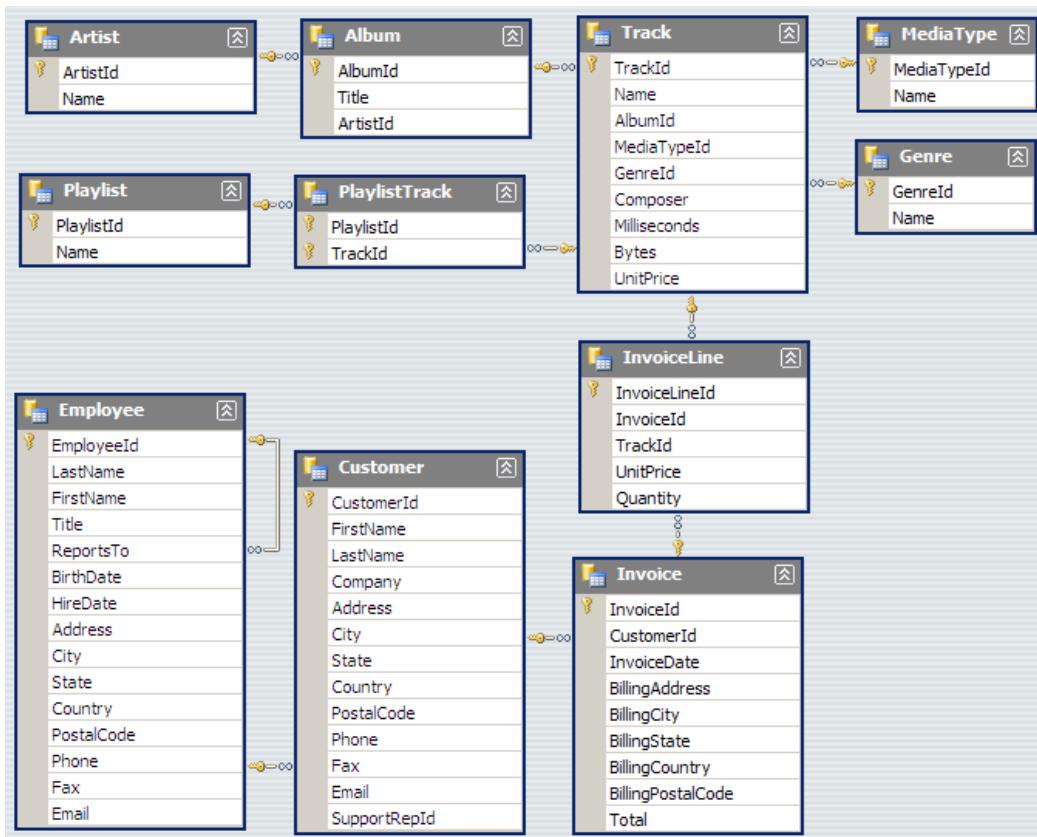


Figure 9.1 – Chinook database tables and schema

It represents a digital media store, including tables for artists, albums, media tracks, invoices, and customers.

The media-related data was created using real data from an iTunes library, although customer and employee information was manually created using fictitious names, addresses, emails, and so on. Information regarding sales was auto-generated randomly.

In the Chinook schema, there are two *almost* separate subsystems:

- Artist/Album/Track/Playlist, and so on
- Employee/Customer/Invoice

These are linked via the `InvoiceLine` (`TrackId`) table to the `Track` table.

Notice that the schema is not completely normalized as the InvoiceLine table can also be joined to PlaylistTrack and bypass the Track table.

## ODBC

ODBC was a level of middleware introduced by Microsoft in 1992. It adds an extra layer between the application program, the database interface, and the underlying database to impose a standard mode of access.

ODBC provides a fallback solution to interfacing with a database in cases where no other exists. However, one of its major criticisms is concerning speed, especially in situations involving large datasets and distributed (networked) systems.

I won't cover ODBC in detail. The Julia Database group provides reasonably comprehensive documentation (<https://odbc.juliadatabases.org/stable>), including a discussion on the problem of setting up ODBC connections on Unix and macOS.

### Note

This may be of more interest to users of Julia on Windows as ODBC interfacing (and their administration) is supported natively, even on Windows 11.

## (Native) MySQL

Julia provides a native package that wraps around the MySQL API and provides a comprehensive interface, so this is a better (and quicker) alternative to using ODBC.

As mentioned previously, I am running a MySQL database on a Mac macOS, using Mamp Pro, and have loaded the relevant Chinook SQL file. One of the features of MAMP is that the socket file, `mysql.sock`, is in a non-standard location, whereas the default is usually located in the `/tmp` folder.

However, the connection routine allows this to be specified; otherwise, we would need to set up a symbolic link that would be removed on reboot since `/tmp` is cleared.

We can set up the database to be used in the connection string. There are many parameters to do this, all of which are discussed at <https://mysql.juliadatabases.org/>.

The methods that we'll use are analogous to those we used previously when working with SQLite:

```
julia> conn = DBInterface.connect(MySQL.Connection,
    "localhost", "malcolm", "mypasswd", db="Chinook",
    unix_socket="/Applications/MAMP/tmp/mysql/mysql.sock");
```

It is possible to run a simple query to list the tables in the Chinook database:

```
julia> using DataFrames, MySQL;
julia> df =
```

```
    DataFrame(DBInterface.execute(conn,"show tables"));
julia> tb = df[:,1]
julia> for i = 1:size(tb)[1]
    print(tb[i], " ")
end
Album
Artist
Customer
Employee
Genre
Invoice
InvoiceLine
MediaType
Playlist
PlaylistTrack
Track
```

All the customers in the Chinook dataset are from the USA, so let's see how many have a last name starting with C:

```
julia> sql = "select FirstName,LastName,Address,City,State"
julia> sql *= " from Customer where LastName like 'C%'";
julia> DataFrame(DBInterface.execute(conn, sql))
2 rows x 5 columns
 FirstName LastName Address City State
 1 Kathy     Chase   801 W 4th Street Reno NV
 2 Richard Cunningham 2211 W Berry Street Fort Worth TX
```

By joining the `Invoice` table, we can sum up the `Total` field and see the overall total spent by these customers:

```
julia> sql = """select a.FirstName,a.LastName,
      sum(b.Total) as 'Total spend' from Customer a
      join Invoice b on a.CustomerId = b.CustomerId
      group by a.FirstName, a.LastName
      having a.LastName like 'C%'''';
julia> DBInterface.execute(conn,sql) |> DataFrame
2 rows x 3 columns
 FirstName LastName Total spend
 1 Kathy     Chase     37.53
 2 Richard Cunningham 47.62
# If we are done then it is polite to drop the connection
DBInterface.close!(conn);
```

Naturally, it is possible to use prepared statements and cursors via the DBInterface routines, but they will not be discussed here.

### **Using PyCall**

The other method I want to explore in working with a MySQL database is using routines from another programming language with the type of connectivity we discussed in *Chapter 5*. The choice here will be Python since we can make use of the PyCall interconnect.

Previously, we have seen that Python can be used for plotting via the PyPlot package that interfaces with `matplotlib`. The ability to easily call Python modules is a very powerful feature in Julia and we can use this as an alternative method to connect to databases.

Although Python is much slower compared to Julia, this is not as bad an option as it may appear at first glance since the interface to the DBMS will be written in C and compiled. In cases such as Oracle (currently), this may be an interesting stopgap, but this is no substitute for a direct wrapper around Oracle's OCI.

Our current MySQL setup already has the Chinook dataset loaded, so we will execute a query to list entries in the `Genre` table.

In Python, we will first need to download the MySQL Connector module to the default Python installation being used. We can check this by running Python and verifying that the `mysql.connector` module can be imported.

The query (*in Python*) to list the `Genre` table would be as follows:

```
# Running this in Python
# The mysql connector has to be installed
$> python
>>> import mysql.connector as mc
>>> cnx = mc.connect(user="malcolm", password="mypasswd")
>>> csr = cnx.cursor()
>>> qry = "SELECT * FROM Chinook.Genre"
>>> csr.execute(qry)
>>> for vals in csr:
...     print(vals)
...
(1, u'Rock')
(2, u'Jazz')
(3, u'Metal')
(4, u'Alternative & Punk')
(5, u'Rock And Roll')
...
...
```

```
>>> csr.close()
>>> cnx.close()
```

We can execute the same in Julia by using the PyCall module in the `mysql.connector` module. The code's form is remarkably similar:

```
julia> using PyCall
julia> @pyimport mysql.connector as mc;
julia> cnx = mc.connect(user="malcolm",password="mypwrd");
```

Any database that can be manipulated by Python is also available to Julia, so let's look at this approach to running the previous (Python) query:

```
julia> query = "SELECT * FROM Chinook.Genre"
julia> csr = cnx.cursor();
julia> csr[:execute] (query)
julia> for vals in csr
    id = vals[1]
    genre = vals[2]
    @printf "ID: %2d, %s\n" id genre
end
ID: 1, Rock
ID: 2, Jazz
ID: 3, Metal
ID: 4, Alternative & Punk
ID: 5, Rock And Roll
julia> csr[:close]()
julia> cnx[:close]()
```

Note that the form of the call is a little different from the corresponding Python coding. Since Julia is not object-oriented, the methods for a Python object are constructed as an array of symbols. For example, the `csr.execute(qry)` Python routine is called in Julia as `csr[:execute]`.

Also, be aware that although Python arrays are *zero-based*, this is translated to *one-based* references by PyCall, so the first value is referenced as `vals [1]`.

This rather lengthy discourse concludes all I wish to discuss here, but you are encouraged to run a series of queries against Chinook and look into the use of prepared statements and cursors.

Next, we will briefly look at two other relational databases – PostgreSQL and Apache Derby.

## PostgreSQL

I have been a long-time advocate of using Postgres as a relational database and have been working with it extensively. Postgres contained transactions, stored procedures, and triggers long before these were added to MySQL, mainly as an afterthought mimicking developments in MariaDB.

The original Julia package, `PostgreSQL.jl`, has now been retired and replaced by `LibPQ.jl` with a wrapper around the PostgreSQL shared library, which is available after installing Postgres.

Installation is system-dependent; as I am using macOS, it is easy to use a DMG package. Database access is done through a set of programs, so it is convenient to have these on the execution path so that it can be started using the `pg_ctl` program:

```
$> export PATH =
/Applications/Postgres.app/Contents/Versions/latest/bin:$PATH
$> pg_ctl -D /data/psq -l logfile start
$> ps -ef | grep postgres # check the server has started
```

We are going to work with Chinook; there is a load script available alongside the Chinook files accompanying this book.

It is necessary to create an empty database using `createdb` and then build/load the tables with `psql`:

```
$> cd $HOME/MJ2/DataSource/Chinook/Dataset
$> createdb chinook
$> psql -d chinook -a -f Chinook_PostgreSQL.sql
```

As an alternative to using Postgres utility programs, several administration tools are available, such as DBVisualiser and Postico; these usually need to be purchased but may provide a free tier:

```
julia> using LibPQ, DataStreams
julia> conn = LibPQ.Connection("dbname=chinook");
```

A feature of SQL in Postgres is that statements are case-insensitive, all covered in lowercase. Since the Chinook dataset tables and fields are in mixed case, they have to be quoted. In Julia, these quotes need to be escaped:

```
julia> res = execute(conn, "SELECT * FROM \"MediaType\"")
|> DataFrame;

julia> res = Data.stream!(res, NamedTuple)
(MediaTypeId = Union{Missing, Int32}[1, 2, 3, 4, 5],
 Name = Union{Missing, String}[
 "MPEG audio file", "Protected AAC audio file",
 "Protected MPEG-4 video file",
```

```
"Purchased AAC audio file",
"AAC audio file"])
```

Alternatively, the result, `res`, of the query can be found by using a single routine- that is, `fetch!()`:

```
julia> res = fetch! (NamedTuple,
    execute(conn, "SELECT * FROM \"MediaType\""));
julia> res[:Name]
5-element Array{Union{Missing, String},1}:
 "MPEG audio file"
 "Protected AAC audio file"
 "Protected MPEG-4 video file"
 "Purchased AAC audio file"
 "AAC audio file"
```

For complex queries, escaping all the tables and fields can become onerous, so it is better to use a Julia multiline string, in which case single quotes are acceptable.

The following query finds the number of sales made by different employees:

```
julia> sqlx = """ select e."FirstName", e."LastName",
    count(i."InvoiceId") as "Sales" from "Employee" as e
    join "Customer" as c on e."EmployeeId" = c."SupportRepId"
    join "Invoice" as i on i."CustomerId" = c."CustomerId"
    group by e."EmployeeId" """
```

Now, we can loop through the named tuple:

```
julia> qry = fetch! (NamedTuple, execute(conn, sqlx))
julia> using Printf
julia> for i in 1:length(qry)
    @printf("%s %s has %d sales\n",
        qry.FirstName[i], qry.LastName[i].qry.Sales[i])
end
Margaret Park has 70 sales
Jane Peacock has 97 sales
Steve Johnson has 84 sales
```

The last relational database we'll discuss here, Derby, is different in how it uses Java. The interconnect in this case is via the `JavaCall.jl` and `JDBC Julia` modules.

## JDBC databases

In *Chapter 5, Interoperability*, we discussed the use of the `JavaCall` package to interface Julia with the **Java Virtual Machine (JVM)**. This gives Julia access to the entire suite of JVM modules. One that's

especially important is **Java Database Connectivity (JDBC)**, so much so that a separate package called `JDBC.jl` has been written to simplify its use.

JDBC is another middle layer that functions similarly to ODBC, and naturally, there is a JDBC driver, termed a connector, for both MySQL and MariaDB that can be obtained from the latter's download page.

As an alternative to the examples used previously, I am going to discuss a pure Java database, Derby, and how to build/load/query it using the *quotations* dataset that we created earlier. Derby is an Apache DB subproject; it is an open source relational database implemented entirely in Java and available under the Apache License, version 2.0, which can be obtained from [https://db.apache.org/derby/derby\\_downloads.html](https://db.apache.org/derby/derby_downloads.html). I have included sources with this book.

First, we need to set up the Java classpath so that it includes the derby JAR files and also add the binaries to the executable path. The following commands are those typical of Linux and macOS; Windows will require a few modifications:

```
$> DH=$HOME/MJ2/db-derby  
$> export CLASSPATH=$DH/lib/derby.jar:$CLASSPATH  
$> export PATH=/Users/malcolm/MJ2/db-derby/bin:$PATH
```

Next, we must start the network server binary and switch to the code directory of this chapter. This is because the Derby database is a folder with the same name as the proposed database, which we wish to create as a subdirectory in `Code09`:

```
$> startNetworkServer  
$> cd $HOME/MJ2/Chp09/Code09
```

To create the database, we can use the `ij` program, which is part of the Derby suite of binaries, and so now on the path. We can make use of the same quotes to build and load the files that we used when working with SQLite:

```
ij> connect 'jdbc:derby:Quotes; create=true'  
ij> run '/Users/malcolm/MJ2/Datasources/Quotes/build.sql'  
ij> run '/Users/malcolm/MJ2/Datasources/Quotes/qloader.sql'  
ij> select count(*) from quotes;
```

The final statement checks that the database is up and running. There should also be a subdirectory called `Quotes` in the `README_DO_NOT_TOUCH_FILES.txt` file, a couple of lock files, and a folder called `seg0`, the latter of which contains the loaded dataset.

To run this in Julia, we need to use the `JavaCall.jl` and `JDBC.jl` packages. The `JULIA_COPY_STACK` environment variable must be set to 1 (or yes), which can be done before starting Julia or afterward, as we'll see shortly.

---

Before initializing the JVM, it is necessary to add `derby.jar` to the Java classpath so that we can pick up the Derby JDBC driver:

```
julia> ENV["JULIA_COPY_STACKS"] = 1;
julia> using JavaCall, JDBC
julia> derbyjar=ENV["HOME"]*"/MJ2/db-derby/lib/derby.jar");
julia> JavaCall.addClassPath(derbyjar);
julia> JavaCall.init(["-Xmx128M"]);
```

Running a query is quite straightforward.

We need to define `cursor` for the Derby Quotes database, define a SQL query, and then execute it.

As an example, let's list quotes from Oscar Wilde:

```
julia> csr = cursor("jdbc:derby:quotes");
julia> sql = """select q.quotext from quotes q
           join authors a on a.id = q.aid
           where a.autname = 'Oscar Wilde'""";
julia> execute!(csr, sql)
```

The `execute!()` statement modifies the cursor, returning the results by looping through the rows, as follows:

```
julia> for r in rows(csr)
           println(r[1], ".\n")
end
The only way to get rid of a temptation is to yield to it.
There is only one thing in the world worse than be talked about, and
that is not being talked about.

I am not at all cynical, I have merely got experience, which, however,
is very much the same thing.
```

To love oneself is the beginning of a lifelong romance.

We are all in the gutter, but some of us are looking at the stars.

London society is full of women of the very highest birth who have, of
their own free choice, remained thirty-five for years.

Note that in the rerun of the SQL statement, once the results have been obtained, it needs to be re-executed and a new cursor must be set up.

This concludes our discussion on relational databases. In the next section, we will turn our attention to another major type of database: NoSQL.

## NoSQL databases

Compared to their relational cousins, NoSQL databases are more scalable and provide superior performance, and their data model(s) address several issues that the relational model was not designed for when dealing with large volumes of structured, semi-structured, and unstructured data.

They have become important with the rise in the use of “big data,” in dealing with data from the internet, online purchases and other credit card transactions, and so on.

The different types of NoSQL databases can roughly be classified under the following headings, although no taxonomy is perfect:

- **Key-value (KV):** These are among the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or “key”), along with its value.
- **Document:** These pair each key with a complex data structure known as a document, which may contain many different KV pairs, key-array pairs, or even nested documents.
- **Columnar:** These are optimized for queries over large datasets and store columns of data instead of rows.
- **Graphic:** These are used to store information about networks using a data model consisting of nodes and their associated parameters, as well as the relationships between the nodes.

NoSQL datastore technologies are especially prevalent in handling systems that have a large volume of data and require high throughputs, and often, they are distributed over multiple physical servers.

We will briefly discuss some of Julia’s means of working with NoSQL databases both directly here and in the next section by employing the REST API.

### KV datastores

KV systems are the earliest databases, ironically preceding relational ones, and have now re-emerged in the age of large datasets.

The first was known as the **index sequential access method (ISAM)**. Despite the prevalence of relational databases in the SQL database era, they continued in various forms with their use in LDAP and **Active Directory (AD)** services. We will cover an example of the latter (**Lightning Mapped Database (LMDB)**) later in this section.

First, I wish to discuss **in-memory databases (IMDBs)**, database management systems that primarily rely on main memory for data storage rather than on disk.

Because IMDBs on volatile memory devices will lose all stored information when the device loses power or is reset, maintain checkpointing, snapshots, and transition logging.

A discussion of the (very) flexible KV database, **Redis**, follows, plus a brief mention of another, **Memcache**.

## Redis

Redis uses a simple set of commands to create and retrieve records through a messaging system between the server and the client. It has many data structures and commands that support them: simple scalars, lists, hashes, and sets. These are all supported by the `Redis.jl` Julia package.

The package also implements the publish/subscribe messaging paradigm, where published messages are delivered to designated channels, without knowledge of what (if any) subscribers there may be. Subscribers express interest in one or more channels and receive messages without knowledge of what (if any) publishers there are. This removes the need for polling the database to retrieve up-to-date information.

Setting up to use Redis is very straightforward and there are two principal methods:

- Get a Redis distribution and run it locally
- Use a web-based system (available from Redis Labs)

In the first case, downloads are available from <https://redis.io>. This includes a compressed source archive, Docker images, and binaries that will execute on Linux and macOS but are not officially supported on Windows. However, guidelines on how to run it there as well are included there.

### Note

For Windows users, my advice is to register with Redis Labs, which is free and provides generous data limits.

To work with the command line, it is necessary to use the `redis-cli` client utility, which is part of the method (1) downloads. There are also some Redis Management programs available, both web-based and as executables.

Alternatively, registration Redis Labs provides a unique URL (via AWS), a port number on which Redis is running, and an authentication password.

A local installation does not need the password but clearly, this is a necessity when working with a cloud-based service.

A simple session via `redis-cli` would be as follows:

```
# The server, port and auth string are not mine
$> redis-cli -p 99999 \
-h redis-99999.z99.eu-west-1-2.ec2.cloud.redislabs.com
# It is also possible to do this via the CLI using -a switch
redis> auth aBcde18FGhiJk1M77noPQrstuv
# PING will check all is working - if so responses PONG
redis> PING
PONG
# Set a simple key-value, check it is set and retrieve it
```

```
redis> set me "Malcolm Sherrington"
OK
redis> keys *
1) "me"
redis> get me
"Malcolm Sherrington"
```

With Redis.jl, we use the `RedisConnection()` routine – this defaults to using localhost and Redis' well-known port (6379), but it will accept parameters to connect to remote hosts.

So, in Julia, the preceding CLI session would be as follows:

```
julia> using Redis
julia> RLHOST =
    "redis-99999.z99.eu-west-1-2.ec2.cloud.redislabs.com";
julia> const RLPORT = 99999
julia> const RLAUTH = "aBcde18FGhiJklM77noPQrstuv"
julia> conn = RedisConnection(host=RLHOST,
    port=RLPORT, password=RLAUTH)
# Check with connection is alive and setup a key-values
julia> ping(conn)
PONG
julia> set(conn, "me", "Malcolm Sherrington")
OK
# Note that the return status for the SET is the string "OK"
# List the keys
julia> keys(conn, *)
Set(AbstractString["me"])
# ... and return the value
julia> get(conn, "me")
"Malcolm Sherrington"
```

Redis has several SET commands from its various data structures and corresponding GET commands, plus other associated message commands:

- Scalars (SET/GET)
- Multiple scalars (MSET/MGET)
- Lists (LSET/LINSERT/LPUSH/LPOP/LINDEX/LLEN)
- Hashes (HSET/HGET/HGETALL)
- Multi-hashes (HMSET/HMGET)
- Sets (SADD/SCARD)
- Sorted sets (ZADD/ZCARD/ZCOUNT)

There are a bewildering number of commands, of which the previously mentioned is a small subset.

See the online command reference documentation for a complete list and also the `commands.jl` file in the `Redis.jl/src` folder to see what has been implemented.

As an example, we will grab the stock market prices from Quandl (via NasDeq), store them in Redis, retrieve them, and display them graphically.

It is necessary to decide on the form of the key and the type of data structure to use. The key needs to be a composition that reflects the nature of the data being stored.

Quandl returns stocks against a four (or less) character code and a set of values such as Open, Close, High, Low, and Volume, plus the Date.

For Apple stocks, a choice of key for a closing price may be `APPL~Date~Open`, where `~` is a separator that will not occur in the data.

However, we could use a hash to store Open, Close, High, and Low against the Date, but to retrieve this data, we will need to make multiple queries.

It would be better to use a set of lists for each type of price (and the dates). So, we only need a couple of queries to get the data – one for the price and a second for the dates.

We would like to obtain update financial data from several sources.

Quandl datasets are now incorporated on Nasdaq with a wide variety of others sources and “historic” datasets can be downloaded by providing the symbol identifier and using the following link: <https://www.nasdaq.com/market-activity/quotes/historical>

Popular stocks such as Apple (APPL), Microsoft (MSFT), Amazon (AMZN) etc., have quick-links from this page.

Historical data are provided as CSV files, are *free* and do not require a Nasdaq API key. There are: 1M, 6M, YTD, 1Y, 5Y and MAX datasets available for OHLCV values (i.e., Open/High,/Low/Closed/Volume) which can be read from disk and processed using the CSV package.

It is also possible to grab data by means of the HTTP package and a REST request, more of which later in this chapter and are incorporated in the “Funky” module discussed in *Chapter 11*.

For it is necessary to obtain an API-key from Nasdaq:

```
julia> function nasdaq(ticker, apikey)
    wiki = "https://data.nasdaq.com/api/v3/datatables/WIKI";
    url = string(wiki, "/PRICES.json?ticker=", ticker, "&qopts.
    columns=date,open,close,volume&api_key=", apikey);
    res = HTTP.get(url);
    return JSON.parse(String(res.body));
end
```

This function returns open, close, and volume data for the full (MAX) data but is possible to formulate query strings, for limited date ranges and incorporating high, low and adjusted closing prices.

Calling this routine for Apple stocks returns a Dict as:

```
julia> n_appl = nasdaq("AAPL", "<Your-API-KEY">
Dict{String, Any} with 2 entries:
  "datatable" => Dict{String, Any}(
    "data"=>Any[
      Any["2018-03-27", 173.68, 168.34, 3.89628e7],
      Any["2018-03-26", 168.07, 172.77, 3.62726e7],
      "meta" => Dict{String, Any}("next_cursor_id"=>nothing)
    # The data is referenced via the "datatable"
    # Display the first 4 rows.
julia> n_aapl["datatable"]["data"] [1:4]
4-element Vector{Any}:
  Any["2018-03-27", 173.68, 168.34, 3.8962839e7]
  Any["2018-03-26", 168.07, 172.77, 3.6272617e7]
  Any["2018-03-23", 168.39, 164.94, 4.0248954e7]
  Any["2018-03-22", 170.0, 168.845, 4.1051076e7]
```

If preferred, this can be converted into a data frame, although the referencing in this case is a bit obscure.

```
julia> d_aapl = DataFrame(n_aapl)
julia> d_aapl[!, "datatable"] [1] ["data"]
9400-element Vector{Any}:
  Any["2018-03-27", 173.68, 168.34, 3.8962839e7]
  Any["2018-03-26", 168.07, 172.77, 3.6272617e7]
  Any["2018-03-23", 168.39, 164.94, 4.0248954e7]
  .
  .
  .
  .
  Any["1980-12-15", 27.38, 27.25, 785200.0]
  Any["1980-12-12", 28.75, 28.75, 2.0939e6]
```

Note that the Vector{Any} datatype as the first item, i.e. the date string. Is non-numeric.

For those who don't wish to register with Nasdaq, for a variety of stocks have been downloaded and are provided in the data sources folder accompanying this book: MT2/DataSources/WIKI; these can be read using CSV.jl, as previously discussed and then converted to data frames.

For a comparison of the Apple and Microsoft share prices it is useful to rescale to the prices relative to the prices on the (*same*) starting date.

Quandl data (normally) finishes on the same date but may begin on different dates for different stocks. So, we are going to plot the data for the period when *both* stocks were listed:

```
julia> n = minimum([length(aapl), length(msft)]);
julia> t = collect(1:n);
```

We will push these to Redis as a list using the AAPL~Close and MSFT~Close keys:

```
julia> conn = RedisConnection()
julia> for i = n-1:-1:0
    rpush(conn, 'AAPL~Close', aapl[end-i])
    rpush(conn, 'MSFT~Close', msft[end-i])
end
```

Now, we can retrieve the values against these values and plot the results:

```
# Redis lists are zero-based
julia> aapl_rdata = float.(lrange(conn, "AAPL~Close", 0, n-1));
julia> msft_rdata = float.(lrange(conn, "MSFT~Close", 0, n-1));
julia> using PyPlot
julia> plot(t, aapl_rdata, color="red", label="AAPL")
julia> plot(t, msft_rdata, color="blue", label="MSFT")
julia> xlabel("Time")
julia> ylabel("Scaled Price")
julia> legend()
```

Here's the output:

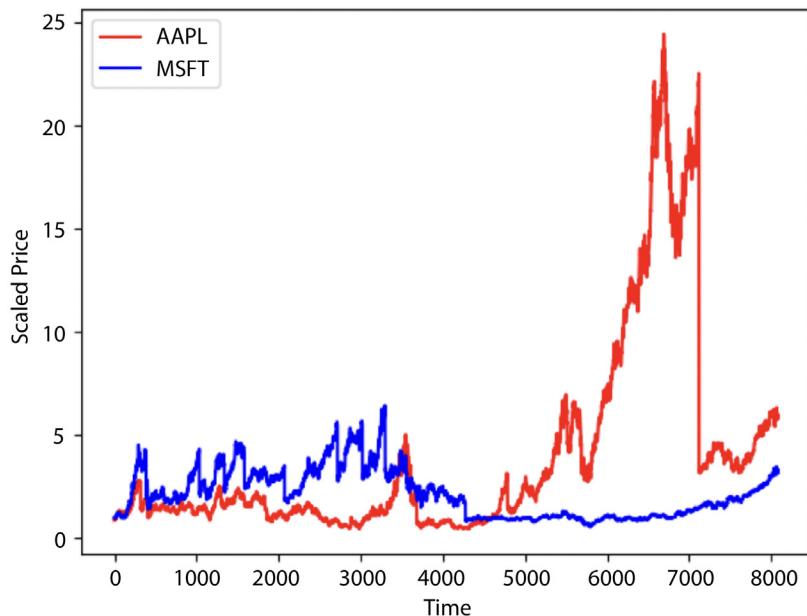


Figure 9.2 – Apple and Microsoft stock prices (scaled)

Note that Apple has performed better than Microsoft since 2,000, probably on the back of iPads and iPhones and the return of a certain Steve Jobs.

The large drop in the AAPL price around the 7,000 mark is due to a script issue by Apple to reduce the unit cost of their stock price, which corresponds to a reduction from 645.57 to 93.70 over the weekend of 2014-06-06.

**Memcache** is an alternative to Redis that's similar in operation but more limited in scope, and Redis Labs offers Memcached databases as well as Redis ones.

It provides only SET/GET functionality with a few associated routines such as append, prepend, incr, and decr, so it is much more lightweight than Redis. If simple key values are required, this may be the better option.

The package is a pure Julia client and implements all commands. It has been untouched for many years and details can be found at <https://github.com/tanmaykm/Memcache.jl>.

## LMDB

I mentioned earlier that Oracle's acquisition of MySQL led to the production of MariaDB. A similar occurrence was when Oracle bought Netscape via Sun Microsystems and finished with the ownership of **Berkeley Database (BDB)**. At the time, BDB formed the underlying database for the OpenLDAP project but work then proceeded to switch the backend to LMDB.

LMDB uses memory-mapped files, so it has the read performance of a pure in-memory database but still offers the persistence of standard disk-based databases and is only limited to the size of the virtual address space:

```
# Create a folder for LMDB database files
julia> mkdir("mydb")
```

Create the LMDB environment and open/create the LMDB files:

```
julia> using LMDB
julia> env = create();
julia> open(env, "mylmdb");
julia> set!(env, LMDB.MDB_NOSYNC)
```

Now, create a pointer to a transaction (`txn`) and open it:

```
julia> txn = start(env)
Transaction(Ptr{LMDB.LibLMDB.MDB_txn} @0x00007f9cab1fe140)
julia> dbi = open(txn)
DBI(0x00000001, "")
```

This transaction is used to write KV values, which then need to be committed:

```
julia> put!(txn, dbi, "malcolm", "malcolm@amisllp.com");
julia> put!(txn, dbi, "james", "james@amisllp.com");
julia> put!(txn, dbi, "barbara", "barbara@amisllp.com");
julia> commit(txn);
```

It is possible to list the current state of the LMDB environment:

```
julia> show(env)
Environment is opened
DB path: testdb
Size of the data memory map: 1048576
ID of the last used page: 2
ID of the last committed transaction: 1
Max reader slots in the environment: 126
Max reader slots used in the environment: 0
```

Now, let's delete the key for Barbara and export the remaining entries to a Julia dictionary hash:

```
julia> d = LMDBDict{String, String}("mydb");
julia> keys(d)
2-element Vector{String}:
 "james"
 "malcolm"
```

Finally, close the transaction and environment:

```
julia> close(env, dbi);
julia> close(env);
```

Note that the preceding code has created a couple of files called `data.mdb` and `lock.mdb` in the `mylmdb` subdirectory, which is where the dataset is stored.

## Document databases

Document-oriented databases are designed for storing, retrieving, and managing semi-structured data.

The difference between these and purely KV stores lies in the way the data is processed. In a KV store, the data is inherently opaque to the database, whereas a document-oriented system relies on the internal structure of the document to extract metadata that the database engine uses for further optimization.

One of the most popular document databases is MongoDB. We will look at the Julia interface for Mongo here.

MongoDB is supported in Julia by a package that is a wrapper around the C API. To use this package, it is necessary to have the Mongo C drivers installed and available on the library search path.

A record in MongoDB is a document, which is a data structure composed of field and value pairs and documents that are like JSON objects. The values of the fields may include other documents, arrays, and arrays of documents. Mongo itself is easy to install, and the server (*daemon*) process is run via the `mongod` executable.

The location of the database is configurable, but by default, it is in the `/data/db` folder, which must exist and be writeable.

There is a Mongo shell client program (`mongo`) that can be used to add and modify records, as well as used as a query engine:

```
$> mongo
MongoDB shell version: 2.6.7
> use test
switched to db test
> db.getCollectionNames()
[ "system.indexes" ]
```

If a collection does not exist, one can be created by inserting a record into it.

I have supplied a `quotes.js` file that comprises a set of insert statements:

```
db.quotes.insert({category:"Words of Wisdom", author:"Hofstadter's
Law", quote:"It always takes longer than you expect, even when you
take Hofstadter's Law into account"})
```

It is also possible to alias the `db.quotes` by (say) `qc = db.quotes` and write the insert statement using `qc.insert({...})` instead of `db.quotes.insert({...})`.

Loading all the *Quotes* data is straightforward on Linux/macOS – all you need to do is redirect input from the shell in the usual way and use the MongoDB client:

```
$> cd $HOME/MJ2/DataSources/Quotes
$> mongo < quotes.js
```

Because Mongo is schemaless, in cases where the author is unknown, the field is not specified and the category is provided in full as a text field and not as an ID.

The Mongo shell can also be used to retrieve records, like so:

```
$> qc.find({author:"Oscar Wilde"})
{"_id" : ObjectId("54db9741d22663335937885d"), "category": "Books &
Plays", "author": "Oscar Wilde", "quote" : "The only way to get rid of
temptation is to yield to it."}
```

The object that's returned is clearly in JSON syntax – more accurately, BSON. Now that our Mongo database has been loaded with a dataset, we can start to use Julia on it.

The package we will use is `Mongoc.jl`; it acts as an interface to the Mongo driver:

```
julia> import Mongoc
julia> const mc = Mongoc
# Create a connection to Mongo
julia> client = mc.Client
julia> db = client["test"];
julia> quotes = db["quotes"];
# Check we are connected and seeing the quotes collection
julia> length(quotes)
36
```

The previous query on Oscar Wilde is coded as follows:

```
# Note the need for triple quoting the BSON string# Escaping in single
#"s (i.e. \"") which work equally well
julia> mc.find_one(quotes, mc.BSON("""{"author" : "Oscar Wilde"}"""))
BSON("{"_id" : { "$oid" : "5c5f2e27a5f0227251bb66bc" }, "category" :
"Books & Plays", "author" : "Oscar Wilde", "quote" : "The only way to
get rid of temptation is to yield to it."}")
```

The **Create, Read, Update, and Delete** operations (**CRUD**) are all supported by `Mongoc`.

Previously, we saw the **Read** `find_one()` method, so here, we will concentrate on creating and deleting records.

Updating is possible using `update_one()` and `update_many()`. The package supports transactions, so a large update can be achieved by using a combination of a delete and an insert.

Start by setting up the BSON representation for a couple of quotes:

```
julia> using BSON
julia> doc1 = mc.BSON()
julia> doc1["author"] = "Orson Welles";
julia> doc1["quote"] = "If you want a happy ending, it all depends on
where you stop your story";
julia> doc1["category"] = "Words of Wisdom";
julia> doc2 = mc.BSON()
julia> doc2["author"] = "Bo Bennett";
julia> doc2["quote"] = "Visualization is daydreaming with a purpose";
julia> doc2["category"] = "Computing";
```

BSON docs can be created by passing a JSON string and/or a Julia `Dict` value as a parameter.

Alternatively, the BSON object can be converted back into these formats using `as_json()` and `as_dict()`, respectively:

```
julia> mc.as_dict(doc2)
Dict{Any,Any} with 3 entries:
"quote" => "Visualization is daydreaming with a purpose"
"author" => "Bo Bennett"
"category" => "Computing"
```

To insert a record, we can use the `push!()` routine:

```
julia> res1 = push!(quotes, doc1)
Mongoc.InsertOneResult(BSON("{ \"insertedCount\" : 1 }"),
"5c5f5869ebf7a80449023ab2")
julia> oid1 = res1.inserted_oid
"5c5f5869ebf7a80449023ab2"
julia> length(quotes)
37
```

To delete a record, first, we must create a selector. The simplest way to do this is to use the `oid` value since this is unique to each document:

```
julia> selr = mc.BSON();
julia> selr["_id"] = oid1;
julia> mc.delete_one(quotes, selr)
BSON("{ \"deletedCount\" : 1 }")
```

Using the `empty!()` method will clear out a collection (or database) but leave it in place. To remove it completely, use `destroy!()`.

We can terminate Mongoc by making a call to `destroy!(client)`.

It is possible to do multiple inserts by passing an array of docs (as BSON) to `append!()`:

```
julia> append!(quotes, [doc1, doc2])
Mongoc.BulkOperationResult(BSON("{ \"nInserted\" : 2, \"nMatched\" : 0,
\"nModified\" : 0, \"nRemoved\" : 0, \"nUpserted\" : 0, \"writeErrors\" : [ ]
}"), 0x00000001, Union{Nothing, String}["5c5f5a16ebf7a80449023ab3",
"5c5f5a16ebf7a80449023ab4"])
julia> length(quotes)
38
```

MongoDB also provides map-reduce operations to perform aggregation. These are supported by `Mongoc.jl` but are beyond the scope of our discussion here.

## Interfacing with REST

REST refers to a software architecture style that's designed to create scalable web services. It has gained widespread acceptance across the web as a simpler alternative to SOAP and WSDL-based web services.

RESTful systems typically communicate over hypertext transfer protocol with the same HTTP verbs (GET, POST, PUT, DELETE, and so on) used by web browsers to retrieve web pages and send data to remote servers.

With the prevalence of web servers, many systems now feature REST APIs and can return plain text or structured information. A typical example of plain text might be a time-of-day service, but structured information is more common for complex requests as it contains meta-information to identify the various fields.

Historically, this was returned as XML, but recently, JSON has become more popular since this is more compact and ideal for the web, where bandwidth may be limited. As with XML, which we looked at earlier, the JSON representation can be converted into an equivalent Julia hash array (`Dict`) expression.

To access them, we need a method to mimic the action of the web browser programmatically and to capture the returned response.

Earlier, we saw that this can be done using a task, such as `curl`, with the appropriate command line. We will see some further examples of this later in this section.

Alternatively, in Julia, we can use the `HTTP.jl` package in the same way we used it in the routine provided to get the stock datasets from Quandl. This is an example of a REST service. The aim here is to explore how this can be used as access to a database.

REST has become so prevalent that even when systems provide a more conventional API, they often implement REST as well, albeit this may be a little more restrictive.

### JSON/BSON formats

Web databases tend to use JSON as the format for storing and retrieving records rather than the more verbose XML style. JSON is supported in Julia via the `JSON.jl` package.

Here are some details of my company in JSON:

```
julia> company = """{  
"founder" : "Malcolm Sherrington", "company" : "AMIS Consulting LLP",  
"website" : "amisllp.com", "partners" : 5, "incorporated" : "1981-06-  
01", "areas_of_work" :  
["Aerospace", "Healthcare", "Finance", "Web Development"], "experience_  
years" : [31, 26, 15, 21],  
"programming_languages" :
```

```
[ "Fortran", "C", "Ada", "Perl", "PHP", "Julia", "Python", "R"]
} """
```

This can be parsed as follows:

```
julia> import JSON
julia> amis = JSON.parse(company)
Dict{String,Any} with 8 entries:
"areas_of_work"    => Any["Aerospace", "Healthcare", "Finance", "Web
Development"]
"partners"        => 5
"incorporated"     => "1981-06-01"
"founder"          => "Malcolm Sherrington"
"company"          => "AMIS Consulting LLP"
"website"          => "amisllp.com"
"programming_languages" =>
Any["Fortran", "C", "Ada", "Perl", "PHP", "Julia", "Python", "R"]
"experience_years" => Any[31, 26, 15, 21]
```

Parsing returns the details as a `Dict` value. Data types are inferred where possible but arrays are returned as `Any []`. Since JSON does not impose restrictions on mixed types, it may be necessary to cast or broadcast the type(s):

```
julia> using Dates
julia> Date(amis["incorporated"])
1981-06-01

julia> Integer.(amis["experience_years"])
4-element Array{Int64,1}:
31
26
15
21
```

**BSON** was introduced previously when we worked with MongoDB. It is the binary encoding of JSON-like documents that Mongo popularized for storing collections of documents. One of the reasons for this is that it adds support for data types such as date and binary, which are not covered by JSON.

BSON is a serialization format encoding format while JSON is in a human-readable standard file format and is designed so that it (normally) needs less space than JSON, but it is not extremely efficient.

It is implemented in such a way that it has a comparatively faster encoding and decoding technique.

Both `JSON.jl` and `BSON.jl` are supported by the JuliaIO group, whose work covers many of the packages we have covered already, including `FileIO.jl`, `JLD2.jl`, `HD5.jl`, and others.

## Web databases

Here are some examples of common systems that provide a REST API:

- **Riak:** This is an alternative KV datastore that operates more as a conventional on-disk database than the in-memory Redis system.
- **CouchDB:** This is an open source system that's now part of the Apache project. It uses JSON-style documents and a set of KV pairs for reference.
- **Couchbase:** This is a priority product that takes the CouchDB protocol and has both a REST API and a conventional one.
- **Neo4j:** This is a prominent example of a graphics database that stores nodes, parameters, and interconnect relationships between nodes in a sparse format rather than as conventional record structures.
- **BigQuery:** This is a RESTful web service that enables interactive analysis of massively large datasets working in conjunction with Google Storage.

In the following examples, we will look at the `HTTP.jl` package and see how we can use the less well-known one, **QuestDB (QDB)**, which is quite easy to install and run locally.

### HTTP

`HTTP.jl` provides both client and server functionality for the HTTP and WebSocket protocols. As a web client, it allows us to make a wide range of requests via GET/ POST, form data, multipart chunking, and cookie handling.

There is also advanced functionality to provide client-side middleware and generate your own customized HTTP client. On the server side, it provides the ability to listen, accept, and route HTTP requests, with middleware and handler interfaces to provide flexibility in processing responses, all of which will be discussed in *Chapter 11*.

The primary call is `HTTP.request`, which returns an HTTP response:

```
HTTP.request(method, url [, headers [, body]]; options)
```

The method must be GET, POST, PUT, HEAD, PATCH, or DELETE. For convenience, aliases for each are defined, as shown here:

```
HTTP.get( . . . ) => HTTP.request(GET, . . .)
```

The response that's returned comprises three parts, split from the stream, by the routine:

- **status:** The HTTP status code – this is normally 200 for OK but others are possible
- **header:** Additional information about the response returned as a `Dict` value, such as the server type, content length, and content type (very useful!)

- **body:** The actual content to be displayed by the browser or to be handled by the program; this is returned as a {UInt8} vector and normally will need to be changed to a string before being processed

As an example, we will get a copy of PacktPub's website home page:

```
julia> using HTTP
julia> url = "https://www.packtpub.com";
julia> r = HTTP.get(url, retry=false, readtimeout=30);
julia> r.status
200
```

A status code of 200 is the expected reply for OK.

The response also returned a series of headers:

```
julia> r.headers
12-element Vector{Pair{SubString{String}, SubString{String}}}:
  "Date" => "Fri, 20 Oct 2023 14:28:40 GMT"
  "Content-Type" => "text/html; charset=UTF-8"
  "Transfer-Encoding" => "chunked"
  "Connection" => "keep-alive"
  "Cache-Control" => "no-cache, private"
  "set-cookie" => "XSRF-TOKEN= . . . "
  "x-frame-options" => "SAMEORIGIN"
  "x-xss-protection" => "1; mode=block"
  "x-content-type-options" => "nosniff"
  "CF-Cache-Status" => "DYNAMIC"
  "Server" => "cloudflare"
  "CF-RAY" => "8191f2d76e25652b-LHR"
```

Of these, the most interesting is Content-Type as this indicates what type of document has been returned; others may indicate imagery, video, JSON coding, and so on.

The actual data is contained in r.body as a byte array that, if textual, needs to be converted into a (single) string and split into lines on \n:

```
julia> typeof(r.body)
Vector{UInt8} (alias for Array{UInt8, 1})
julia> length(String(r.body))
5431
```

An alternate way is via the `HTTP.open()` routine, which sends an HTTP request message and opens an I/O stream from which the response can be read and processed sequentially. This is required when we're receiving large data streams:

```
julia> HTTP.open("GET", url) do http
    i = 0
    while (!eof(http) && (i <= 8))
        i = i + 1
        s = read(http)
        (i > 1) && println(s)
    end
end
HTTP.Messages.Response:
"""
HTTP/1.1 200 OK
Date: Fri, 20 Oct 2023 14:50:37 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: no-cache, private
. . . .
CF-RAY: 819212ff0d8448c9-LHR
"""


```

This will be followed by the content body, which is comprised of 5,431 bytes.

## **QDB**

QDB is a columnar database that can be self-hosted. It is quite fun to use and requires less setup and loading than CouchDB.

To achieve speed, QDB does not support primary or foreign keys or any other constraints. The schema can be the same as other relational databases, except in the absence of a primary key, the user is responsible for the uniqueness of the records.

However, QDB-SQL can include `DISTINCT`. This can be used in the query string and ensures, at least, the absence of any duplicate results.

The open source version can be obtained from <https://questdb.io/get-questdb/>, which provides downloads for Windows, Linux, and FreeBSD plus a Docker image. For Apple/Mac, it can be installed more easily using homebrew.

Start by running `questdb start`. Usually, the server starts on the default port of 9000 and states from where the configuration file is read – for example, `/usr/local/var/questdb/conf/log.conf`.

There are also `status` and `stop` arguments, plus the command, without an argument, which gives us a strap helpline. When the server is running, `status` gives us a PID, and `stop` does exactly what might be imagined.

Using a web browser and going to `localhost:9000` displays a web page frontend, as shown in *Figure 9.3*. Load via SQL files in `$HOME/MJ2/DataSources/QDB`, *not* the `Quotes` folder, as this folder does not contain the primary key.

Also, it is possible to load data from a CSV file directly rather than using SQL DML statements via the `import` tab on the LHS sidebar; this method creates the appropriate table and sets up the field names from the CSV header.

Look at the topic in the QDB guide (<https://questdb.io/docs/guides/importing-data/>) for a detailed description of the various methods for loading datasets:

The screenshot shows the QuestDB web interface. On the left, there's a sidebar with a search icon, a 'Tables' section containing 'AAPL.csv', 'authors', 'categories', and 'quotes', and a 'Grid' tab selected. In the main area, a code editor shows a SQL query:

```

90 | select distinct a.autname as Author, q.quotext as Quote
91 |   from quotes q
92 |     join categories c on c.id = q.cid
93 |       join authors a on a.id = q.aid
94 |         where c.catname='Classics' order by a.autname;
95 |
96

```

Below the code editor is a table with two columns: 'Author' (string) and 'Quote' (string). The data is:

Author	Quote
Aristophanes	You cannot teach a crab to walk straight
Euripides	Whom, the Gods would destroy, they first make mad
Socrates	True knowledge exists in knowing that you know nothing

At the bottom, there's a log section with the message [15:35:14] ✓ 3 rows in 15ms Execute: 4.48ms Network: 10.52ms Total: 15ms Count: 0 Compile: 3.52ms Select distinct a.autname as Author, q.quotext as Quot... and a status bar indicating 'Connected' to 'QuestDB 7.3.2'.

Figure 9.3 – QDB web interface

The REST interface can be used via `curl` to query the data via the `/exp` or `/exec` commands; `/exp` outputs the results as CSV files while `/exec` outputs the results as JSON formatted strings:

```

$> curl -G --data-urlencode "query=select count(*) from quotes"
http://localhost:9000/exec
{"query":"select count(*) from quotes",
"columns": [{"name": "count", "type": "LONG"}], "timestamp": -1,
"dataset": [[36]], "count": 1}

```

Moving on to Julia, we need the `HTTP` and `JSON` packages. For this, it is convenient to work with the `/exec` style query.

Using imported spreadsheet-style (CSV) files has the advantage of not requiring extensive joins in the GET query string, after which they can use the returned dataset, manipulating it in the programming language.

For example, the following code uses the CSV file, which is a data dump from the three separate query tables using the query string:

```
select distinct a.autname as Author,
               c.catname as Category,
               q.quotext as Quotext
from quotes q
join authors a on a.id = q.aid
join categories c on c.id = q.cid
```

This is saved as `Quotes.csv` and imported into QDB.

The following Julia code selects the author name and quotes text from this file, sorted by author name:

```
julia> using HTTP, JSON
julia> uri = "http://localhost:9000/exec?query=";
julia> sql = "select distinct Author, Quotext from Quotes.csv order
by Author";

julia> query = uri * replace(sql, " " => "%20");
julia> r = HTTP.request("GET", query);
julia> r.status
200
```

`replace(sql, " " => "%20")` changes any spaces into their ASCII code in hex format (%20). This is equivalent to the `--data-urlencode` argument in the preceding `curl` statement.

If this is OK (that is, `r.status = 200`), then convert the return request body into a Julia Dict.

The first entry in the request header provided the count of the records returned from the query:

```
julia> d = JSON.parse(String(r.body))
Dict{String, Any} with 5 entries:
  "count"      => 31
  "query"      => "select distinct Author, Category, Quotext from
Quotes.csv order by Author"
  "timestamp"  => -1
  "columns"    => Any[Dict{String, Any}("name"=>"Author",
"type"=>"STRING"), Dict{String, Any}("name"=>"Category",
"type"=>"STRING"), Dict{String, Any...}
  "dataset"    => Any[Any["Adolf Hitler", "Politics", "The great mass
```

```
of the people will more easily fall victims to a big lie than a small
one"] , A ...
```

Now, use the dictionary to output all the quotes in the Classics category. Since `r.body` is in the form of a `UInt8` array, this needs to be converted into a string:

```
julia> for i in 1:length(d["dataset"])
    author = d["dataset"][i][1]
    category = d["dataset"][i][2]
    quotext = d["dataset"][i][3]
    if category == "Classics"
        @show author, quotext
    end
end
(author, quotext) = ("Aristophanes", "You cannot teach a crab to walk
straight")
(author, quotext) = ("Euripides", "Whom, the Gods would destroy, they
first make mad")
(author, quotext) = ("Socrates", "True knowledge exists in knowing
that you know nothing")
```

One use of QDB is that it can work with time series data.

As a second example, we will load and work with the Apple stocks file, `AAPL.csv`, and run the query to select the adjusted closing prices against the date, beginning at 01/01/2004:

```
julia> uri = "http://localhost:9000/exec?query=";
julia> sql = "select Date, AdjClose from AAPL.csv where
    Date > '2004-01-01' order by Date asc";
julia> query = uri*replace(sql, " " => "%20");
julia> r = HTTP.request("GET", query);
julia> r.status
200
```

QDB can display the output using the chart tab rather than grid. This can be done in Julia by parsing the JSON response as before and pushing `Date` and `AdjClose` onto arrays:

```
julia> X = String[];
julia> Y = Float64[];
julia> d = JSON.parse(String(r.body));
julia> for i in 1:length(d["dataset"])
    push!(X, d["dataset"][i][1])
    push!(Y, d["dataset"][i][2])
end
```

```
julia> using Plots; gr();
julia> plot(x, y)
```

Here's the output:



Figure 9.4 – Chart of Apple stocks from 01/01/2004

As we saw in *Chapter 7*, files in the form of tabulated data, (DLM), such as CSV and TSV formats, form a basis for datasets and in this chapter are used in loading databases.

What is lacking is a general approach to working directly with such files and loading, querying, and outputting the results without the need for a separate database. This is the aim of a set of packages termed the Queryverse (<https://www.queryverse.org>), which has been produced by David Anthoff and collaborators. Numerous video tutorials on it are available online. This is the theme of the final section of this chapter.

## (The) Queryverse

Julia's Queryverse provides a flexible way to work with data; it can be used by adding the “metapackage” Queryverse (see <https://github.com/queryverse>), which provides access to many of the component packages.

The Queryverse enables users to seamlessly import, transform, and analyze data through a user-friendly and expressive syntax.

The ecosystem includes packages such as `DataFrames.jl` for data storage and manipulation, `VegaLite.jl` for interactive data visualization, and `Query.jl` for efficient query capabilities. They can be used in combination to offer an intuitive environment for data exploration and analysis.

As we will see in the following examples, the Queryverse's syntax makes extensive use of macros and pipes (`|>`).

Data can be loaded/saved from and to a variety of data file types, including CSV and XLSX (Excel) formats, and also from packages such as `RDatasets.jl` and `VegaDatasets.jl`.

By using `DataFrames.jl` and `Query.jl`, it is possible to work with tabular data and perform complex data manipulations using both SQL-like and LINQ-style syntaxes.

## Querying the stocks dataset

We will use one of the datasets we are familiar with from earlier chapters – that of stock market share prices. However, this time, we'll take it from the `VegaDatasets` package:

```
julia> using VegaDatasets, IndexedTables  
julia> stocks = dataset("stocks") |> table
```

The Queryverse supports reading/writing to Excel format :

```
julia> using ExcelFiles  
julia> stocks > save("stocks.xlsx")
```

*The type of file to be created is determined by the filename extension.*

It consists of five separate company stock prices: Amazon (AMZN) Apple (AAPL), Google (GOOG), IBM, and Microsoft (MSFT), and ranges from the first days of each month from January 2000 until March 2010, except for Google, which only begins on August 2004.

The following query creates a DataFrame consisting of just Apple (AAPL) stock prices.

Note the style of the query: we are using macros such as `@filter`, `@map`, and `@orderby`, the results of each are passed to the next via the `|>` operator, and then they are piped to create the DataFrame:

```
julia> using IndexedTables, DataFrames, Query, Dates  
julia> dfa = stocks |>  
    @filter(_.symbol=="AAPL") |>  
    @map({date=Date(_.date,"u d Y"),_.price}) |>  
    @filter(_.date >= DateTime(2005)) |>  
    @orderby(_.date) |>  
    @drop(12) |> DataFrame  
51x2 DataFrame  
Row | date      price
```

```

1 | 2006-01-01    75.51
2 | 2006-02-01    68.49
3 | 2006-03-01    62.72
4 | 2006-04-01    70.39
.
.
.
50 | 2010-02-01   204.62
51 | 2010-03-01   223.02

```

In this example, the query *drops* the first 12 records – that is, 1 year. So, although the data is filtered on dates from 01/01/2005, the output only starts from 2006.

As with the QDB database previously, it is convenient to work with a `Dict` value:

```

julia> aapl = stocks |>
    @filter(_.symbol=="AAPL") |>
    @map(Date(_.date,"u d Y") => _.price) |> Dict

Dict{Date, Float64} with 123 entries:
Date("2009-06-01") => 142.43
Date("2001-10-01") => 8.78
Date("2002-12-01") => 7.16
.
.
```

In this case, the data is not ordered and contains the entire date range from 01/01/2000 to 01/03/2010.

We can select the values easily by looping through the `Dict` value:

```

julia> for yr in 2005:2010
    for mn in 1:12
        dd = Date(yr,mn,1)
        try
            println(dd, " : ", aapl[dd])
        catch(ie)
            break
        end
    end
end
2005-01-01 : 38.45
2005-02-01 : 44.86
2005-03-01 : 41.67
2005-04-01 : 36.06
.
.
.
2010-03-01 : 223.02

```

As we have seen before, Apple performed extremely well in this period, going from a price of \$38.45 to \$223.02 – hindsight is a wonderful thing!

To see how all the five stocks performed, we can run a `groupby` query:

```
julia> gp = stocks |>
    @groupby(_.symbol) |>
    @map({ky=key(_),
          ln=length(_),
          mi=minimum(_.price),
          mx=maximum(_.price),
          ro=maximum(_.price)/minimum(_.price)})
```

5x5 query result

ky	ln	mi	mx	ro
MSFT	123	15.81	43.22	2.73371
AMZN	123	5.97	135.91	22.7655
IBM	123	53.01	130.32	2.4584
GOOG	68	102.37	707.0	6.90632
AAPL	123	7.07	223.02	31.5446

This confirms that Apple was the best stock to buy over the period in question. Notice that the Google results are based on 68 values, while all the others are based on 123, reflecting that the Google pricing data started later than the other four stocks.

## LINQ queries

For Microsoft .NET enthusiasts, the Queryverse also supports **Language-Integrated Query (LINQ)** style queries.

LINQ statements more resemble SQL ones than the standard Queryverse format we looked at previously.

Here is a query that selects Apple and Microsoft processes from the `stocks` DataFrame and “collects” the results as a DataFrame:

```
julia> dq = @from s in stocks begin
    @where s.symbol=="AAPL" || s.symbol=="MSFT"
    @orderby Date(s.date,"u d Y")
    @select {s.date, s.price, stock=s.symbol}
    @collect DataFrame
end;
# Output the first 6 entries
julia> dq[1:6, :]
6x3 DataFrame
Row | date           price   stock
```

---

1	Jan 1 2000	39.81	MSFT
2	Jan 1 2000	25.94	AAPL
3	Feb 1 2000	36.35	MSFT
4	Feb 1 2000	28.66	AAPL
5	Mar 1 2000	43.22	MSFT
6	Mar 1 2000	33.95	AAPL

For an alternate query, to demonstrate using database table joins, we can turn back to the TSV files for the three tables that comprise the Quotes dataset:

```
julia> using CSV
julia> Qdir = ENV["HOME"] *"/MJ2/DataSources/Quotes")
julia> dfq = CSV.read("$Qdir/quotes.tsv",DataFrame)
julia> dfc = CSV.read("$Qdir/category.tsv",DataFrame)
julia> dfa = CSV.read("$Qdir/author.tsv",DataFrame)
julia> df = @from i in dfq begin
    @join j in dfc on i.cid equals j.id
    @join k in dfa on i.aid equals k.id
    @select {k.autname, j.catname, i.quotext}
    @collect DataFrame
end;
# Show entries 7 thru' 10
julia> df[7:10, :]
4x3 DataFrame
Row | autname      catname      quotext
-----|-----|-----|-----
1 | Heller`'s Law  Words of Wisdom  The first myth of management
is that it exists
2 | Anonymous      Computing      There are two ways to write
error-free programs, only the third one works
3 | Fingle`'s Creed Science      Science is true, don`t be
misled
4 | Anonymous      Words of Wisdom  Today is the tomorrow you
worried about yesterday
```

---

## Vega-Lite

Vega-Lite is an open source JavaScript library that's capable of creating interactive visualizations. It is built on top of the general-purpose visualization framework Vega.

In the previous chapter, we skipped the Julia implementation of `VegaLite.jl` as it is used as the main vehicle for creating graphic displays via the `@vlplot` macro.

We can generate a simple line plot based on the `dfa` DataFrame, which was the LINQ query that was run previously on the stocks dataset:

```
julia> using VegaLite  
julia> dfa |>  
    @vlplot(:line, x=:date, y=:price,  
            height=400, width=400,  
            title="Apple Stock Price")
```

This code creates the plot shown on the left-hand side of *Figure 9.5*. This is displayed in a web browser wrapper by a temporary HTML file.

However, if the command is continued by piping through `| > save ("AAPL.svg")`, this will create an SVG file on disk; using file extensions such as `png` or `eps` will create PNG or Postscript files, respectively. Naturally, much more complex scripts, and corresponding displays, are possible via the `@vlplot` macro.

The following filters out both Google and Microsoft data, leaving Amazon, Apple, and IBM, which are displayed as area plots. This can be seen on the right-hand side of *Figure 9.5*:

```
julia> dataset("stocks") |>  
    @vlplot(  
        mark=:area, width=300, height=140,  
        transform = [{filter="datum.symbol !== 'GOOG' &&  
                     datum.symbol !== 'MSFT'"}],  
        x = {"date:t", axis={title="Time", grid=false}},  
        y = {:price, axis={title="Price", grid=false}},  
        color ={:symbol, legend=nothing},  
        row = {:symbol, header={title="Symbol"}}  
    )
```

Here's the output:

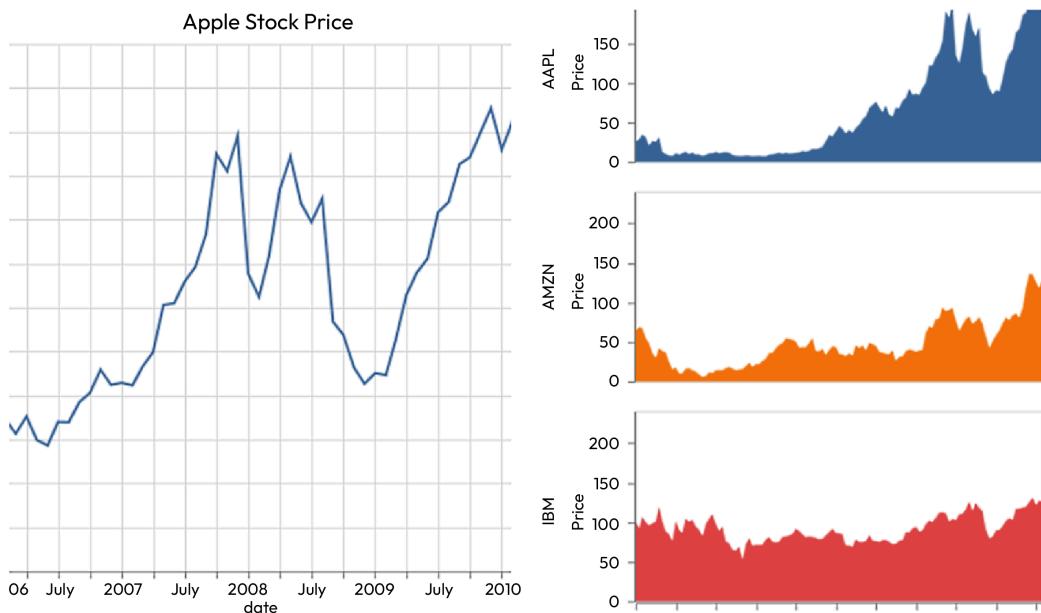


Figure 9.5 – Vega-Lite plots from the “stocks” dataset

The comments I made in the previous chapter in terms of dealing with the complexity of visualization frameworks such as Makie and StatsPlots can also be applied equally to Vega-Lite.

A comprehensive overview is provided by <https://github.com/queryverse/VegaLite.jl> and there are also several Julia workshops presented by David Anthoff at JuliaCon meetings.

## Summary

This chapter has looked at how Julia interacts with data held in databases and data files. Until recently, the majority of databases conformed to the relational model, the so-called SQL database. However, the rapid explosion in data volumes accompanying the big data revolution has led to the introduction of a range of databases based on other data models. These are normally grouped under an “umbrella” heading of NoSQL and are categorized as KV, document, and graphic databases.

We identified some examples in each category and identified Julia’s approaches specific to each case, and the appropriate packages and methods for loading, maintaining, and querying the different types of databases.

In the next chapter, we will discuss working with various networked systems and look at developing internet servers, as well as working with web sockets and distributed systems.



# 10

## Networks and Multitasking

In *Chapter 9*, we discussed interfacing with databases and RESTful access. While this can apply to data stored in a local environment, it is more often associated with data held on servers that's accessed over an internet or intranet connection.

In this chapter, we will develop the methods of working with network connections further, starting with the basic IP sockets and then services more commonly associated with the worldwide. We'll conclude by running tasks in parallel on multiprocessors over the network, as well as using distributed data manipulation and Dagger.

This chapter will cover the following topics:

- Sockets and servers
- Working with the web
- Parallel processing
- Distributed data sources

We'll start by considering the use of network sockets and how these are used in creating server processes.

### Sockets and servers

A socket is one endpoint of a two-way communication link between two programs running on the network. Normally, a server (*daemon*) process runs on a specific computer and has a socket that is bound to a specific port number; the server just waits, listening to the socket for a client to make a connection request.

A program running on the client side knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself with the server, so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

If the client and server are on the same machine, this is referred to as localhost and always has an IP address of 127.0.0.1 assigned to it.

If everything goes well, the server accepts the connection and upon acceptance, the server gets a new socket bound to the same local port and has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

## Well-known ports

The concept of “well-known” ports was introduced to Unix by the Berkeley development on the open source operating system in the early 1980s and has formed the bedrock of almost all computing operating systems ever since.

The idea is that a particular service is associated with a port number and that network packets are sent “tagged” with that port number. For example, originally, `f tp` used port 21, `s ssh` used port 22, and `s endmail` used port 25; other ports for these services may now be in use in addition to these.

One port that you will be familiar with is 80 as it is used by web servers to deliver HTTP content. Security firewalls often block specific traffic on specific IP ports but in general, transmissions on port 80 are allowable. Ports from 1023 and below are reserved and should only be used for the purposes they are intended for, but high port values are available to the general programmer. We saw a few examples of some SQL and NoSQL servers in the previous chapter.

For instance, the MySQL database commonly uses port 3306, the MongoDB server uses 28017, and the Redis KV store uses 6379. It is possible to start any of these servers on other ports, but the use is so widespread that the numbers that are assigned may be considered “semi-well-known” and are invariably used unless additional servers are to be run on the same platform.

A network service will be started to run in a loop, listening for traffic on a particular port. Its purpose is to act on any requests coming from that port. Usually, when these are dealt with, the service resumes in a listening state. This operation is usually referred to as a socket to a particular port. A client program sends a request to that socket and in most cases will establish a connection to facilitate bidirectional communication.

It is a primary function of the operating system that will deny the creation of additional sockets to avoid network clashes occurring, which would lead to ambiguities in servicing the requests. However, for services that must deal with a high volume of requests from different sources, such as web and database servers, it is not practicable to have a single resource dealing with each synchronously. Due to this, a sophisticated method of a server creating copies of itself, running in parallel, is often the norm.

## UDP and TCP sockets

The IP protocol specifies two types of sockets, unreliable and reliable. The concept of an unreliable socket may seem a little strange at first, but there are some requests that, if they are not serviced, can merely be ignored or retried.

An example of this may be requesting the network time from a time (NNTP) server. The establishment of a reliable socket necessitates a more complex procedure, but most situations require it. The unreliable sockets are termed as operating via the **User Datagram Protocol (UDP)** and are connectionless, while reliable sockets use the **Transmission Control Protocol (TCP)** and are connection-full. The latter are so common that often, they are merely referred to as sockets.

Julia supports both UDP and TCP sockets and **named** pipes; named pipes are available under the Linux/MacOS operation systems but are more useful when using Microsoft Windows.

The source code is mainly provided in the `socket.jl` and `streams.jl` modules, which are part of Julia's `stdlib`. This means they do not need to be installed; however, they must be referenced by a `using` statement.

We'll look at an example of a more extensive TCP server in the next section, so let's spend some time looking at an example of using UDP. UDP services are less common but have a much lower overhead:

```
julia> using Sockets
julia> s1 = UDPSocket();
julia> s2 = UDPSocket();
julia> (s1, s2)
(UDPSocket(init),UDPSocket(init))
```

Port numbers under 1024 are privileged but higher ones can be assigned by the user:

```
julia> bind(s1,ip"127.0.0.1",8011)
true
julia> bind(s2,ip"127.0.0.1",8012)
true
```

This creates two sockets to the localhost on ports 8011 and 8012. The operations are asynchronous, so they do not block.

For a UDP socket, the process is termed **binding** and the IP address of the localhost is defined by the special `ip"127.0.0.1"` string. Alternatively, the `IPv4(127,0,0,1)` function call is used. To use a socket in a daemon process, the success of the binding should be checked. A common reason for this to fail is that the port number is in use:

```
julia> using Dates
julia> send(s2,ip"127.0.0.1",8011,string(Dates.now()));
```

This statement sends the current date and time over the `s2` socket to the `s1` socket, with latter being bound to port 8011:

```
julia> (wip, msg) = recvfrom(s1)
julia> println(String(msg))
2023-11-09T23:47:37.179
julia> close(s2); close(s1);
```

The message can be read by the `recv()` or `recvfrom()` function, with the only difference being that the latter returns the IP address of the server that sent the message. It is a byte stream that needs to be converted into an ASCII string; in essence, this forms the basis for a simple NNTP time-of-day server.

## A “looking-glass world” echo server

In contrast to UDP services, which use the `send()` and `recv()` routines to send and receive messages, TCP services need to establish a duplex channel between the server and the client via `connect()`, `listen()`, and `accept()`.

There are several forms a TCP client connection can take. A typical one would call a routine such as `connect(host::ASCIIString, port::Integer)`.

If the host is not given, it is assumed to be `localhost` (`v"127.0.0.1"`), and the port is required.

An alternative to using the host string as the first argument is to pass a predefined `TCPsocket()`.

Connecting to a socket is simple and if successful, it will return an open status. It is then possible to send messages to the server over the socket and read the reply since it is a full duplex connection. Closing the socket will terminate the client and the server response but normally, the server will be written so that it can continue listening:

```
julia> sock = connect("mj2.website", 80)
TcpSocket(open, 0 bytes waiting)
julia> close(sock);
```

At the base of this functionality is the `getaddrinfo()` routine, which will do the appropriate address resolution:

```
julia> getaddrinfo("mj2.website")
ip"82.170.83.1"
```

Servers operating over TCP sockets use `listen()` and `accept()` to wait for connection. The “Hello World” example of a simple server accepts text inputs and simply sends them back to the client process. Here is my take on a “through the looking glass” version, which reverses the text before echoing it back.

---

We need to mung the returns by stripping off and re-appending so that they do not occur at the beginning of each line. The script is written using the bang-hash (# !) convention to indicate that it must be run via the Julia REPL. This will work on both Linux and OS X (BSD) operating systems once the script has been designed as executable using a `chmod u+x <script>` command; it needs to be in a folder on the executable path. For Windows, it is necessary to create a DOS batch file or Powershell script.

The `-q` switch is useful to suppress the Julia opening banner and `-depwarn=no` is also sensible.

The initial part of the script has more complex arguments than with the previous database ETL example, so we must make use of the `ArgParse` module to simplify the parsing.

This adds some overhead to the startup time but for a daemon server, this is scarcely of any great importance. We will return to this topic (*for utility scripts*) in the final chapter.

Alternatively, it can be run as `julia -q -depwarn=no echos.jl`.

The following code is the start of `script echos.jl` and is available in the code section accompanying this chapter. It should be run from the REPL rather than a Jupyter or Pluto notebook:

```
#! /usr/bin/env julia -q
#
using ArgParse, Sockets
const ECHO_PORT = 4000
const ECHO_HOST = "localhost"
function parse_commandline()
    s = ArgParseSettings()
    @add_arg_table s begin
        "--server", "-s"
        help = "hostname of the echo server"
        default = ECHO_HOST
        "--port", "-p"
        help = "port number running the service"
        arg_type = Int
        default = ECHO_PORT
        "--quiet", "-q"
        help = "run quietly, i.e. with no server output"
        action = :store_true
    end
    return parse_args(s)
end
pa = parse_commandline()
```

Each option is specified in a short (-) and a long (--) format.

The accompanying help strings enable `parse_commandline()` to create standard help text that can be displayed by using `-h` or `--help`:

```
$> echos.jl -h
usage: echos.jl [-s SERVER] [-p PORT] [-q] [-h]
optional arguments:
-s, --server SERVER hostname of the echo server (default: "localhost")
-p, --port PORT port number running the service (type: Int64, default:
4000)
-q, --quiet run quietly, i.e. with no server output
-h, --help show this help message and exit
```

The server defaults to `_localhost_` and will listen on port 4000.

It will echo responses to `stdout`. It is possible to provide additional information to log the information to disk; I'll leave that as an exercise for you.

Next, we assign some variables (as a convenience) that are passed in a `pa` hash from the `parse_command_line()` routine:

```
julia> ehost = pa["server"];
julia> eport = pa["port"];
julia> vflag = !pa["quiet"];
julia> pp = (ehost == "localhost" ? "" : "$ehost");
```

After processing the arguments and defaults, we enter the main processing loop. The server listens on `eport` and loops continuously, waiting for connection requests that are accepted. This processing is run asynchronously using the `@async` macro:

```
julia> vflag && println("Listening on port $eport")
julia> server = listen(eport)
julia> while true
    conn = accept(server)
    @async begin
        try
            while true
                s0 = readline(conn)
                s1 = chomp(s0)
                (length(chomp(s1)) > 0) && (s2 = reverse(s1))
                if s2 == "."
                    println("Done.")
                    close(conn)
                    exit(0)
                else
                    write(conn, string(pp, s2, "\r\n"))
                end
            end
        end
    end
end
```

```

    end
    end
  catch err
    println("Connection lost: $err")
    exit(1)
  end
end
end

```

Input is processed line by line rather than as a single complete string so that the order of a multiline is preserved.

To be certain of the integrity of each line, the trailing returns are stripped using `chomp()` and then re-appended when writing to the connection. This may not be necessary but we have no notion of how it will be coded by the client. Operating systems differ in the specification of end-of-line character(s).

Empty lines are ignored but a line consisting of a single `.` will shut down the server.

The server can be started as a background job, `_` (on Linux/macOS), by appending:

```
$> ./echos.jl &
Listening on port 4000
```

Testing is straightforward – open a connection to the port (3000) and create an asynchronous task to await any responses sent back over the socket:

```
julia> sock = connect(4000)
TcpSocket(open, 0 bytes waiting)
julia> @async while true
       write(stdout, readline(sock))
end
Task (waiting) @0x00007fc81d
```

Since this is a Looking Glass server, it seems appropriate to test this on some poetry from the book.

The following code reads the first verse of *You are Old Father William* and returns it in a manner that Alice would comprehend:

```
julia> const ALICE_DIR = ENV["HOME"] * "/MJ2/Alice/";
julia> fw =
  String(read(ALICE_DIR*"father-william-1v.txt"));
julia> println(sock,fw);
,dias nam gnuoy eht ",mailliW rehtaF ,dlo era uoY"
;etihw yrev emoceb sah riah ruoy dnA"
,daeh ruoy no dnats yltnassecni uoy tey dnA
"?thgir si ti ,ega ruoy ta ,kniht uoy oD
```

### **Named pipes**

A named pipe uses a “special” file as a communication channel. It is a **first-in, first-out (FIFO)** stream, which is an extension of the traditional pipe mechanism on Linux/OS X. It is also present on Windows, although the semantics differ substantially.

The example given here is specific to the Linux and macOS (OS X) operating systems when creating the FIFO stream (in the /tmp folder):

```
$> mkfifo /tmp/mj2_pipe
```

On the server side, we define a function that gets its input from the FIFO stream.

When a message is received, it can be processed and dispatched. In our case, to mimic the Looking Glass service, it can be reversed and logged by just writing it to the terminal:

```
julia> function rserver()
    open("/tmp/mj2_pipe", "r") do pipe
        while true
            sin = readline(pipe)
            if length(sin) > 0
                rsin = reverse(sin)
                println("recv> "*rsin)
            end
        end
    end
end
```

Now, the server is started asynchronously:

```
julia> @async rserver()
```

On the client side, all that is required is to open a channel to the FIFO stream and write a `strinf` call:

```
julia> function client(s::String)
    open("/tmp/mj2_pipe", "w") do pipe
        println(pipe, s)
    end
end
julia> client("You are old Father William.")
```

The output from the server is as follows:

```
recv> .mailliW rehtaF dlo era uoY
```

In the next section, we will leave the topic of sockets and servers and turn to working with web services, a subject we touched on briefly in the previous section when discussing querying databases using REST interfacing.

## Working with the web

The World Wide Web has become a popular medium for returning information.

Initially, this was limited to web browsers because of CGI or server-side scripting but recently, it has been extended to more generalized services due to the introduction of encoding schemes such as XML, JSON, and BSON.

In addition to browsers being almost universally available, we should note that most firewalls are configured to permit the traffic of HTML data. Also, programs such as `wget` and `curl` are available to query web servers in the absence of a browser.

In this section, we will consider Julia's approach and, in particular, the facilities available under the community group, JuliaWeb.

The main modules are supported by JuliaWeb and the roadmap currently aims to combine all older packages such as `HttpServer`, `HttpClient`, `Requests`, `WebSockets`, and more into one behemoth of a package we've met already: `HTTP.jl`.

The `HTTP` package aims to provide a complete suite of HTML and web-related functionality. Broadly, it encompasses the following:

- Parser interface
- Cookies
- Streaming interface
- Connection pooling
- AWS authentication
- Web sockets

The parser interface uses the `HTTP.Request()` routine to process/return the entire HTML entity, whereas the streaming interface uses `HTTP.Open()`.

The request routine has the following generic call:

```
HTTP.request(method,url[,headers[,body]] ; <key arguments>)
where the method is one of :
[ "GET" | "POST" | "PUT" | "DELETE" | "PATCH" ]
```

These methods are overloaded by individual calls, which leads to a slight simplification.

For example, the GET method can be written as follows:

```
HTTP.get( url [,headers [,body]] ; <keyword arguments> )
```

Other than the method, only the resource (URI) is a required parameter, but optionally, a request header and accompanying body may be provided.

Other parameters such as timeouts, retry count, redirect flags, and more may be passed and used by the service as necessary.

## HTTP methods

Standard HTTP methods are used in REST operations to map CRUD operations to HTTP requests. We covered the simplest form (GET) when we looked at queries of databases such as QuestDB in *Chapter 9*.

We encountered other uses of GET when downloading datasets.

Here, we will be working exclusively on executed GETs, but for completeness, the functionalities of the other methods are as follows:

- **GET:** Retrieves information. GET requests must be safe and idempotent, meaning regardless of how many times they are repeated with the same parameters, the results are the same – that is, we do not expect GET to have any side effects on the server.
- **POST:** Requests that the resource at the URI does something with the provided entity. POST is often used to create a new entity, but it can also be used to update an existing one.
- **PUT:** Stores an entity at a URI. PUT can create a new entity or update an existing one. A PUT request is idempotent. Idempotency is the main difference between the expectations of PUT versus a POST request.
- **PATCH:** Updates only the specified fields of an entity at a URI. A PATCH request is neither safe nor idempotent and therefore PATCH cannot ensure the entire resource has been updated.
- **DELETE:** Requests that a resource be removed. However, the resource does not have to be removed immediately; it could be an asynchronous or long-running request.

To look at GET in a little more detail, which typically involves interacting with a dynamic web page, as provided by PHP, ASP, or JSP server-side scripting, parameters are passed as a series of `key=value` statements preceded by `?` and separated by `&`. Because of the use of special characters (such as `?, &,` and others), the entire query string needs to be encoded.

The following is a GET request that's passing a couple of parameters, `q` and `me`. The response is sent back to the calling program.

I've coded a very simple PHP script called `showvars.php` to identify the variables and encapsulate them in the response body. This is available on the `mj2.website` server:

```
julia> using HTTP
julia> const SHOWVARS =
           "https://mj2.website/test/showvars.php"
julia> HTTP.get(SHOWVARS*"?q=OK&me=Malcolm%20Sherrington")
HTTP.Messages.Response:
"""
HTTP/1.1 200 OK
Date: Sat, 12 Nov 2023 11:29:28 GMT
Server: Apache
Upgrade: h2, h2c
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8
<p>GET variables<br/>
q = Hello<br />
me = Malcolm Sherrington<br />
</p>
"""
```

The PHP script is straightforward and can be executed on a local web server, such as one provided by XAMPP (Linux), MAMP (macOS), or WAMP (Windows), and is written as follows:

```
<?php
$ng = count($_GET);
$np = count($_POST);
if ($ng > 0) {
    echo "<p>GET variables<br/>\n";
    foreach ($_GET as $gkey => $gval)
        echo $gkey." = ".$gval."<br />\n";
    echo "</p>\n";
}
if ($np > 0) {
    echo "<p>POST variables<br/>\n";
    foreach ($_POST as $pkey => $pval)
        echo $pkey." = ".$pval."<br />\n";
    echo "</p>\n";
}
?>
```

This script identifies (and echoes) variables passed in either a GET or PUT request separately.

PUT usually arises from data submitted (that is, posted) via web forms, often because there is a large amount of text to be passed to the server and we do not wish this to be visible as part of the browser's query string.

In this case, it is necessary to pass a content type in the `headers` array and the query string as the body rather than part of the URL:

```
julia> HTTP.post(SHOWVARS,
  ["Content-type"=>"application/x-www-form-urlencoded"] ,
  "q=OK&me=Malcolm%20Sherrington")
```

Note that with PUT, it is possible to have both a query string *and* an encoded body, and `showvars.php` script will distinguish between each; try it! PHP does not dispatch on the method type, `$_GET` variables are those in the query string, and `$_PUT` variables are those in the request body regardless of the method. There is a third variable called `$_SERVER` that combines them as a single list.

An additional overloaded request is HEAD, which returns the response header. It is possible to convert the header into a `Dict` value from which we can then extract some useful additional information:

```
julia> h = HTTP.head("http://mj2.website/uc");
julia> fieldnames(typeof(h))
(:version, :status, :headers, :body, :request)
julia> hd = Dict(h.headers);
Dict{SubString{String},SubString{String}} with 6 entries:
"Content-Length" => "138"
"Last-Modified"   => "Fri, 03 Nov 2023 22:14:11 GMT"
>Date"           => "Sun, 12 Nov 2023 11:46:58 GMT"
"Content-Type"   => "text/html"
"Server"         => "Apache"
"Accept-Ranges"  => "bytes"
```

## Utility functions

Part of the HTTP package consists of a set of utility functions to help when working with web protocols. We will identify some of these here.

A web server is a little different functionally from the echo server we developed in the previous section. After some initial setup, it grabs a socket on which to “listen” and runs asynchronously, waiting for a connection that it then services.

What is different is that the web server needs to be able to return a variety of different file formats, and the browser must be able to distinguish them. The simplest are textual data in plain form as HTML markup, but there are also images (JPEG, PNG, GIFs, and so on) and attachments such as PDFs and Word documents.

So from the `Dict` value we created previously, we have the following:

```
julia> hd["Content-Type"]
"text/html"
```

A client (that is, a web browser) must know a series of formats to display the data returned correctly. These are termed **MIME types**. Here, the server sends metadata information as part of a response header by providing it as a `Content-Type` value.

The header precedes the data information, be it text or images, and in the case of binary information, it may be necessary to specify the exact length of the data stream using some additional metadata. This is provided via the `Content-Length`: key.

To signal the end of the header information, a web server will output a blank line, before sending the accompanying data.

We saw one example of a MIME type in practice when we posted the data to our little PHP script – that is, `application/x-www-form-urlencoded`. All MIME types follow the general scheme, comprising a group type (in this case, `application`) followed by a more specific type, `x-www-form-urlencoded`, separated by `/`.

Two common examples are `text/plain` and `text/HTML`, whose document formats are clear, but there are also MIME types to indicate PDF, XML, and an alternative group to cover the various imagery forms listed previously.

The HTTP module provides a routine to check that a URL is correctly formed and then, by reference to the web server, determine the MIME type:

```
julia> url = SHOWVARS
https://mj2.website/showvars.php
julia> HTTP.isvalid(url)
true
julia> HTTP.sniff(url)
"text/html; charset=utf-8"
```

Here, we can see that our `showvars.php` routine returns an HTML text file with an encoding scheme of UTF-8.

A URL leads to the concept of a **Uniform Resource Indicator (URI)**. This is a representation of the URL split into component entries such as scheme, host, port, and so on; the scheme doesn't have to be just HTTP or HTTPS and can include `mail`, `stmp`, `ws`, and more.

Now, let's create a URI for our PHP script and look at the names of the component fields and their values:

```
julia> uri =
    HTTP.URI(SHOWVARS*"?q=OK&me=Malcolm%20Sherrington");
julia> fieldnames(typeof(uri))
```

```
(:uri,:scheme,:userinfo,:host,:port,:path,:query,:fragment)

# Choose some of these to display
julia> @show (uri.scheme, uri.host, uri.port, uri.query);
(uri.scheme, uri.host, uri.port, uri.query) =
("https", "mj2.website", "/test/showvars.php",
 "q=OK&me=Malcolm%20Sherrington")
```

All the components are returned as strings, even the port (*which is always a positive integer*).

The port is *only* set because it was specified in the URL; otherwise, it would be returned as an empty string.

Likewise, an authentication string of the `username : password` form will be returned if present in the URL.

The path is the location following the host and beginning with /, excluding query parameters if present.

The query string is returned in the `name=var` style but it is possible to split this into a Dict value using the `queryparams()` routine:

```
julia> HTTP.queryparams(uri)
Dict{SubString{String},AbstractString} with 2 entries:
"me" => "Malcolm Sherrington"
"q" => "OK"
```

It is also possible to segment the path using `splitpath()`, which operations on the URL, not the URI – in this case, an array is returned and it includes the scheme and host as well as the location folders but *not* the final file.

We can find this by splitting the path on /. However, recall that a URL may have a default such as `index.html`, which may not be present:

```
julia> import HTTP.URIs: splitpath, escapeuri, unescapeuri
julia> splitpath(url)
5-element Array{String,1}:
"https:"
""
"mj2.website"
"test"
"showvars.php"
```

It was noted that URLs make special use of characters such as, :, /, ?, &, and also do not allow spaces in the string. The HTTP package has a routine to escape URLs that replaces special characters by their hex values preceded by % – that is, / is changed to %2F.

There is also a complementary function to unescape an encoded URL:

```
julia> eurl = escapeuri(url)
"https%3A%2F%2Fmj2.website%2Ftest%2Fshowvars.
php%3Fq%3DOK%26me%3DMalcolm%2520Sherrington"
julia> unescapeuri(eurl)
"https://mj2.website/test/showvars.php?q=OK&me=Malcolm%20Sherrington"
```

## TCP servers

The `showvars.php` script is running on the `mj2.website` server, which in addition to being capable of returning static HTML, image files and the like has a PHP scripting engine built in.

Earlier, when discussing databases, we identified some free packages called XAMPP (Linux), MAMP(macOS), and WAMP (Windows), which are all essentially the same: an Apache web server coupled with a MySQL database and PHP interpreter.

Apache is the most popular web server (it is free) and is a sophisticated example of a TCP server – that is, one that provides its services over TCP sockets.

Essentially, a web server is a little different functionally from the echo server we developed in the previous section. After some initial setup, it grabs a socket on which to `listen` and runs asynchronously, waiting for a connection that it then services.

The earliest versions of web servers merely supplied static text on request, which is relatively easy to do.

As an example, we are going to look at a service that gets a random quote (plus the author) from the SQLite Quotes database we created previously and echo it back.

### *A query server*

Our quotes server will use SQLite as a database, so we need to create a function to get a random quote, return it, and associate the author information.

This can be done by counting the number of entries in the quotes table and creating a random variable in the range 1 to the number of quotes since quote IDs are contiguous. However, this can be done as part of the SQL query, so we will use this method here.

In cases where the `author` field is not set, we will return `Anon` to indicate anonymous authorship.

We will not return to the matter of selecting category information. The extension is not too difficult – we just need to make a join on the categories table.

It is necessary to compute the length of the quotes table before selecting a random quote and it is convenient to pipe the results of the queries into DataFrames to extract appropriate information more easily.

First, point to the database and connect to it:

```
julia> using HTTP, SQLite, DataFrames
julia> DQ = ENV["HOME"] * "/MJ2/DataSources/Quotes";
julia> db = SQLite.DB("$DQ/quotes.db");
```

The SQLite connection is passed to a routine to get a random quote, which will return a tuple of the author's name and a corresponding quote:

```
julia> function getquote(db::SQLite.DB)
    sql = "select count(*) as nq from quotes";
    res = DBInterface.execute(db, sql) |> DataFrame
    nq = res[!, :nq][1];
    sql = """select a.autname, q.quotetext
        from quotes q
        join authors a on a.id = q.aid
        where q.id = (1 + abs(random() % $nq))"""
    res = DBInterface.execute(db, sql) |> DataFrame
    auth = res[!, :autname][1]
    if (auth == "missing")
        auth = "Anon"
    end
    quotetext = res[!, :quotetext][1]
    (auth, quotetext)
end
```

So, our `getquote()` function returns an author and a quote. We should test it before writing the server:

```
julia> (auth, quot) = getquote(db)
julia> @printf "%s => %s" auth quot
Thomas Huxley => The great tragedy of science: The slaying of a
beautiful theory by an ugly fact
```

Coding up the server is very simple. We will run it on port 8082 and use the `HTTP.Stream` interface to accept any requests on that port.

To answer any request, we must grab a random quote and return this as an HTML page – we need to set the status as OK (200) and the content type as `text/html` and then create some very simple HTML body text:

```
julia> HTTP.listen("localhost", 8082) do http::HTTP.Stream
    (auth, quot) = getquote(db)
    HTTP.setstatus(http, 200)
    HTTP.setheader(http, "Content-Type" => "text/html")
    HTTP.startwrite(http)
```

```
write(http, "<html><body>\n")
write(http, "$quot<br/>\n")
write(http, "<i>$auth</i><br/>\n")
write(http, "</body></html>\n")
end
```

This can be tested via a web browser on the local machine using `https://localhost:8082`.

## Routing

The query service we've presented was one of an app server, albeit one that performs a single function.

Nowadays, a web server can normally execute much richer functionality. One of the purposes of routing is to dispatch to different processes, as signaled by the URL, coupled with any accompanying parameters.

The example presented here is a web service that allows users to create and retrieve details of their pets:

```
julia> using HTTP, JSON3, StructTypes, Sockets
julia> mutable struct Pet
    id::Int
    userId::Base.UUID
    type::String
    breed::String
    name::String
    Pet() = new()
end
julia> StructTypes.StructType(::Type{Pet})=StructTypes.Mutable()
```

The `pet` struct is defined as mutable since adding an entry for a pet will create a new dummy version via the `Pet() = new()` statement and the attributes, such as `type`, `breed`, and `name`, added later in the routine. This allows an update routine (that is, one of the CRUD methods) to be implemented. Note that `delete` will not be included here.

As a repository for the pet data, a Julia `Dict` value is used. This means that the entries for pets will be lost when the service is shut down. For a more persistent repository, the data must be written to a data store, and that reread no startup of the server.

The following code implements the creation of the pet data and its retrieval from the `Dict` value:

```
julia> const PETS = Dict{Int, Pet}()
julia> const NEXT_ID = Ref(0)
julia> function getNextId()
    id = NEXT_ID[]
    NEXT_ID[] += 1
    return id
end
```

```

end
julia> function addPet(req::HTTP.Request)
    pet = JSON3.read(req.body, Pet)
    pet.id = getNextId()
    PETS[pet.id] = pet
    return HTTP.Response(200, JSON3.write(pet))
end
julia> function getPet(req::HTTP.Request)
    PetId = HTTP.getparams(req) ["id"]
    pet= PETS [parse(Int, PetId)]
    return HTTP.Response(200, JSON3.write(pet))
end

```

The routines need to return a 200 (OK) status value, along with the response body as a JSON string.

These are then registered as HTTP POST and GET methods in the server, which is started on a user-chosen port (that is, one greater than 1024):

```

julia> PET_SERVER = HTTP.Router();
julia> HTTP.register!(PET_SERVER, "POST", "/pet", addPet);
julia> HTTP.register!(PET_SERVER, "GET", "/pet/{id}", getPet);
julia> const PORT = 7878;
julia> server =
    HTTP.serve!(PET_SERVER, Sockets.localhost, PORT);

```

All we need to do now is add details of my cat Harry, who we will meet again in the final chapter of this book. We can do this using an `HTTP.post()` command:

```

julia> p0 = Pet();
julia> p0.type = "cat";
julia> p0.breed = "Domestic";
julia> p0.name = "Harry";
julia> resp = HTTP.post("http://localhost:$PORT/pet",
    [], JSON3.write(p0));
julia> resp.status
200

```

In a previous life, I owned a small sheep farm in the UK and had a Jack Russell terrier called Ben and a sheepdog called Jess – I will input those as well:

```

julia> p1 = Pet();
julia> p1.type = "dog";
julia> p1.breed = "Jack Russell";
julia> p1.name = "Benny";
julia> resp
= HTTP.post("http://localhost:$PORT/pet", [], JSON3.
write(p1));

```

```
julia> resp.status  
200  
  
julia> p2 = Pet();  
julia> p2.type = "dog";  
julia> p2.breed = "Border Collie";  
julia> p2.name = "Jessie";  
  
julia> resp = HTTP.post("http://localhost:$PORT/pet", [], JSON3.  
write(p2));  
julia> resp.status  
200
```

Ben was a particularly good ratter, totally bonkers and always getting stuck down rabbit holes, so let's give him the center stage in displaying his information:

```
julia> resp = HTTP.get("http://localhost:$PORT/pet/1")  
julia> pp = JSON3.read(resp.body, Pet)  
julia> println!("$(pp.name) is a $(pp.breed) of $(pp.type)")  
Benny is a Jack Russell breed of dog  
# . . . and close down the service explicitly.  
julia> close(server)  
[ Info: Server on 127.0.0.1:7878 closing]
```

## Mux

Web app servers are much more complex than what can be set up by just using `HTTP.route()` commands.

Mux is designed to fill this gap by providing the ability to define servers in terms of highly modular and composable components termed, **middleware**, as simply as possible – it uses HTTP methods to provide the underlying functionality.

You may be familiar with NodeJS, which is a popular middleware platform.

Mux comprises three core concepts:

- Dispatching
- Stacking
- Branching

Dispatching is equivalent to the request handlers we saw previously. It consists of a page routine with two arguments – the target URL and a `respond()` call that either texts or maps a `req` object to a function. If the URL is omitted, then the page default is assumed.

So, the simplest dispatch is `page(respond("Hi there, Blue Eyes"))`.

Stacking is analogous to pipelining – that is, writing a command that consists of a set of functions operating serially and outputting a response object. An example is `mux(munge, auth, check, app)`.

This will return a new app (`request -> response`), wrapped with three middlewares:

- `munge`: To make changes to the incoming request
- `auth`: To apply and authentication procedures
- `check`: To validate and possibly alter the outgoing response

Branching is interesting as it enables alternate routes to be determined when applied to the requested URL.

Let's look at a simple example that incorporates all of these features.

`Mux.defaults()` sets the host to `ip"0.0.0.0"` with a port address of 8000.

Notice that the app chain ends with `Mux.notfound()`, which proves the same function as the HTML page not found (404) response in the previous section:

```
julia> bye = """
Surely you are not leaving yet?
Don't call me Shirley!
""";
julia> using Mux
julia> @app muxtest = (
    Mux.defaults,
    page(respond("/", "<h1>We are off to catch the White Whale</h1>")),
    page("/about", respond("Call me,<b>Ismail</b>")),
    page("/user/:user",
        req -> "Hello, $(req[:params][:user])!",
        page(«/bye», respond(bye)),
        Mux.notfound())
julia> serve(muxtest)
```

`Mux` is capable of acquiring and serving static resources such as HTML pages, stylesheets, images, and more. Previous versions looked for these resources in a special folder called `/assets` but this has been changed so that it uses the `AssetRegistry` module:

```
julia> using AssetRegistry
julia> bacon = ENV["HOME"] * "/MJ2/DataSources/Files/bacon.html";
julia> assb = AssetRegistry.register(bacon)
"/assetserver/243f4cc0f7c3769dcb9b8da88895c744ceeb8bb5-bacon.html"
```

You will need to generate a different asset registration key (the page is included in the support files), and then use a URL in the form of `http://localhost:8000/assetserver/243f4cc0f7c3769dcb9b8da88895c744ceeb8bb5-bacon.html`:



## A Slice of Pork

Bacon ipsum dolor amet ea jerky nostrud, aliqua cow velit officia capicola short loin. Nostrud kielbasa pork chop venison elit chicken, filet mignon pariatur officia consequat kevin cupidatat. Elit labore chuck lorem alcatra proident commodo consectetur salami sausage fatback. Tail drumstick pariatur, capicola elit excepteur turducken. Ham alcatra pariatur kielbasa, lorem short loin elit. Fugiat hamburger duis id ipsum, in beef ribs. Ad shoulder filet mignon, fatback ball tip lorem hamburger shankle duis burgdoggen officia dolore.

Figure 10.1 – “Top” portion of the bacon.html file

Be aware that any other file(s), such as the banner image here, needs to be found/viewed by the browser.

This will apply to CSS stylesheets and Javascript libraries and it is good practice to provide a complete file specification to an online source.

## Web crawlers

We are all familiar with search engines such as Google, Bing, and others, which harvest links from websites. Their operation is quite simplistic. They start with the default page, search for (local) links on that page, and recursively retrieve and operate on them.

In practice, identifying links is not so easy. The page will be returned as a text source, usually HTML, so the syntax of the HTML schema must be understood, read, and parsed. This has become more standardized with the introduction of the **Document Object Model (DOM)** and the frameworks that operate on it. These can be client-side Javascript libraries or server-side ones written in PHP, Java, and similar.

In Julia, the corresponding package is called Gumbo, and the HTTP package is also required to grab the web pages:

```
julia> using HTTP, Gumbo
```

Define a function to get a page, check its status, and ensure that it has content:

```
julia> function fetchpage(url)
    response = HTTP.get(url)
    if response.status == 200 &&
        parse(Int, Dict(response.headers) ["Content-Length"]) > 0
        String(response.body)
    else
        <>>
    end
end
```

The OK status code is 200. If this is returned in the response struct and there is some zero-length context, it returns the body as a string. In theory, the MIME type of the page should also be checked.

For a full crawl, we will need to fetch the website's home page, which is then passed as a starter to a recursive procedure that follows these steps:

1. Look for `href` tags that start with / as this filters out the local anchors as they will begin with #.
2. Push each link onto the `links []` array.
3. Repeat recursively for any `child` elements.

Let's code this up as the `extractlinks` function:

```
julia> function extractlinks(elem, links)
    if isa(elem, HTML_ELEMENT) &&
        tag(elem) == :a &&
        in(«href», collect(keys(attrs(elem))))
    url = getattr(elem, "href")
    startswith(url, "/") &&
        length(url) > 1 && push!(links, url)
    end
    for child in children(elem)
        extractlinks(child, links)
    end
end
```

We will grab the default page of the `julialang.org` website and push all the links on an empty string array, using `Gumbo.parsehtml()` to set up a DOM for the page:

```
julia> content = fetchpage("https://julialang.org")
<!doctype html> <html lang=en > <meta charset=utf-8 >
<meta name=viewport content=\"width=device-width, initial-scale=1,
shrink-to-fit=no\">
<meta http-equiv=x-ua-
```

```
compatible content=\"ie=edge\">
<meta name=author content=\"Jeff
Bezonson, Stefan Karpinski, Viral Shah, ...
julia> jinx = String[]
julia> if !isempty(content)
    dom = Gumbo.parsehtml(content)
    extractlinks(dom.root, jinx)
end
```

We need to display the links that were found only once:

```
julia> display(unique(jinx))
28-element Vector{String}:
 "/downloads/"
 "/learning/"
 "/blog/"
 "/community/"
 "/contribute/"
 . .
 .
 .
 .
 "/community/#julia_user_and_developer_survey"
 "/shop/"
 "/contribute"
```

It is worth noting that this procedure is somewhat simplistic. For example, the `/contribute` page appears twice on the list, once because the first is terminated by `/` and the second because it is not.

Also, the `/community` page shows up twice, the second time due to the presence of the local anchor (`#`).

In the list (not shown previously), we have the following:

```
"/research/#publications"
"/research/#sponsors"
```

These also appear twice as they correspond to a local jump to a section down the research page.

Finally, we will leave the topic of web servers by mentioning a full-stack development framework, written entirely in Julia, called Genie.

## Genie

Genie is a behemoth of a package that implements a framework to develop web applications to develop backend and data apps. It can integrate with databases and provides production-ready web services and APIs, all written just in Julia. It was inspired by Django (from the Python community), which before that itself was a *child* of Ruby on Rails.

A useful place to start looking at Genie is <https://genieframework.com/> but there is also the excellent <https://www.freecodecamp.org/news/how-to-build-web-apps-in-julia/> from Logan Kilpatrick.

Here, we will introduce a couple of features of Genie, so you are encouraged to delve into the two links that we've just referenced.

First, here's a simple application of Genie to create a single web page that provides a few sayings from Humphrey Bogart:

```
julia> using Genie
julia> route("/looking") do
    "Here's looking at you, kid !!!"
end
[GET] /looking => #5 | :get_looking
```

Now, we'll bring up the service on port 8888 and retrieve the `looking` page:

```
julia> up(8888)
[ Info:
└ Web Server starting at http://127.0.0.1:8888
[ Info: Listening on: 127.0.0.1:8888, thread id: Genie.Server.
ServersCollection(Task (runnable) @0x00000001118295f0, nothing)
```

We can check the page in a browser by using the `http://localhost:8888` URL or the following code:

```
julia> using HTTP
julia> print((HTTP.get("http://localhost:8888/looking")))
[ Info: GET /looking 200
HTTP/1.1 200 OK
Content-Type: */*
Server: Genie/Julia/1.9.3
Transfer-Encoding: chunked
Here's looking at you, kid !!!
```

Notice that the content type is set as `*/*`. If viewed on a browser, it appears as vanilla text. If HTML is required, usually, a different render is used, and the output is piped through an HTML filter:

```
julia> using Genie.Renderer.Html
julia> route("/kissing") do
    h2("Just put your lips together and blow.") |> html
end
[GET] /pucker-up => #y | :get_kissing
```

Here, we can see that the content type is set to Content-Type: text/html; charset=utf-8, so it will be rendered appropriately by a browser. Although this works somewhat, it does not provide a complete web page and relies on the catholic nature of most browsers to render the page properly. We can see this by using curl to retrieve the page source:

```
shell> curl http://localhost:8888/kissing
[ Info: GET /kissing 200
<h2>Just put your lips together and blow.</h2>
```

Alternatively, when serving web requests, the response is usually in JSON and requires a different renderer. This time, we'll pipe the response body through the json filter:

```
julia> using Genie.Renderer.Json
julia> route("/playing") do
    json("Play it again, Sam") |> json
end
[GET] /playing => #9 | :get_playing
julia> resp = HTTP.get("http://localhost:8888/playing");
[ Info: GET /playing 200
julia> println(String(resp.body))
{"version": {"major": 1, "minor": 1}, "status": 200, "headers": [{"Content-Type": "application/json; charset=utf-8"}], "body": [34, 80, 108, 97, 121, 32, 105, 116, 32, 97, 103, 97, 105, 110, 44, 32, 83, 97, 109, 34], "request": null}
```

#### Note

We need to restart the web server before adding the extra routing information; otherwise, the server will return an error page response.

To form more complete websites, Genie may be utilized in its “Ruby-on-Rails” style mode by calling a newapp\_webservice() routine:

```
julia> Genie.Generator.newapp_webservice("BlueEyes")
[ Info: Done! New app created at /Users/malcolm/MJ2/Code10/BlueEyes
[ Info: Project.toml has been generated
[ Info: Installing app dependencies
  Activating project at `~/MJ2/Chp10/Code10/BlueEyes`
  Resolving package versions...
  Updating `~/MJ2/Chp10/Code10/BlueEyes/Project.toml`
[c43c736e] + Genie v5.23.1
  Updating `~/MJ2/Chp10/Code10/BlueEyes/Manifest.toml`
```

```

.
.
.
└ Info:
  └ Web Server starting at http://127.0.0.1:8000

```

This creates a bunch of files in a BlueEyes folder, among which are a `welcome.html` page and 404 (not found) and 500 (error) pages in the public subdirectory of the folder.

Also of interest to us is an embryonic `routes.jl` file, which sets up a route to the (default) welcome page:

```

using Genie.Router
route("/") do
    serve_static_file("welcome.html")
end

```

*Figure 10.2* shows the `welcome.html` page and the 404 response after requesting the `/about` page, which does not exist yet:

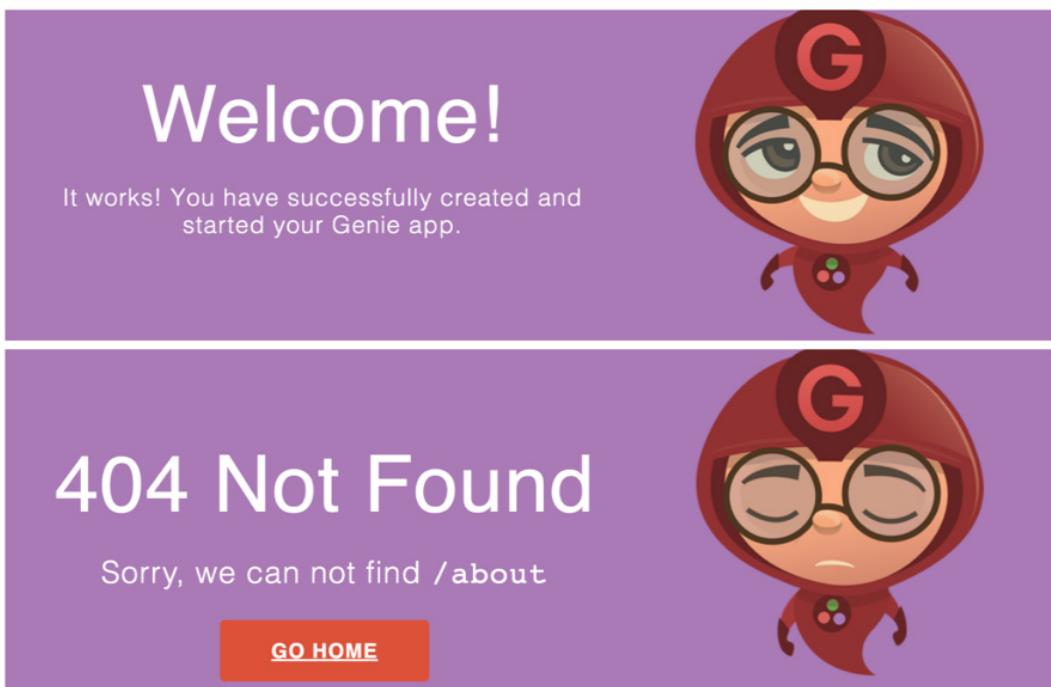


Figure 10.2 – Web app default welcome and Not Found pages

In Rails parlance, this provides the scaffold for replacing the welcome and error pages, imagery, style sheets, and more and creating additional pages.

One thing that's missing is the ability to interact with any backend databases. However, Genie can handle this too using the `newapp_mvc()` routine, which creates a scaffold for a feature-rich **model-view-controller (MVC)** application, including both REST API endpoints and user authentication:

```
julia> Genie.Generator.newapp_mvc("BlueEyes",
                                    dbadapter= :SQLite)
```

In this case, Genie will bootstrap a new application, creating the necessary files and installing dependencies for the database connection, including all the necessary support and configuration for Genie's ORM. The backends that are currently supported are SQLite, MySQL/MariaDB, and PostgreSQL, although the user can interface with other databases using the Silverlight ORM, a package (`Silverlight.jl`) that is also included in the Genie framework.

A final comment is that Genie was written by Adrian Salceanu and is discussed, among other things, in his excellent book *Julia Programming Projects*, published by Packt© 2018, which is well worth a read.

## Tasks and remote procedures

Julia has several packages that support parallel computing. These include modules for programming GPUs via CUDA and OpenCL, together with OpenMP, Spark, and MPI, which I will introduce briefly in the next chapter. With certain packages for tackling machine learning, such as Flex and Turing, processing on the GPU can also be done transparently in the presence of a GPU, otherwise defaulting to the CPU.

While Julia can be used with MPI, the natural model of communication in Julia is based on one-sided communication based on threads, tasks, futures, and channels.

The Julia distributed computing model is based on distributing tasks and arrays to a pool of workers. Workers are either Julia processes running on the current host or other hosts in your cluster.

### Tasks

In Julia, a task, also termed a co-routine, is a function that must be callable with no parameters. When first created, it is marked as runnable but is not executed until it's scheduled to:

```
julia> ta = Task(() ->
   let s = 0; sum(s += i for i in 1:100); println(s) end)
Task (runnable) @0x00000001c365210
```

This task sums the first 100 integers and is not executed until it is scheduled:

```
julia> istaskdone(ta)
false
julia> schedule(ta)
```

```
5050
Task (done) @0x000000011c365210
```

For something a little more interesting, let's look at generating the Fibonacci sequence using tasks:

```
# Define a Fibonacci function
julia> fib(n::Integer) = n < 3 ? 1 : fib(n-1) + fib(n-2);
# ... and create an asynchronous task
julia> chnl = Channel{Int}(1)
julia> m = 8
julia> @async for i=1:m
    put!(chnl, fib(i))
end
Task (runnable) @0x000000011bd79450
```

The task blocks until we `take!` the values from the channel, as follows:

```
julia> for i in 1:m
    println(i, " => ", take!(chnl))
end
1 => 1
2 => 1
3 => 2
4 => 3
5 => 5
6 => 8
7 => 13
8 => 21
julia> close(chnl);
```

Tasks are iterable, so we may need to break out to close the task:

```
julia> chnl = Channel{Int}(1)
julia> m = 20
julia> @async for i=1:m
    put!(chnl, fib(i))
end
Task (runnable) @0x000000011bd798d0
```

Now, we can take off the channel and bail out if `x` becomes large (say greater than 100):

```
julia> for p in chnl
    if p > 100
        break
    else
```

```
    println(p, " ")
    yield()
end
end
1 1 2 3 5 8 13 21 34 55 89
julia> close(chnl)
```

## Remote procedures

To execute in parallel, we will need to add some additional processors. This can be done as part of the `julia` command line by using the the `-n <nprocs>` switch, but otherwise, we can use `addprocs(n)`.

On a single machine, there is little to be gained in specifying more than the number of physical processors, which for an Intel x86 chip on my Mac is 4.

We will also need to reference the `Distributed` module, which is part of Julia's `stdlib`:

```
julia> using Distributed
julia> addprocs(4)      # Add 4 additional processors
```

As a first example, let's compute the value of  $\pi^2$  on an alternate processor. We can do this by scheduling a `remotecall` procedure and fetching the result:

```
# Run this on PID = 2
julia> r1 = remotecall(x -> x^2, 2, π)
Future(2, 1, 18, ReentrantLock(nothing, 0x000000, ...))
# ... and get the result
julia> fetch(r1)
9.869604401089358
```

Because this is such a common procedure, it can be done in a single call. This time, however, we will compute  $\pi/4$ . I'll leave it to you to multiply it by 4:

```
julia> remotecall_fetch(sin, 5, π/4)
0.7071067811865475
```

Now, we will compute some factorials on different processors: each one needs to “know” the `fac()` definition. We can ensure this by defining it as `@everywhere`:

```
julia> using Distributed
julia> addprocs(4)      # Only necessary if not done already
```

This is one of our simple recursive definitions of factorial and the `@everywhere` macro ensures that it is defined on each processor:

```
julia> @everywhere function fac(n::Integer)
    @assert n > 0
    return ((n == 1) ? 1 : n*fac(n-1))
end
```

Spawn the task over all the workers and calculate a different factorial on each worker:

```
julia> r = [@spawnat w fac(3 + 2*w) for w in workers()];
```

This set up a four-element vector, spawning the factorial on each processor for a value of  $3 + 2 \times$  the worker number – that is, values of 7, 9, 11, and 13:

```
# Return all the results
julia> for id in r
    @show id, fetch(id)
end
fetch(id) = 5040
fetch(id) = 362880
fetch(id) = 39916800
fetch(id) = 6227020800

# Just as a check ...
julia> @show(fac(7), fac(9), fac(11), fac(13))
(fac(7), fac(9), fac(11), fac(13)) =
(5040, 362880, 39916800, 6227020800)
```

## Needles and PI(ns)

This method of calculating pi is based on an experiment called Buffon's needle.

In the 18th century, Georges-Louis Leclerc, Comte de Buffon determined that you can approximate pi by dropping needles on a grid of parallel lines and calculating the probability that they will cross a line, which is related to the grid spacing and the length of the needles.

The probability is directly related to pi and gives an estimate by calculating twice the stick length divided by the number of sticks, all divided by the distance between lines multiplied by the number of sticks crossing lines:

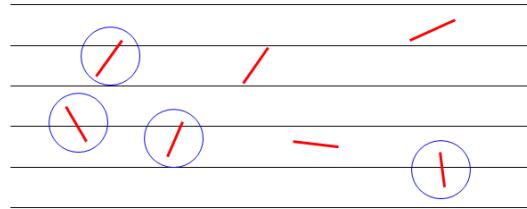


Figure 10.3 – How pins fall in the crossing lines in the grid

This estimate is usually not as good as the quadrant Monte-Carlo method we covered in *Chapter 1*, so more trials (that is, sticks) are needed.

We will execute the simulation both sequentially and in parallel to compare results and runtimes.

The first function is written in a sequential style:

```
# A sequential version
julia> function needles_seq(n::Integer)
    @assert n > 0
    k = 0
    for i = 1:n
        ρ = rand()
        φ = (rand() * π) - π / 2 # angle at which needle falls
        xr = ρ + cos(φ)/2 # x-location of needle
        xl = ρ - cos(φ)/2

        # Count times when needle crosses either x == 0 or x == 1?

        k += (xr >= 1 || xl <= 0) ? 1 : 0
    end
    m = n - k
    return (n / k * 2)
end;
```

The parallel version employs the distributed macro prefixing the for loop; otherwise, it is virtually the same code!

```
# A parallel version
julia> function needles_par(n)
    k = @distributed (+) for i = 1:n
        ρ = rand()
        φ = (rand() * π) - π / 2
        xr = ρ + cos(φ)/2
        xl = ρ - cos(φ)/2
```

```

    (xr >= 1 || xl <= 0) ? 1 : 0
end
m = n - k
return (n / k * 2)
end

```

First, run the two routines to compile the code. We will need less than 25% of the trials for the parallel version since it is running on four processors:

```

julia> needles_seq(1000)
3.0945741324921135
julia> needles_par(250)
3.114796238244514

```

Now, we can time both approaches, computing it in the REPL.

The numbers are very variable and we are only taking 1,000 random samples, so we will bump that up to  $10^8$ :

```

julia> @time needles_seq(100*10^6)
4.136637 seconds (1.95 k allocations: 117.152 KiB)
3.1417953013225968

julia> @time needles_par(25*10^6)
1.728617 seconds (311 allocations: 22.766 KiB)
3.1412370543338515

```

## Distributed arrays and Map-Reduce

Map-Reduce became famous as an approach offered by Google to deal with (very!) large volumes of data they were handling. The same was rapidly adopted by Amazon, Facebook, and the rest, giving rise to the age of “big data.”

The initial approach was to use Java and the JVM, with the introduction of Hadoop, followed by Spark, Apache’s take on the problem. This led to the rise of various NoSQL databases, which can be processed over disparate systems, some of which we met in *Chapter 9*.

Julia’s approach to distributed processing is quite different, not requiring the use of a framework (such as the JVM). It is made simple due to block code being injected by provisioning macros, almost transparently.

The Distributed Array package does what it says on the tin – that is, it always us to define, populate, and work on arrays on different virtual and/or physical machines.

Again, we will split it into four separate workers:

```
julia> using Distributed  
julia> addprocs(4);
```

The first macro we encounter is `@everywhere`, which ensures that all the necessary packages are visible on all the processors that have been added so far:

```
julia> @everywhere using DistributedArrays, Statistics
```

The functions for creating a distributed array are equivalent to those for conventional arrays, except that `d` is prefixed to the call.

So, they create a distributed array, filling it with random variates populated as a normal distribution with zero mean and unit standard deviation using `drand()` rather than `randn()`:

```
julia> da = drandn(10,10,10,10)  
10×10×10×10 DArray{Float64, 4, Array{Float64, 4}}:  
 . . .  
. . .  
julia> da[1][1][1][1] # Check it has been populated  
0.21022376288936018
```

In addition to the actual data values, the Distributed Array package has a set of associated meta-information that is used to act on the actual data in each particular instance:

```
julia> fieldnames(typeof(da))  
(:id, :dims, :pids, :indices, :cuts, :localpart, :release)
```

Let's pick some of the more interesting details:

```
# 'dims' corresponds to the dimensions of the darray  
julia> da.dims  
(10, 10, 10, 10)  
# 'pids' lists the workers process ids  
julia> da.pids  
1 × 1 × 2 × 2 Array{Int64,4}:  
[:, :, 1, 1] = 2  
[:, :, 2, 1] = 3  
[:, :, 1, 2] = 4  
[:, :, 2, 2] = 5  
# 'indices' shows how the data is distributed over the 4 workers  
julia> da.indices  
1×1×2×2 Array{NTuple{4, UnitRange{Int64}}, 4}:  
[:, :, 1, 1] = (1:10, 1:10, 1:5, 1:5)  
[:, :, 2, 1] = (1:10, 1:10, 6:10, 1:5)
```

```
[:, :, 1, 2] = (1:10, 1:10, 1:5, 6:10)
[:, :, 2, 2] = (1:10, 1:10, 6:10, 6:10)
```

A simple Map-Reduce to (*say*) calculate some statistics on the data can be achieved by spawning a function to operate on the `localpart` instances of each chunk separately, which are the actual values on that specific worker.

Then, we can fetch back each result to the originating host to accumulate and compute the final result:

```
julia> for ip in procs(da)
    rp = @spawnat ip mean(localpart(da))
    @show fetch(rp)
end
fetch(rp) = 0.013371161093273734
fetch(rp) = 0.025448070828582907
fetch(rp) = -0.009815370783141398
fetch(rp) = 0.017782139758342332
```

All of these are close to zero, which is what we expect since for a normal distribution, the variables have ( $\mu \sim 0.0$ ,  $\sigma \sim 1.0$ ).

To calculate this explicitly, we must spawn the mean and standard deviation tasks from the `Statistics` package (defined as `@everywhere`):

```
# Calculate the mean (~ 0.0)
julia> using Printf
julia> let
    μ = 0.0
    for ip in procs(da)
        rp = @spawnat ip mean(localpart(da))
        μ += fetch(rp)
    end
    @printf "\nMean = %.6f\n" μ/length(workers())
end
Mean = 0.013035

# ... and the standard deviation (~ 1.0)
julia> let
    σ = 0.0
    for ip in procs(da)
        rp = @spawnat ip std(localpart(da))
        σ += fetch(rp)
    end
    @printf "Standard Deviation = %.6f\n" σ/length(workers())
```

```
end
Standard Deviation = 1.007323
```

If we begin with a local array, this can be converted into a distributed one via the `distribute()` routine:

```
# Make sure remote processors have a copy of the packages
# I want to use a Bessel function, so have included SpecialFunctions
package here#
julia> @everywhere using Distributed, DistributedArrays,
          SpecialFunctions

#= The matrix 'aa' is created first and then split over our
processors. These are uniform variates in [0:1] here, so call is to
rand and not randn. =#
julia> aa = [rand() for i = 1:100,j = 1:100];
julia> da = distribute(aa);
# Again we will check da is a Distributed Array
julia> da.dims
(100, 100)
julia> da.indices
2x2 Matrix{Tuple{UnitRange{Int64}, UnitRange{Int64}}}:
 (1:50, 1:50)      (1:50, 51:100)
 (51:100, 1:50)    (51:100, 51:100)
```

We can check that the calculations are the same whether they're applied to the local array or the distributed one:

```
# This mapping works ...
julia> @elapsed bb = map(x -> abs(bessely0(x)), aa)
0.042339166
# ... and also so does this just a little more slowly!
julia> @elapsed db = map(x -> abs(bessely0(x)), da)
0.337633125
```

We can also check that any individual element has the same value for the same set of indices:

```
# Pick an element at random
julia> i = rand(1:100); j = rand(1:100); (i,j)
(37,5)
julia> bb[i,j] == db[i,j]
true
# . . . and that the overall sum of all the 10000 Bessel functions is
the same
julia> abs(sum(bb) - sum(db))
1.39871363 E-17
```

## Running on multiple machines

It is one of the surprising features of Julia that running tasks on a set of computers requires no additional coding than what has already been done; it is all dependent on how each instance of Julia is started up on individual machines.

Not only this but a combination of multiple computers and processors can be specified and the number of processors per machine may vary.

Julia can be started in parallel mode with the `-p` or `--machinefile` option:

- `-p <n>` will launch an additional *n* worker processes
- `--machinefile <mfile>` will launch a worker for each line in the *mfile* file

The machines defined in *mfile* must be accessible via a *passwordless* SSH login, with the Julia distribution installed on the remote workers at the same location as on the current host.

Each machine definition takes the form of `\ [count] [user@]host [:port] [bind_addr [:port]]`, where we have the following:

- `user` defaults to the current user
- `port` to the standard SSH port (22)
- `count` is the number of workers to spawn on the node (defaults to 1)

As you may have observed, the minimum information that needs to be specified is the name of each host (one per line).

The optional bind-to `bind_addr [:port]` specifies the IP address and port that other workers should use to connect to this worker.

## Distributed data sources

The JuliaData group includes a package called JuliaDB, which was heralded as a package for working with large

We will need a few packages that need to be available and can be defined in the `Project.toml` file for this chapter. They can be accessed in the usual way:

```
julia> import Pkg; Pkg.activate("..")
julia> using Distributions, StatsBase, OnlineStats
julia> using DataFrames, DTTables, Query, CSV, Printf
```

In the CSV folder of the `DataSources` directory are a couple of files referring to the statistics of the 2010 FIFA World Cup competition, one corresponding to the teams taking part and the other to players in individual teams. For our purposes, the latter data has been split into three separate CSV files (`P1`, `P2`, and `P3`). So, the first thing we need to do is define a vector listing these files:

```
julia> DS=ENV["HOME"] * "/MJ2/DataSources/CSV";
julia> players = ["$DS/P1.csv", "$DS/P2.csv", "$DS/P3.csv"]
3-element Vector{String}:
 "/Users/malcolm/MJ2/DataSources/CSV/P1.csv"
 "/Users/malcolm/MJ2/DataSources/CSV/P2.csv"
 "/Users/malcolm/MJ2/DataSources/CSV/P3.csv"
```

From this array, we can construct the distributed table for the players and fetch the data, converting it into a `DataFrame`:

```
julia> d = DTTable(CSV.File, players);
julia> tabletype!(d)
NamedTuple
julia> df = fetch(d);
julia> dp = DataFrame(df)
```

The resulting dataset has 593 rows and 8 columns comprising `surname`, `team`, `position`, `minutes (on field)`, `shots`, `passes`, `tackles`, and `saves`; naturally, the last of these only applies to goalkeepers.

To get more detailed statistics, we will need to link this to the `teams` file, which only contains 32 rows, so it is given as a single CSV file:

```
julia> dt = CSV.File("$DS/teams.csv") |> DataFrame
```

This has 10 columns: `team`, (FIFA) ranking, `games`, `wins`, `draws`, `losses`, `goalsFor`, `goalsAgainst`, `yellowCards`, and `redCards`.

The field that links the two datasets is `team`, which is a character string. Let's subset these `DataFrames` for the data we will be using and list the first couple of rows of each:

```
julia> dp1 = dp[:, [:surname, :team, :position]]; first(dp1, 2)
Row | surname    team      position
```

```

|
String31  String15  String15
-----
1 | Abdoun      Algeria    midfielder
2 | Belhadj     Algeria    defender
julia> dt1 = dt[:,[:team,:games,:wins]]; first(dt1,2)
Row | team      games   wins
| String15  Int64   Int64
-----
1 | Brazil      5       3
2 | Spain       6       5

```

First, we would like to look at the number of saves per goalkeeper and list those with 16 or more saves in the entire tournament. For this, we need an additional field from the `players` file, at which point we can compute the results using the `Query` package, which we covered in the previous chapter:

```

#= The actual query statement will also be constructed to do the
sorting in descending order =#
julia> dp2 = dp[:,[:surname,:team,:position, :saves]] |>
@filter(_.position == "goalkeeper" && _.saves >= 16) |>
@orderby_descending(_.saves) |> DataFrame;
julia> println(dp2[:,[:surname,:team,:saves]])
Row | surname      team      saves
| String31    String15  Int64
-----
1 | Neuer        Germany   20
2 | Kingson     Ghana     20
3 | Enyeama     Nigeria   20
4 | Ri Myong-Guk North Korea 19
5 | Kawashima    Japan     17
6 | Eduardo      Portugal  17
7 | Sorensen    Denmark   16
8 | Muslera     Uruguay   16

```

As we can see, the North Korean goalie worked hard, considering that the team did not progress beyond the group stage.

The other statistic I'd like to get is the success in scoring given the number of shots for each team. For the players, this requires a `groupby` computation to find the total number of shots:

```

julia> dp3 = dp |> @groupby(_.team) |>
@map({Team=key(_), Shots=sum(_.shots)}) |> DataFrame
julia> first(dp3,4)
4x2 DataFrame
Row | Team      Shots

```

	String15	Int64
1	Algeria	31
2	Argentina	69
3	Australia	29
4	Brazil	71

The number of goals scored is given as part of the `teams` data, so joining the preceding DataFrame with `dt`, which we created earlier, gives us the desired result:

```
julia> dj = dt |> @join(dp3, _.team, _.Team,
    {_.team, _.goalsFor, _.Shots} ) |>
    @orderby(_.team) |> DataFrame;
# Printing this out as a percentage success rate
julia> for r in eachrow(dj)
    print(r[:team], " => ",
        round(100*r[:goalsFor]/r[:Shots], digits=2), "%\n")
end
Algeria => 0.0%
Argentina => 14.49%
Australia => 10.34%
Brazil => 12.68%
. . .
. . .
Uruguay => 12.86%
```

Finally, let's repeat the calculation but just pick out four teams – England, Germany, North Korea, and the USA:

```
julia> pc(x,y) = round(100*x/y, digits=2
julia> count = 0
julia> for r in eachrow(dj)
    count += 1
    (count == 1) &&
        @printf "%10s %s\n" "Team" "Shot success"
    team = r[:team]
    if team in ["England", "Germany", "North Korea", "USA"]
        goals = r[:goalsFor]
        shots = r[:Shots]
        @printf("%12s %6.2f %%\n", team, pc(goals,shots))
    end
end
England    6.12 %
Germany   19.70 %
```

North Korea	3.12 %
USA	9.09 %

Oh, to be English – well, beaten by Germany in the World Cup again, although at least England fared a little better than the North Koreans; a small compensation!

## Summary

In this chapter, we dipped our toes into the networking and distributed capabilities of Julia. I didn't discuss its threading capabilities since this is purely experimental, but much effort is being directed to it and it should be a major feature in Julia v2.

We started with basic networking services available from UDP and TCP sockets and showed how a simple "Julia through the looking glass" echo sever could be written. Then, we moved on to web clients and servers prescribed by the `HTTP.jl` package. After, we discussed how, when working on the web, it is possible to implement socket-style systems using HTTP protocols.

The last section was somewhat different, in which we considered Julia's approach to executing tasks in parallel over distributed and multiprocessor systems and dealing with datasets distributed over multiple data files. There is one separate topic that must be considered and that is using Julia in the cloud; we will cover this in the final chapter.

This concludes our detailed discussions of various separate topics in Julia. The final chapter is a little different and deals with various aspects of programming development in this remarkable computing language.

# 11

## Julia's Back Pages

The first chapter of this book began with a broad overview of Julia, then we proceeded on to various aspects of the language mainly peculiar to itself, following on with a variety of themed chapters describing how to use Julia in analyzing, displaying, and processing data.

Naturally, various topics had to be omitted or did not fit into the general framework, and it is the purpose of this final chapter to discuss some of these.

Much of the material presented so far has applied to macOS, Linux, and Windows but has been oriented toward the former as the main development has been done on an Apple Mac. However, some of the topics in this chapter are a little different when using a Windows computer, so I will include a separate section to highlight the differences.

So, here's a main summary of the chapter:

- Configuring Julia and the operating system
- Standalone Julia
- Development tools
- Creating packages
- Whither goes Julia from here?

Up to now, the reader should be happy to work alone in the Julia coding in each individual chapter, applying to it their own idiosyncratic styles, but when it comes to contributing their work for the use of others, there are a few other factors that need to be taken into consideration, so that is the main thrust of this chapter.

In addition, we will finish up with a brief overview of some of the major topics in Julia that were too large and involved to be discussed in the detail they deserve and will suggest some avenues for further research.

## Configuring Julia and the OS

In the first chapter, we looked at installing Julia. For Windows and Apple, it is very easy: just a matter of downloading an executable and running it; this does, however, put the systems in a “known” place. Early versions of Linux also came with installations for Debian and Red Hat package managers but now this has been superseded by a (zipped) tar archive, which gives the user more latitude on where to install Julia.

### Getting Julia sources

Julia sources are downloaded from the following directory: <https://julialang.org/downloads>. Currently, there are distributions available for Apple (Silicon) and FreeBSD in this directory. It’s worth noting that FreeBSD is a Linux derivative and served as the precursor to OS X. Therefore, the discussion here is applicable to these platforms as well. You have the option to download a source distribution, not necessarily for the purpose of building it, but rather to explore the code and gain insights into examples of good coding practices. Be careful not to download the source from the Julia GitHub account unless you are seeking the current development edition; however, the repository from previous releases is there at <https://github.com/JuliaLang/julia/releases>.

In addition, the Julia website offers a link to the new(-ish) Julia version manager, `juliaup`, which I will mention at the end of this section.

When installing on macOS and Windows, the sources are (typically) placed in well-known locations, which on my computers (for the version I am using: 1.9.4) are:

- macOS: /Applications/Julia-1.9.app/Contents/Resources/Julia
- Win10: /Users/malcolm/AppData/Local/Programs/Julia-1.9.3

For Windows, I have installed it as non-shared, rather than declaring it as shared. Notice that the macOS installation is in terms of v“Main.Minor” style, whereas on Windows, the incremental version number is included, and when updated on macOS, the current minor version is preserved (for example, v1.9) and the previous one overwritten.

This is useful if you are making an alias to the binary (in the `bin` folder) and adding it to the command path. The installer will create a desktop shortcut, and it is possible to find where the executable is from the REPL by typing `Sys.BINDIR`.

The source tree has some folders of interest, including `base`, `contrib`, `src`, `stdlib`, and `test`, which can also be viewed from the Julia GitHub directory instead of downloading the source.

When first running Julia (of any flavor) it creates a “hidden” `.julia` subdirectory in the user’s home folder, which is very important and which we will be discussing next. I should add that *hidden* is quoted since on Windows, `.` files are hidden in plain sight; that is, not at all.

## The `.julia` subdirectory

It is worth discussing the `.julia` subdirectory in a little detail. First, it should be realized that deleting this will completely reset Julia, and a new folder is created from scratch when Julia is restarted, but it does not uninstall the Julia installation.

Here's the command for listing the folder:

```
$> ls .julia
artifacts
config
logs
registries
compiled
datadeps
packages
scratchspace
```

Some major subfolders are `registries`, `packages`, and `compiled`, which are used by the package manager when adding new packages. The `registries` folder has a compressed version of the general registry, and the `packages` folder does what it says on the tin; viz., it is the place where installed packages are stored.

As Julia is used, other folders may be created, including `artifacts`, `conda`, `makie`, `juliaup`, and possibly a few more that will make an appearance here as Julia is used.

The `.julia` folder, among other things, contains mainly a bunch of **Tom's Obvious Minimal Language (TOML)** files and coding for packages, which I will mention next.

### **TOML files**

The concept of a TOML file is quite similar to older **Yet Another Markup Language (YAML)** files, ones that have the ability to store key-value pairs in a tree-like hierarchy. An advantage of TOML over YAML is its readability, which becomes important when there are multiple nested levels, but it does have its downsides too.

The TOML specification is a minimalistic configuration file format that should be easy to read due to *obvious* semantics, hence the name. Since TOML files are created by Julia and not coded directly by the user, I will not discuss them in detail here; there are many references online, including the main one at <https://toml.io/> and Julia's own STDLIB documentation at <https://docs.julialang.org/en/v1/stdlib/TOML>.

For the markup required for Julia package management (that is, project and manifest files), the TOML format serves its purpose adequately.

### ***packages subfolder***

Packages when added via the package manager (`Pkg`) are stored in the `packages` subfolder folder beneath `.julia`.

One problem is that just like all the English are related to King Charles II, all packages seem to require a vast number of other packages as dependents; for example, installing the `Plots` package in a virgin (clean) system references over 100 packages – I have not got a zero too many here, but once these are in place then all is good (well, not really!).

Being an open system is a strength that Julia has compared to ones where modules are created by a few developers, but it comes with some baggage, not the least being that different packages may use different versions of dependent ones and are created at different times and by different authors.

So, the Julia system needs to be able to keep a handle on this, and we will see how this is done in the next section on Julia environments, but first, a quick mention of the `juliaup` version manager.

### ***juliaup***

`juliaup` is a package that advertises itself as a Julia installer and multiplexer. It comes from the David Anthoff stable, which we met when discussing the Queryverse, and is referenced by the Julia `downloads` package (<https://julialang.org/downloads/>) that has a link to its GitHub source, which naturally has quite extensive exposition there.

It is possible to use `juliaup` to install specific Julia versions, and it alerts users when new Julia versions are released, but also does this automatically if installed as such.

Naturally, to function correctly, it must be somewhat invasive and, depending on which operating system it is installed, involves possibly modifying the startup files and execution path; the advice is to ensure that the Julia binary is not found elsewhere before installing it.

For my purposes, I like to have several versions of Julia installed: the one I am working with, a previous one, perhaps a development one that has reached **release candidate (RC)**, status, and possibly a **long-term support (LTS)** version, these being on several different platforms and operating systems. In no sense am I a power user of `juliaup`, and the README on the GitHub page (<https://github.com/JuliaLang/juliaup>) does make impressive claims, so I urge the user to read it and try the system out if it seems attractive. Who knows? Perhaps I will become a convert after this book is finished.

## **Julia environments**

As mentioned previously, Julia wants to install many sources, and some of these may cause conflicts with previous versions when required as dependents on older packages. As such, the package management system requires a means to accommodate this.

As an example we look at GR, the graphics backend system often associated with the Plots package. A listing of the GR subfolder in the `.julia/packages` directory I am currently using gives the following output:

```
$> ls -ltr ~/.julia/packages/GR
drwxr-xr-x 13 malcolm staff 416 22 Jul 13:55 CMI3m
drwxr-xr-x 13 malcolm staff 416 15 Aug 21:52 jehu0
drwxr-xr-x 13 malcolm staff 416 25 Oct 15:41 yBe3g
drwxr-xr-x 13 malcolm staff 416 29 Oct 22:14 M9Dkl
drwxr-xr-x 12 malcolm staff 384 25 Nov 22:45 IVBgs
```

Those five strange alpha mnemonics are hashes created by Julia when versions of GR have been required, and the contents of each of these are all slightly different, so how are each of these versions used?

When a package is installed on Julia, it goes by default into a general-purpose manifest, stored by version number, and one that can quickly get out of hand:

```
$> pwd; ls -l
/Users/malcolm/.julia/environments/v1.9
total 360
-rw-r--r-- malcolm staff 173545 25 Nov 22:22 Manifest.toml
-rw-r--r-- malcolm staff    7836 25 Nov 22:22 Project.toml
```

This horror represents well over 800 packages, for which I apologize.

#### Note

The Pluto IDE also hijacks the `environments` folder as it “finds” packages running, without prompting; so, there may be manifests from Pluto here too.

With such a mess, conflicts are bound to occur, and the observant reader will have seen that in many examples, I have used this as the first command:

```
julia> import Pkg; Pkg.activate("..")
```

Or, you could use the alternative `activate .` command from the package manager subsystem; a “dot” is required in both cases here.

This has the effect of creating a new manifest, local to the directory you are working in, in which you then need to add the packages to be used, and any conflicts in dependencies should be resolved in that process.

It is possible to see the effect after activating a manifest using the `Pkg.status()` command, and the mess a general manifest may be in if `activate` is not used.

In some cases when using TOML files from an existing source, such as the ones distributed in the sources of this book, it is a good idea to use `Pkg.instantiate()` to resolve the dependencies; no “dot” this time.

## Startup configuration(s)

Armed with some knowledge of how Julia installation and package management works, it is useful to tailor your system to meet your individual requirements. This is one case that is operating system-dependent, and the discussion here refers mainly to Linux-style systems; that is, including macOS and FreeBSD.

In the `.julia` folder, there are a couple of additional subfolders I should mention, `logs` and `config`, although you may have to create the latter one. The `logs` folder contains some TOML usage files, together with one that is a history of REPL commands, (`repl_history.jl`) – ones that worked and ones that did not – and gets very lengthy unless deleted occasionally.

The history file is useful if you are trying to remember what you did in the past, but the layout is a bit annoying and needs some pruning to remove lines and pesky leading tabs.

I have piped it through the line editor used to create a more manageable version, the gobble-de-gook command being as follows:

```
$> cat repl_history.jl | sed '/^#\ time/d' | \
>   sed '/^#\ mode/d' | sed 's/^t//' > repl_history.txt
```

Into the `config` folder, you can place any commands to be executed, unsurprisingly, when Julia starts up. What you put into this is up to you but should not substantially slow Julia in its initialization process.

Here is a sample of mine that I have commented on just to aid with the dialogue, but one that is not so in practice:

```
#= Packages which I usually need and are in Base
  so add them here. =#
using Pkg, Printf, Random, REPLHistory

#= I use this to check for installed packages in my setup.jl, it is a
little more verbose since the haskey() function was deprecated =#
isinstalled(pkg::String) =
  any(x -> x.name == pkg && x.is_direct_dep,
      values(Pkg.dependencies()))

#= Julia has some pseudo Unix commands such as cd, pwd,
  mkdir, rm, but these are missing - these can, of course,
  be run in shell mode =#
ls(dir::String = ".") = run(`ls -l`)
ls(dir::String = ".") = run(`ls -l $dir`)
cat(fname::String)    = run(`cat $fname`)
```

```

more(fname::String) = run(`more $fname`)
# A little superfluous perhaps
pwup(x, p::Number) = x*p
pwup(x::Array) = x.*p
sq(x) = pwup(x, 2)
sqrt(x) = pwup(x, 0.5)
cb(x) = pwup(x, 3)
croot(x) = pwup(x, 1//3)
# Defined in the book, so I have added them here.
mad2(a,b,c) = a*b + c
systime() = ccall(:time, "libc"), Int32, ());
# A macro to activate and instantiate the current directory
macro PkgSetup()
    Pkg.activate(".")
    Pkg.instantiate()
end
# ... and one to execute an 'immediate' if statement.
macro iif(cond,doit)
    if (eval(cond)) doit end
end
# I miss this older statement, so defined it here.
function linspace(x0::Real, x1::Real, N::Integer)
    @assert (x0 < x1) && (N > 0)
    h = (x1 - x0) / N
    return collect(x0:h:x1)
end
#= Another way to activate folders, automatically when
Julia starts by defining a new environment variable
in the user's shell start up file =#
if haskey(ENV, "JULIA_ACTIVATE") &
    first(uppercase(ENV["JULIA_ACTIVATE"])) == 'Y'
    Pkg.activate(".")
    Pkg.instantiate()
end

```

Some of these are a little frivolous, but hopefully, you, dear reader, get the idea. It is also possible to include the functions in a separate file, but that seems to be an extra overhead. The main thing to avoid is adding anything that substantially increases what is euphemistically termed the time to first plot.

Turning quickly to the user shell, the configuration depends on which shell is being used, but is often `.profile` or `.bashrc` for the bash shell and `.zlogin` for the korn shell.

Regardless, what you may wish to do will be the same, usually defining environment variables and aliases, and amending the execution and (perhaps) library paths.

One alias I sometimes find useful when starting Julia is this:

```
juliet="julia -i -e 'import Pkg; Pkg.activate(\".\")'"
```

Notice that the double quote must be escaped as it is used twice in a different context.

This alias leaves the general start command (viz., `julia`) free to be used in cases where I do not wish to activate the current folder and can, in most cases, be used with other standard Julia switches (see the next section for these).

One last thing to mention is the `JULIA_DEPOT_PATH` environment variable that controls the global `DEPOT_PATH` variable, which in turn determines where the package management system looks for, among other things, package registries, configuration files, and the default location of the REPL's history file.

For a discussion of this and other special environment variables, look at the standard documentation at <https://docs.julialang.org/en/v1/manual/environment-variables>.

## Standalone Julia

We know that the Julia command recognizes a number of “switches” to modify its behavior; these can be listed by typing `julia -h` (or `--help`).

The dual nature of `-h` and `--help` is an example of a short and a long argument. All arguments have a long form, but only those deemed to be “common” also have a short one.

We met one such, `--procs`, in the previous chapter, which does have a shortened version, `-p`, the purpose of which is to define the number of processors rather than using the `nprocs()` routine in the code.

In the definition of the prior `juliet` alias, a couple of short forms were used (`-i` and `-e`), but `-interactive` and `-eval` could equally well have been chosen.

One form of start command I often use is `julia --banner=no --color=no`, since I have no need for a banner and do not like the color scheme in the REPL. It would also be possible to use `-q/-quiet` as a substitute for the `--banner` switch, but this has the additional effect of suppressing REPL warnings.

## Scripting

When it comes to running scripts from the command line, it is worth observing that any modules must be present in the general manifest as well as in the startup script; in fact, the startup file can be ignored using the `--startup-file=no` switch.

For modules that take a long time to load, it is possible to create system images, which we will discuss later.

---

How to use written scripts and handle command switches (that is, options) is analogous to the parameters passed by the Julia binary to itself, and we can identify three separate strategies:

- Handle switches as part of the native code
- Use a “heavyweight” package such as ArgParse to do some of the heavy lifting
- Use a “lightweight” package such as Getopt with less flexibility and correspondingly less overhead

I will consider all these in terms of a familiar piece of coding: the `fstop()` routine, which was developed in *Chapter 5, Interoperability*. To reprise this, it is a function to determine the ones that are of the greatest size from a specific directory and/or any associated subdirectories by executing the `find` system utility. The call was `fstop(dir=". ", nf=10)`, but with the script, we would like to pass the start directory (`dir`) and the number of files to list (`nf`).

Here, we will look at turning this into a runnable program on Linux/OS X.

Firstly, we need to specify the hash-bang (#!) header to find the Julia program and set any startup options. We wish to suppress the header banner and any warnings, especially dependence warnings.

The next code will first handle the options natively; that is, using method 1.

There are only three parameters to deal with:

- `-h`: To display a usage header
- `-d`: To specify the starting location; default: current directory
- `-n`: To set the (max) number of files to output

The usage is specified here by a simple string, although a multiline block of text could equally be used instead.

We pick up the number of arguments in the command line, and because of Julia’s now disapproval of global scalars, we will wrap the entire code in a `let/end` block so that we can set the defaults for the directory and file count and modify these if necessary, in a `while` loop:

```
#! /usr/bin/env julia --quiet --depwarn=no
#
const USAGE = "usage: $PROGRAM_FILE -h -d dir -n nfiles"
let
    nargs = size(ARGS) [1]
    dir = pwd()
    nf = 10
    hflag = false
    if nargs > 0
        i = 0
        while i < nargs
```

```

    i += 1
    s = ARGS[i]
    if s == "-h"
        hflag = true
    elseif s == "-d"
        (i < nargs) && begin
            i += 1
            dir = ARGS[i]
        end
    elseif s == "-n"
        (i < nargs) && begin
            i += 1
            nf = ARGS[i]
        end
    end
end
end
end
end

```

The preceding code should be reasonably clear. It just loops through the number of arguments and resets the directory and file count as required. The arguments are passed by Julia in the ARGS array.

#### Note

Unlike the Unix OS, the name of the scripts is *NOT* passed in this array; rather, the PROGRAM\_FILE scalar is set.

This coding is hardly foolproof, as there is no checking that -d is followed by a directory and that -n is followed by a positive integer, and we should set an exception block to mitigate these in this case the trade-off may be to use one of the modules (ArgParse or Getopt) mentioned previously.

To complete the code, we need to adapt slightly the `fstop()` routine written previously.

Once flags are set, all that remains is to create a `find` string and execute it in a pipeline in the same fashion as the routine in *Chapter 5*:

```

if hflag
    println(USAGE)
else
    efind =
        `find $dir -type f -iname "*" -exec du -sh "{}" + `
    println("Directory: $dir")
    try
        run(pipeline(efind, `sort -rh`, `head -$nf`))
    end
end

```

```
    println("Done.")
catch
    println("No files found.")
end
end
```

To run this under Linux/macOS, we now need to set the executable bit on the set by using the `chmod a+x ftop.jl` command.

### ***Using ArgParse***

One alternative to doing it ourselves is to use a package, and the de facto one is ArgParse. We met this module in the previous chapter where it was used in the looking-glass server, and we'll discuss its functionality now.

Here is how we configure ArgParse to handle the `ftop.jl` script:

```
using ArgParse
s = ArgParseSettings()
@add_arg_table s begin
    "-d", "--dir"
    help = "directory"
    arg_type = String
    default = "."
    "-n", "--nfiles"
    help = "number of files to display"
    arg_type = Int
    default = 10
end
pargs = parse_args(ARGS, s)
dir = pargs["d"]
nf = pargs["n"]
```

The basis of the setup is the argument table created by the `@add_arg_table` macro.

Each option is specified by a string, or a comma-separated list of strings, and a `help = "..."` parameter, type of argument expected (if any), and perhaps a default value.

It is also possible to specify a `required = true` parameter instead of the default value and also to deal with options without any accompanying arguments, in which cases `action = :store_true` is added so that these are remembered rather than a value set.

Notice that there is no `-h` or `-help` option specified as this is created automatically from the `help =` parameters that have been supplied for each individual option.

So, if we replace our previous naive argument handling with this code, there is now no need for the `h` flag in the previous version, and the remainder of the script reduces to this:

```
efind = `find $dir -type f -iname "*" -exec du -sh "{}" + `
println("Directory: $dir")
try
  run(pipeline(efind,`sort -rh`, `head -$nf`))
  println("Done.")
catch
  println("No files found.")
end
```

This code will *NOT* get executed when requesting help via `-h` or `--help`, as in this example:

```
$> ./ftop.jl -h
usage: ftop.jl [-d D] [-n N] [-h]
optional arguments:
-d D directory (default: ".")
-n N number of files to display (type:Int64, default: 10)
-h, --help show this help message and exit
```

Similarly, if an argument requires a value that is omitted, the code is not executed, and instead, an appropriate error response is displayed:

```
$> ./ftop.jl -d
option -d requires an argument
usage: ftop-parse.jl [-d D] [-n N]
```

### ***Getopt versus ArgParse***

While ArgParse is probably the more popular command-line parser, the setup of the argument table is a little involved, it is not a particularly arduous table, and the overhead at startup is not especially great.

This might be a little more of a problem when creating scripts to handle their input and output to operate in command-line pipes. Rather than using native coding (denoted by 1 in the next snippet), there is an alternative based on the standard `getopt` POSIX utility, and help can be obtained from the internet and/or via the standard man(ual) pages:

```
$> man getopt
GETOPT(1) BSD General Commands Manual GETOPT(1)
NAME
getopt -- parse command options
SYNOPSIS
args=`getopt optstring $*`; errcode=$?; set -- $args
```

```
DESCRIPTION
[ lots more text ... ]
```

The `Getopt.jl` Julia package was written by the enigmatically named “attractivechaos” and has not been touched for many years.

That said, it uses reasonable straightforward and works fine, as testing it via the package manager shows:

```
pkg> test Getopt
  Testing Getopt
  .
  .
  .
  Testing Running tests ...
Test Summary: | Pass  Total  Time
Getopt        |    5      5  0.0s
  Testing Getopt tests passed
```

We will be dissecting the construction of Julia packages in the next section, but this represents an easy one to peruse: <https://github.com/attractivechaos/Getopt.jl>.

Rather than looking at `fstop` again, I will look at the following script, `gopt.jl`:

```
#! /usr/bin/env julia
#
using Getopt
short_list = "ho:q"
long_list = ["help", "quiet", "output="]
println("\nSize of original ARGS array: ", size(ARGS))
println("Short list :: ", short_list)
println("Long list :: ", long_list, "\n")
for (opt, arg) in getopt(ARGS, short_list, long_list)
    @show (opt, arg)
end
println("Length of modified ARGS array: ", size(ARGS))
```

This consists of three options specified in both short- ("h,o:q") and long-list ([ "help", "quiet", "output=" ]) formats.

The thing we need to note is the use of `o :` and `output=` in the two forms respectively, both of which indicate that a value needs to be associated with this option. Unlike the `ArgParse` version, we do not know what its type is or the existence of any default value(s), so some additional work is needed after parsing the command line.

Running the script without any options produces the following output:

```
$> ./gopt.jl
Size of original ARGS array: (0,)
Short list :: ho:q
Long list :: ["help", "quiet", "output="]
Length of modified ARGS array: (0,)
```

There are no arguments, so clearly the ARGS array is of zero size and does not change.

This script run by using the `-o <file>` arguments produces the following output:

```
$> ./gopt.jl -o panto.jl
Size of original ARGS array: (2,)
Short list :: ho:q
Long list :: ["help", "quiet", "output="]
(opt, arg) = ("o", "panto.jl")
Length of modified ARGS array: (0,)
```

Here, the ARGS array has length 2, but this is stripped off as a tuple `(opt, arg)`, and `panto.jl` is actually a string and corresponds to a file we will encounter quite soon, but there is no indication of this.

## System images

One of the problems Julia must contend with is code loading times, as this involves compilation to native code via LLVM. This is especially obvious when running scripts that should execute quickly and if these result in graphic output.

The Julia command line is a `-J` (or `-sysimage`) switch with an accompanying system image file, one which is to be used at startup. This is appropriate whether executing a script or beginning a REPL session.

Creating a system image is quite simple; it uses a call in the `PackageCompiler` package, which unsurprisingly is `create_sysimage()` – no prizes for guessing this one!

```
julia> using PackageCompiler
julia> import Pkg; Pkg.activate(".")
julia> Pkg.add("Gadfly");
julia> Pkg.add("RDatasets")
julia> create_sysimage([:Gadfly, :RDatasets];
                     sysimage_path="sys_Gadfly.dylib")
```

The system image conventionally will have a `.so` extension when running under Linux, `.dylib` under macOS (as here), and `.dll` when using Windows

Restart Julia referencing this file following a `-J` (or `-sysimage`) switch:

```
$> julia -J sys_Gadfly.dylib
```

The `using` statement, which often takes a long time to load, is now immediate; I added `Rdatasets` too, but actually, this package is quite quick in loading, so not really necessary

```
julia> using Gadfly, RDatasets
```

The next example is the well-known plot of Fisher's Iris dataset, described in detail at [https://en.wikipedia.org/wiki/Iris\\_flower\\_data\\_set](https://en.wikipedia.org/wiki/Iris_flower_data_set).

To create a plot, we are required to push the Gadfly backend into the =# OS memory stack, whereas other formats such as `.png`, `.jpeg`, and `.df` are possible:

```
julia> pushdisplay(Gadfly.GadflyDisplay());
# Now retrieve a datafram from RDatasets
julia> iris = dataset("datasets", "iris"); first(iris,4)
4x5 DataFrame
Row | SepalLength  SepalWidth  PetalLength  PetalWidth  Species
_____
1 |      5.1       3.5        1.4         0.2       setosa
2 |      4.9       3.0        1.4         0.2       setosa
3 |      4.7       3.2        1.3         0.2       setosa
4 |      4.6       3.1        1.5         0.2       setosa
# Create the plot, but we need to call display() to see it.
julia> d = plot(iris, x=:SepalLength, y=:SepalWidth,
                  color=:Species, Geom.point);
julia> display(d)
```

*Figure 11.1* shows the resulting display, which when using the REPL will pop up in a browser window; otherwise, with an IDE in a separate panel:

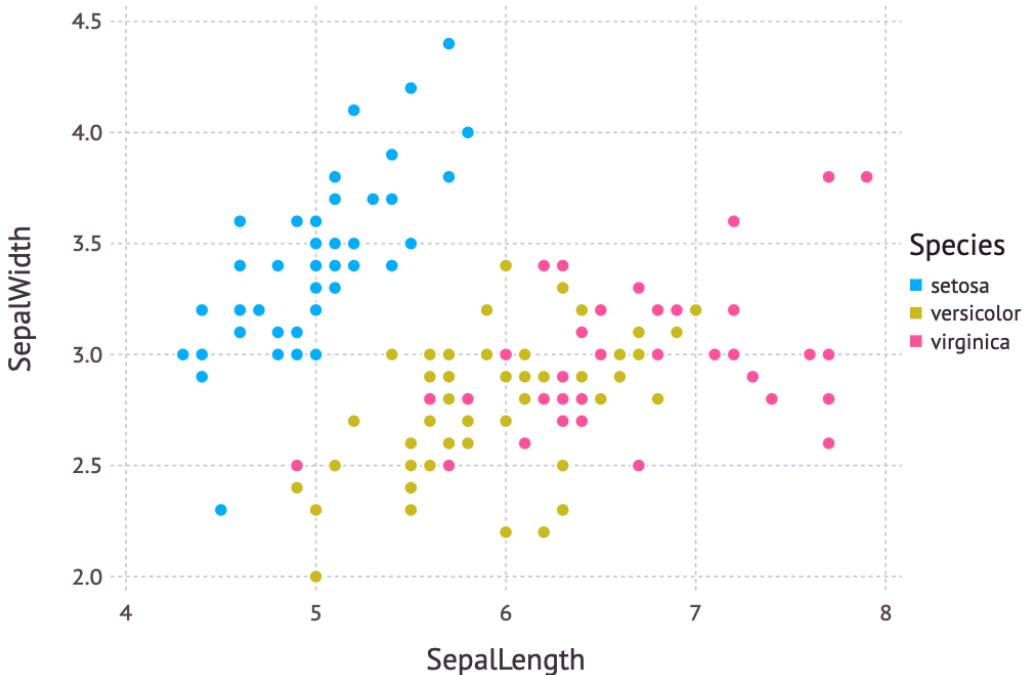


Figure 11.1 – The display of the Iris dataset using Gadfly

It is also possible to create a disk file by piping the plot to the `SVG()` function, which we saw in *Chapter 8* is supported directly by Gadfly:

```
julia> d |> SVG("iris_plot.svg", 6inch, 4inch);
```

Recall that other formats such as `.png`, `.jpeg`, and `.pdf` are also possible but need a little more work.

## Development tools

In this section, I want to introduce some factors that should be considered when creating code for others to use and possibly contribute to, which will not necessarily be as strictly adhered to here as we are doing so for our own usage.

These cover topics such as documentation, debugging, revision, and profiling, together with some degree of standardization, which I will tackle here, but not necessarily in that order. For an actual example of creating a package, that will be deferred until the following section.

## Document strings

It is pantomime season in the UK at present, and I with my family go every year to the current offering in Wimbledon. All children over the age of 2 love it, although hopefully they don't get all the jokes, and it gives the adults (who mostly do) a chance to go as well.

So, for some of the rest of this chapter, I am using a file introduced in the preceding  `getopt.jl` example, `panto.jl`, with three pantomime characters: Aladdin, the Genie, and Ali Baba (plus his 40 thieves), but now put to some better use:

```
$> cat panto.jl
"""
Calculate the sum of the series i/(i+1)^2 using the genie function for
an integer i in the range [1:n].
"""

function aladdin(n::Integer)
    @assert n > 0
    s = 0.0
    for i in 1:n
        s += genie(i,2)
    end
    return s
end
"""
Compute the value of the expression x/(x+1)^k where x is a numeric and
k is a (non-complex) number.
"""

genie(x,k) = x/(x+1)^k
const N_THIEVES = 40; # Ali Baba has 40 thieves in the fairy story.
"""

Compute and store the items using Aladdin's genie storing each partial
sum of the series i/(i+1)^2 in an array upto a value of 40.
So the preferable count to choose is a multiple of 40 in order to
capture the final value sum of the series in the array.
"""

function alibaba(n::Integer)
    @assert n >= N_THIEVES
    s = 0.0
    k = n/N_THIEVES
#= We could define a fixed array but for only 41 items in
Ali Baba band has 40 thieves, so push values instead =#
    a = Float64[]
    push!(a,s)
    for i in 1:n
        s += genie(i,2)
```

```

        (mod(i, k) == 0) && push!(a,s)
    end
    return a
end
function panto()
    helptxt = "Help is available on each of the individual pantomime
characters: alibabi, aladdin, genie."
    println(helptxt);
end

```

Including this in the REPL adds not only the function code but help text as well. This is because any string immediately preceding a function, with no intervening blank line, is interpreted as help text (referred to as a document string) and inserted into Julia's help system:

```

$> julia -q
julia> include("panto.jl")
panto
julia> panto()
Help is available on each of the individual pantomime characters:
alibabi, aladdin, genie
help?> alibaba
search: alibaba
Compute and store the items using Aladdin's genie storing
each partial sum of the series  $i/(i+1)^2$  in an array upto
a value of 40.
The preferable count to choose is a multiple of 40 in
order to capture the final value sum of the series in the
array.

```

Appropriate text for all routines in a module should be supplied, and in these cases, state the calling parameters with defaults if those exist, any optional ones together with what function the routines perform, and also what value(s) are returned.

Compute the minimum series sum, up to 40, and the array size should be 41, as we must include Ali Baba himself plus all the thieves:

```

julia> y = alibaba(40);
julia> size(y)
(41,)
julia> round(y[end], digits=6)
2.682094

```

Running the function for a larger count ( $10^6$ ) by using `BenchmarkTools` in order to aggregate some samples, we find the following:

```
julia> using BenchmarkTools
julia> @benchmark alibaba(10^6)
BenchmarkTools.Trial: 288 samples with 1 evaluation.
Range (min ... max): 17.066 ms ... 18.784 ms
Time (median): 17.159 ms
Time (mean ± σ): 17.247 ms ± 247.697 μs
███████████ 17.1 ms Histogram: log(frequency) by time 18.4 ms <
Memory estimate: 480 bytes, allocs estimate: 3.
```

Computing the first 4 values in the final limit to 40, 400, 4000, and 40000 gives us this:

```
julia> for i in 1:4
    k = 4*10^i
    X = alibaba(k)
    @show (k, round(X[end], digits=5))
end
(k, X[end]) = (40, 2.68209)
(k, X[end]) = (400, 4.92998)
(k, X[end]) = (4000, 7.22696)
(k, X[end]) = (40000, 9.52898)
(k, X[end]) = (40000, 11.83151)
(k, X[end]) = (400000, 14.13409)
(k, X[end]) = (4000000, 16.43667)
(k, X[end]) = (40000000, 18.73925)
```

*Figure 11.2* shows the first curve for the limit of 40.

The plot has a familiar look, and if we correlate the partial sums, `X [end]`, against the log of the count, `log(k)`, then these correlate with a value of 0.99999567, so the series is generating a logarithmic function and will continue without limit:

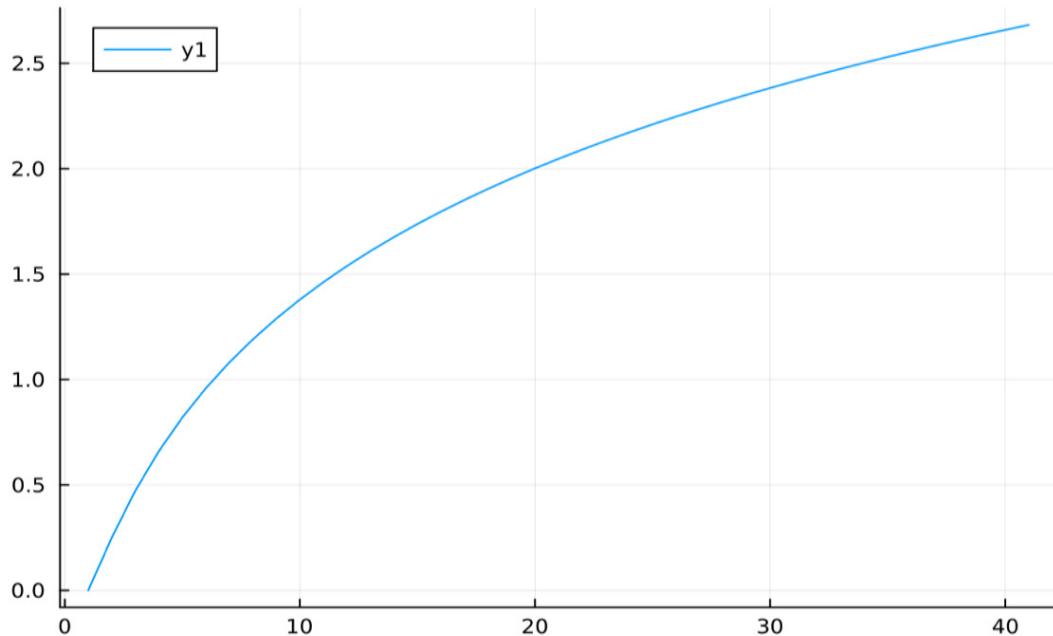


Figure 11.2 – Plot of Ali Baba curves for a limit of  $n = 40$

For the best time, I wished to compute the timings using `BenchmarkTools`, which for the  $4 \times 10^9$  on an M1 MacAir laptop is around 16.6 seconds.

Since the computation is linear, each decade takes 10 times longer, and we would expect that the sum up to  $4 \times 10^{10}$  will take around 166 sec (2 min 46 sec) if using integer values.

We will see later that we can do much better than this:

```
julia> @benchmark alibaba(4*10^9), samples=1, evals=1
BenchmarkTools.Trial: 1 sample with 1 evaluation.

  Single result which took 16.596s (0.00% GC) to evaluate,
  with a memory estimate of 480 bytes, over 3 allocations.
```

Use of the `@benchmark` macro means it is possible to specify the number of samples and evaluations per sample to take after, of course, exercising the function once on a smaller count to eliminate the computing time.

In this case, the final sum is up to a value of 20.1029. Soon after this, the algorithm as written “blows up” when using integer arithmetic. Of course, the algorithm (including `genie()`) could be rewritten using big integers, which would certainly influence the timings.

## Performance tips

Most people disagree on politics, religion, and taxation; actually, almost everyone agrees about taxation – that is, it is too high when applied to themselves. So, it is the case to some extent with respect to performance tips, and possibly there is a need to heed the founding fathers, with some reservations, so the best starting point is Julia's online documentation: <https://docs.julialang.org/en/v1/manual/performance-tips>.

I am going to pick a few of these that I feel are more important and, on a good day at least, I try to follow. Some of these are quite familiar, while others are a little more obscure:

- Avoid global variables
- Measure performance with `@time` or equivalent
- Pay attention to excessive memory allocation
- If possible, avoid parameterization by using abstract types
- Access arrays in memory order; viz., along the columnar direction
- Consider using views for slices
- Avoid string interpolation for I/O
- Fix deprecation warnings

### *Traceur*

Traceur is essentially a codified version of the Julia performance tips that examines running code to detect any obvious performance traps.

The driving routine is the `analyze` call, which checks for problems with globals, locals, dispatch, and return types and then outputs an appropriate warning.

To achieve something more interesting with our pantomime characters, I have altered the code very slightly by changing the `N_THIEVES` constant to a `nn_thieves` variable:

```
julia> nn_thieves = 40;
julia> function alibaba_1(n::Integer)
    @assert n >= nn_thieves
    s = 0.0
    k = n/nn_thieves
    a = Float64[]
    push!(a,s)
    for i in 1:n
        s += genie(i,2)
        (mod(i, k) == 0) && push!(a,s)
    end
```

```

    return a
end
alibaba_1 (generic function with 1 method)

```

The Traceur analysis is performed by use of the `@trace` macro, which works by retrieving the code type of the variety of a function:

```

julia> using Traceur
julia> @trace alibaba_1(40);
└ Warning: uses global variable Main.nn_thieves
└ @ REPL[6]:2
└ Warning:   is assigned as Union{Nothing, Tuple{Int64, Int64}}
└ @ REPL[6]:7
└ Warning:   is assigned as Union{Nothing, Tuple{Int64, Int64}}
└ @ REPL[6]:10
└ Warning: dynamic dispatch to _2 >= Main.nn_thieves
└ @ REPL[6]:2
└ Warning: dynamic dispatch to _2 / Main.nn_thieves
└ @ REPL[6]:4
└ Warning: dynamic dispatch to Main.mod(φ (%28 => %25, %56 => %51), _2
/ Main.nn_thieves)
└ @ REPL[6]:9
└ Warning: dynamic dispatch to Main.mod(φ (%28 => %25, %56 => %51), _2
/ Main.nn_thieves) == 0
└ @ REPL[6]:9
└ Warning: dynamic dispatch to Base.AssertionError("n >= nn_thieves")
└ @ REPL[6]:2

```

Due to our modification to `alibaba()` this now produces a bevy of warnings. Since the function does work, these are natural warnings and not errors.

## Debugging

There is an English aphorism, “*The road to hell is paved with good intentions*,” so I want to start by heaping praise on the souls who have, and still are, tackling the task of creating a debugger for Julia.

Why is this so difficult? In my dim and distant past when I wrote assembler for DEC PDPs and VAXs, debugging was straightforward since there was roughly a one-to-one correspondence between the machine code and the assembler source. This approach is still possible on Unix platforms via the ubiquitous `gdb` program but I don’t know how much it is used, although I believe it was once touted for use with Julia and LLVM.

Similarly, in conventional scripting languages, some of which immediately spring to mind, there is a reasonable correspondence between where you are (in the listing) and the computer (in the

executable code). Alas, this is not the case in Julia, which compiles the source via a few intermediate stages, eventually finishing up with native machine code and on different machines at that – and hence the problem.

I assume that most developers build up their code in the REPL (or VS Code) before launching it to an unforgiving public; at least, I do.

The first attempt at a Julia debugger was by Toivo Henningsson, who had a first stab at the name `Debug.jl`, and this was, at the time of writing, over 10 years ago. For the time it was quite useable, but suffered from the necessity to alter the code to use it.

A second attempt (Gallium) was from Keno Fischer, not one to shy away from difficult problems. Gallium employed an alternative paradigm, which was to change the actual native code when it was running. Unfortunately, Julia's internals change more often than the managers in the English football (soccer) Premier League, so maintaining Gallium was difficult, and Keno had a second attempt at linking to the “lowered” code.

The approach has been taken up by Tim Holy et al. in the `Debugger.jl` package, which is what I will describe now.

There are several core debugging operations to be expected from a source code debugger, which are listed here with the equivalent Debugger instruction in brackets:

- Execute the current line and move to the next (`n`)
- Jump to a specified line (`u`)
- Step into the next function call (`s`)
- Finish current function and step out (`so`)
- Set a breakpoint (`bp add`) \*\*
- Continue until the next breakpoint (`c`)
- Set a watch on a variable (`w add`) \*\*
- Display values of all variables (`fr`)
- Show current “state” of the program (`st`)

\*\*:

`bp` and `w` (*alone*) display the breakpoint and watch lists. There are `bp rm` and `w rm` variants to remove each of these respectively.

To indicate the use of Debugger, let's look at some of these “main” commands using the `aladdin()` routine.

We start by using the @enter macro followed by a function call; for example, aladdin(10):

```
julia> using Debugger
julia> @enter aladdin(10)
In aladdin(n) at /Users/malcolm/MJ2/Chp11/Code11/panto.jl:5
 5  function aladdin(n::Integer)
>6    @assert n > 0
 7    s = 0.0
 8    for i in 1:n
 9      s += genie(i,2)
About to run: (>)(10, 0)
```

The number of lines can be increased or decreased by typing + or - respectively.

We can use the u instruction to just go to a specific line number:

```
1|debug> u 8
In aladdin(n) at /Users/malcolm/MJ2/Chp11/Code11/panto.jl:5
 5  function aladdin(n::Integer)
 6    @assert n > 0
 7    s = 0.0
> 8    for i in 1:n
 9      s += genie(i,2)
10   end
11   return s
12 end
About to run: (Colon())(1, 10)
```

Then, the n instruction will take us to the next line:

```
1|debug> n
In aladdin(n) at /Users/malcolm/MJ2/Chp11/Code11/panto.jl:5
 6    @assert n > 0
 7    s = 0.0
 8    for i in 1:n
> 9      s += genie(i,2)
10   end
11   return s
12 end

About to run: (genie)(1, 2)
```

Adding a breakpoint is done by referring to the line number in the source code:

```
1|debug> bp add 9
1] /Users/malcolm/MJ2/Chp11/Code11/panto.jl:9
```

Issuing a few continue (c) commands, we can go around the loop, and then we will show the values of the “frame” variables with fr:

```
1|debug> fr
[1] aladdin(n) at /Users/malcolm/MJ2/Chp11/Code11/panto.jl:5
| n::Int64 = 10
| ::Tuple{Int64, Int64} = (4, 4)
| s::Float64 = 0.6597222222222222
| i::Int64 = 4
```

Rather than displaying the entire frame, especially when there are a greater number of variables, it is often more convenient to add watch points using w add.

The w instruction displays all the watched variables. These are removed using the w rm command:

```
1|debug> w
1] i: 6
2] s: 0.9586111111111111
```

Now, add another breakpoint at *line 11*:

```
1|debug> bp add 11
1] /Users/malcolm/MJ2/Chp11/Code11/panto.jl:9
2] /Users/malcolm/MJ2/Chp11/Code11/panto.jl:11
```

Remove the first breakpoint; this is denoted by the entry position in the list and not the line number:

```
1|debug> bp rm 1
1] /Users/malcolm/MJ2/Chp11/Code11/panto.jl:11
```

As a “lowered” debugger, it is possible to look at the code with L; this is a toggle and can be used to return to the source code representation:

```
1|debug>
In aladdin(n) at /Users/malcolm/MJ2/Chp11/Code11/panto.jl:5
14 5 ... %14 = @_3
15 |      i = Core.getfield(%14, 1)
16 |      %16 = Core.getfield(%14, 2)
>17 |      %17 = s
18 |      %18 = Main.genie(i, 2)
19 |      s = %17 + %18
20 |      @_3 = Base.iterate(%9, %16)
```

Toggle back with L and continue (by issuing c) to *line 11*, as this is now breakpoint 1.

Alternatively, we could just go to this line by typing `u 11`:

```
In aladdin(n) at /Users/malcolm/MJ2/Chp11/Code11/panto.jl:5
 8     for i in 1:n
 9         s += genie(i,2)
10    end
>11   return s
12 end
About to run: return 1.4618451509008867
```

A final `c` instance will return to the REPL:

```
1 |debug> c
1.4618451509008867
```

If you enter the function again, the breakpoints and watch variables are still set. This is useful because it is possible to use the `@run` macro rather than `@enter`; this begins a debugging session AND jumps to the first breakpoint.

There is much more that can be done with this debugger. For example, it is possible to step into and out of functions – try doing this on `genie` at *line 9*.

**Note**

`Debugger.jl` has extensive help, which can be displayed by using the `?` instruction.

## Revision

`Revise.jl` makes it possible to modify code and use the changes without leaving Julia.

It is authored by Tim Holy, who is responsible for several packages, some of note we have met already; viz., on image support and the debugger (just now).

With `Revise`, it is possible in the middle of a session to update packages, switch Git branches, and edit the source code in the editor of choice. The changes will typically be incorporated into the very next command issued from the REPL. This saves the overhead of restarting Julia, reloading packages, and waiting for code to compile.

Start with the standard `Example` package, which contains a couple of routines – a Hello World one (`hello()`) and a simple arithmetic routine (`domath()`):

```
julia> using Pkg; Pkg.add("Example")
#= Must import or use Revise BEFORE adding the Example package =#
julia> using Revise
julia> using Example
```

By issuing an `edit` command for either of the `hello()` or `domath()` functions, we will, in fact, open up the entire module source.

The Julia `edit` command opens the source in the default editor. I want to use a terminal style rather than a visual one; the choices for me are `vi(m)` or `nano`, as I never could get to grips with `emacs`:

```
julia> ENV["VISUAL"]="vim";
julia> edit(hello)
```

Now, we can change the `domath()` (say) routine to return  $2^*x + 5$  instead of  $x + 5$  and follow it with the next code to define a factorial function:

```
"""
fac(n::Integer)
Return factorial n! for n >= 0
"""
function fac(n::Integer)
    @assert n > 0
    return (n == 1) ? 1 : n*fac(n-1)
end
```

Don't forget to change the export line by adding the `fac` function; otherwise, it would need to be referenced as `Example.fac()`.

Now, just test that the `domath()` function has changed, and if so, see that the `fac()` function works also:

```
julia> domath(3.5)
12.0
julia> domath(3//7)
41//7
julia> fac(5)
120
julia> fac(0)
ERROR: AssertionException: n > 0
Stacktrace:
 [1] fac(::Int64) at /Users/malcolm/.julia/dev/Example/src/Example.jl:25
 [2] top-level scope at none:0
```

Our definition of a factorial function works for all positive integers, but it is normal also to define `fac(0)` as 1, which you will observe does not alter the values returned since `fac(1) === 1*fac(0)`.

So, we can amend the function, now by editing `fac` (rather than `hello`) to make this change:

```
julia> edit(fac)
The method is defined at line 25.
```

We need to alter the `@assert` and `return` statements and test now that `fac(0)` is allowed and that previous values are unchanged:

```
julia> function fac(n::Integer)
    @assert n >= 0
    return (n == 0)? 1 : n*fac(n-1)
end
julia> fac(0)
1
```

Using Revise, these work without having to exit Julia or restart and reload the package. This applies to changes to any package, whether loaded with `import` or `using`.

For Julia scripts, these need to be loaded with an `include` command.

Consider the following `fibs.jl` script, which has two definitions for the Fibonacci sequence, the top being active and the other commented out:

```
$> cat fibs.jl
function fib(n::Integer)
    @assert n > 0
    return (n < 3)? 1 : fib(n-1) + fib(n-2)
end
#=#
function fib(n::Integer)
    a = big(0)
    b = big(1)
    while n > 0
        (a, b) = (b, a+b)
        n -= 1
    end
    return a
end
#=#
```

If we include this as suggested previously and evaluate the 45th Fibonacci number, this takes nearly 3 seconds; large ones take progressively longer, and around 50, the elapsed time is very great, of the order of several minutes. The reason, as we discussed previously, is the double recursion in the function definition, so execution times for each increase quadratically with each value of `n`:

```
julia> includet("fibs.jl")
julia> @time fib(45)
```

```
12.151672 seconds (5 allocations: 176 bytes)
1134903170
```

The alternate definition does not use recursion, and so editing `fib`, simply by commenting out the first definition and using the second, produces the following output:

```
julia> @time fib(48)
0.035722 seconds (19.24 k allocations: 1.037 MiB)
4807526976

julia> fib(101)
573147844013817084101
```

This is much quicker, and even arbitrary large numbers can be computed because of the use of `big()`; otherwise, the algorithm is still quick but wraps due to LLVM integer overflow.

Revise can also be used to change any file defined by Julia's Base or its standard libraries. For example, try the following:

```
julia> edit(exp, (Int,))
julia> using Base64; edit(Base64.base64encode)
```

To ensure that every Julia session uses Revise, it is necessary to add `using Revise` to the `~/.julia/config/startup.jl` file.

Additionally, if you want to launch Revise automatically within IJulia, then it is necessary to create (or add) the following to a `~/.julia/config/startup_ijulia.jl` file:

```
try
    @eval using Revise
catch
end
```

Tim Holy introduced Rebugger, which was a “*kind*” of debugger to work with Revise, to quote the `Rebugger.jl` GitHub page’s README file.

#### Note

Rebugger is in the early stages of development, and users should currently expect bugs.

Since this has not been touched for over 4 years and `Debugger.jl` has emerged, I would reasonably be confident in assuming that the Rebugger is facing retirement.

## Profiling

Software profiling is a type of analysis that measures memory usage, code complexity, time spent executing specific instructions, and/or the frequency of individual function calls.

Julia has a built-in profiler in its standard library, run using the `@profile` macro, which we will apply to `aladdin()`:

```
julia> using Profile
julia> @profile aladdin(10);
julia> Profile.print()
Overhead | [+additional indent] Count File:Line; Function
=====
| 7 @Base/client.jl:522; _start()
| 7 @Base/client.jl:322; exec_options(opts::Base.JLOptions)
|   7 @Base/client.jl:405; run_main_repl(interactive::Bool,
quiet::Bool, banner::Bool, history_file::Bool, ...
|     7 @Base/essentials.jl:813; invokelatest
|     7 @Base/essentials.jl:816; #invokelatest#2
|       7 @Base/client.jl:421; (::Base.var"#1017#1019"{Bool, Bool,
Bool}) (REPL::Module)
```

Plus, it has lots more undecipherable output, for which I will refer you to the Julia documentation: <https://docs.julialang.org/en/v1/stdlib/Profile>.

That said, there are a number of visual profilers that use the standard library `Profile` module coupled with an additional `FlameGraphs` package, to add functionality and a degree of interactivity; of note are:

- `ProfileView`, a `Gtk`-based GUI for interacting with flame graphs
- `PProf`, an interactive, web-based profile GUI explorer visualization
- `ProfileVega`, a profiler using `Vega` and useful when using `Jupyter` notebooks
- `ProfileSVG`, a package for writing flame graphs to `SVG` format

However, I do not intend to discuss any of these either; rather, I will look at a particular favorite one of mine: `StatProfilerHTML`.

Generating a profile is straightforward, by means of the `@profilehtml` macro, and this time I will choose to profile `alibaba()` rather than `aladdin()`, as it will prove more informative:

```
julia> using BenchmarkTools, StatProfilerHTML
julia> include("panto.jl")
julia> @profilehtml alibaba(10^6);
```

The macro creates a subdirectory called `statprof` that has a number of SVG flame graphs plus a number of HTML files, all held together by an `index.html` file:

```
$> cd("statprof")
$> ls -l flamegraph.svg index.html
-rw-r--r--@ 1 11273 28 Nov 23:12 flamegraph.svg
-rw-r--r--@ 1 8090 28 Nov 23:12 index.html
# There are other html files with correspond to snippets in # the
index file, panto.jl-* .html is one is quite fun.
#
shell> ls panto*.html
panto.jl-34c557c70098b825b16ca1de500fbc5c30b671b7.html
```

*Figure 11.3* shows a portion of the (top) of the flame graph, plus a portion of the `index.html` file (below the graph).

The bit highlighted is this:

```
41 |
12 (75%) | (mod(i, k) == 0) && push!(a, s)
```

This suggests that a considerable length of time is being spent just on this line of code.

We noted before that using an empty array and pushing values onto it might appear elegant, but it was not sensible when we actually knew the size of the array would be 41.

Also, the `mod()` function can be eliminated by using a loop, the outer being over the number of thieves and the inner over the increment (viz., limit /40):

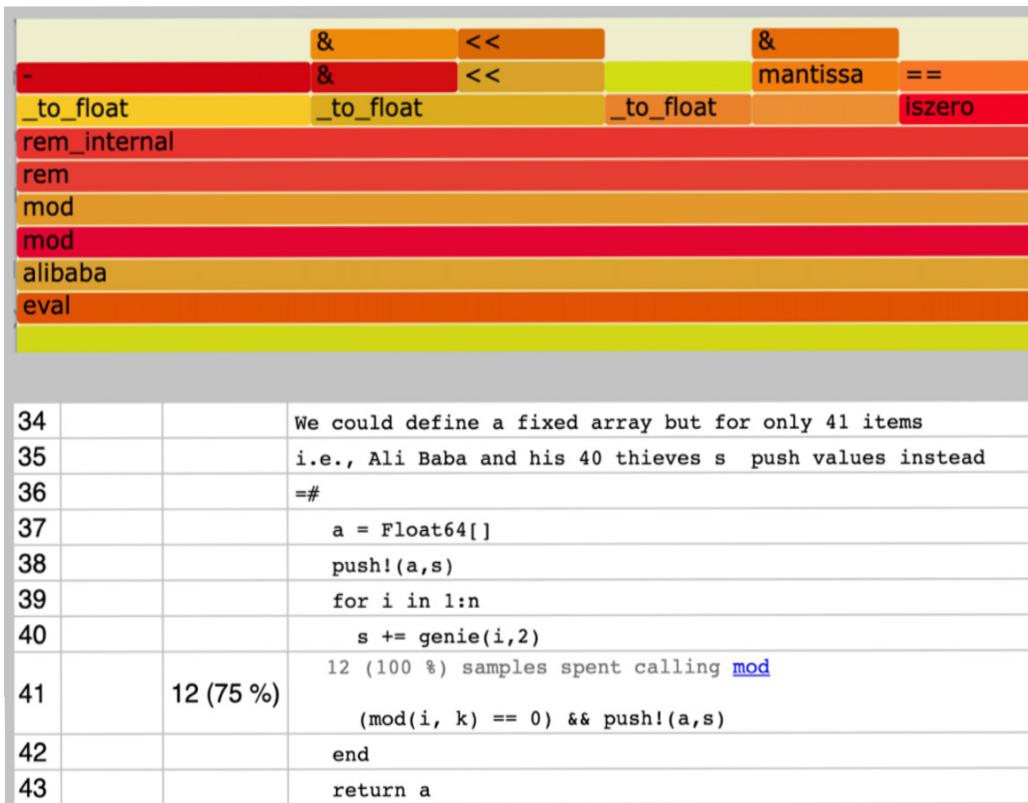


Figure 11.3 – Flame graph (SVG) and panto.jl- 4c557c7xxxx.html

A new version of `alibaba_2()` implements this, and actually, we pass the value of the increment rather than the final limit:

```
julia> function alibaba_2(n::Integer)
    @assert n > 0
    s = 0.0
    a = Array{Float64}(undef, 41)
    a[1] = s
    for i in 0:39
        for j = 1:n
            k = i*n + j
            s += genie(k,2)
        end
        a[i+2] = s
    end
    return a
end
```

First, let's test that the new version creates the same results as the original, noting that the value passed to it will be 40 times less:

```
julia> b1 = alibaba(120)
julia> b2 = alibaba_2(3);
julia> @assert b1[end] == b2[end]
```

Now, run both functions for a bigger limit ( $4 \times 10^6$ ):

```
julia> @btime b1 = alibaba(40*10^5);
       69.707 ms (3 allocations: 480 bytes)
julia> @btime b2 = alibaba_2(10^5)
       16.255 ms (1 allocation: 400 bytes)
```

The result is remarkable, reducing the execution time by over 25%.

You might ask whether the power function in `genie()` takes more time than just a simple multiplication, as in fact the square is being computed. We can easily check this by running this code:

```
julia> sqx(x) = x*x; sqx(1);
julia> @btime let k = 0;
           for i in 1:10^15 k += sqx(i) end end
       2.474 ns (0 allocations: 0 bytes)
julia> sqp(x) = x^2; sqp(1);
julia> @btime let k = 0;
           for i in 1:10^15 k += sqp(i) end end
       2.474 ns (0 allocations: 0 bytes)
```

Why the same result? In fact, LLVM, as is similar to many compilers, has an efficient optimizer and in this case generates the same code, which can be confirmed by the use of the `@code_native` macro.

## Creating packages

When parsing Julia packages either in the `.julia/packages` folder or on GitHub, the reader will have observed that they all have a definitive structure. More specifically, at the top level, there is a `README.md` markdown file, together with at least two TOML files: `Project.toml` and `Manifest.toml`.

In addition, there are always at least the following two subdirectories:

- `src`: This contains the source, including a Julia file that has the package name but with a `.jl` extension
- `test`: This contains a `runtests.jl` script, which contains a series of tests to run via the package manager to check the integrity of the package

The source directory may contain many other Julia packages that are then “included” by the main package files; for example, the `Plots.jl` file has (currently) 19 additional Julia files.

Also, the main program usually contains any `const` definitions and also a list of functions, structs, and so on, whose names will be exported when using rather than importing the package.

Any function can still be used when not exported by fully qualifying it, as we saw earlier when specifying the Gadfly backend as `Gadfly.Display` when calling the `pushdisplay()` routine.

There are often other subdirectories such as `docs`, `example`, `ext`, `data`, and so on, but these are not required, nor are the actual names prescribed, although the package author is likely to choose ones that describe their purposes.

#### Note

Many package authors use `Documenter.jl` to create separate documentation, built up from markdown files and docstring comments in source files, and conventionally, this goes into the `docs` folder; look at the `Flux.jl` package for a comprehensive example.

Choosing an example package to work with presented a problem as the most obvious themes have been tackled, or worse still, ever an author’s dilemma, are in the process of being so. I resolved this by creating a package based on some of the functions developed in this book, which should be familiar and certainly are due to me.

## A “funky” module

I will deal first with the development of the module and then follow it up with the creation of the package layout. All the functions in it (`Funky.jl`) have been presented already and the entire module is available in the code accompanying this chapter, so I will just discuss the principal features here.

We start by defining the sources to be used by Funky and also define a `PUNCT` constant, which is needed by the `wordcount()` function to strip off any trailing punctuation from strings:

```
module Funky
using HTTP, CSV, DataFrames, TimeSeries, IndexedTables
using Printf, Dates, Statistics
const PUNCTS =
    [' ', '\n', '\t', '.', ',', ':', ';', '!', '?', '\'', '\"'];
include("ftop.jl")
include("queens.jl")
include("quandl.jl")
include("panto.jl")
```

The `f_top` routine, to find top files and queens functions, is reasonably lengthy, so the code is placed in its own source files; this is also the case for the pantomime function, which has been introduced in this chapter. The sources are assumed to be found in the same folder as `Funky.jl`.

Also, I have included a `quandl.jl` file that contains a source for a `quandl()` routine, which can be used to download financial data from the internet, and for this, a number of packages are required. These also need to be referenced in the project's TOML files.

Defined are “special” mathematical functions such as `basel`, `hailstone`, and `kempner`, and also my versions of more familiar ones to evaluate `fac` factorials and terms in the `fib` Fibonacci series.

Typical of these is `basel`, and I have shown the actual doctext to indicate how verbose it should be:

```
"""
basel(N::Integer)
Compute the sum of the Basel sequence
Returns a real number for positive values of N.
The solution was finally derived by Euler in 1734 and
is equal to  $\pi^2 / 6$ .
# Examples
julia> basel(10^6)
1.64493306684877
"""

function basel(N::Integer)
    @assert N > 0
    s = 0.0
    for i = 1:N
        s += 1.0/float(i)^2
    end
    return s
end
```

There are a couple of wrapper routines defined in *Chapter 5* that link to C functions present in the `libmyfun`s shared library:

```
basel_c = ccall(:basel,"libmyfun"), Cdouble, (Clong,), n)
horner_c = ccall(:horner,"libmyfun.dylib"), Cdouble,
                (Cdouble, Ptr{Cdouble}, Clong), x, aa, n)
```

Ideally, we will need to supply a `build.jl` script that would reside in a `deps` subfolder of the module, together with the C sources and makefile, in order to build and deploy the library.

The next batch of functions is related to text manipulation, which I will not list here.

Finally, we include the utility macros developed in *Chapter 4*, such as my version of a @bmk timing macro:

```
macro bmk(fex, n::Integer)
    quote
        let s = 0.0
        if $(esc(n)) > 0
            val = $(esc(fex))
            for i = 1:$esc(n)
                local t0 = Base.time_ns()
                local val = $(esc(fex))
                s += Base.time_ns() - t0
            end
            return s/$(esc(n)) * 10e9
        else
            Base.error("Number of trials must be positive")
        end
    end
end
```

At the end (or start) of the module, we wish to define which function and macros will be visible after a reference to it via a `using` statement:

```
export basel, fac, fib, hailstone, Kempner
export isdate, wordcount, filter, ftop, quandl
export @traprun, @bmk, @until
```

At the end, or start, of the module, it is important to export functions and macros that will be visible after a `using` statement. Otherwise, they will need to be called via a fully qualified reference; for example, `Funky.alibaba()`.

To test the module at this point, we need to just include it and involve it with a `Main.Funky` statement or just `.Funky`:

```
julia> include("Funky.jl")
julia> using Main.Funky
```

If all is well, this will have the effect of defining the functions and adding help text. For functions not exported, the help text is available, but again using a fully qualified reference:

```
help?> Funky.alibaba
Compute and store the items using Aladdin's genie storing each
partial sum of the series  $i/(i+1)^2$  in an array up to a value of 40.
So the preferable count to choose is a multiple of 40 in order to
capture the final value sum of the series in the array.
```

## Creating the layout

The next task is to prepare the package framework. Although this can be done using a text editor, there is a convenient `PkgTemplates` package that makes this task much simpler.

To create the required files, use the `Template()` routine to generate the necessary details of the author(s), names, and emails, plus a host of other information that is derived and default unless otherwise written:

```
julia> using PkgTemplates

julia> tpl =
    Template(user="sherrinm",
        authors=["Malcolm Sherrington<malcolm@amisllp.com>"])

Template:
  authors: ["Malcolm Sherrington <malcolm@amisllp.com>"]
  dir: "~/julia/dev"
  host: "github.com"
  julia: v"1.0.0"
  user: "sherrinm"

. . .
. . .
#= === plus lots of other stuff === =#

# Use the template on the proposed Funky package
julia> tpl("Funky")
```

See <https://juliaci.github.io/PkgTemplates.jl/stable/user> for more documentation on the package.

The command creates a `Funky` directory, by default in the `.julia/dev` folder, which contains the following files/folders:

```
$> cd Funky
$> ls
LICENSE  Manifest.toml  Project.toml  README.md  src  test
```

One file not mentioned so far is `LICENSE`, which is a text file by default providing MIT software, referencing the author and the current year:

```
MIT Licenses
Copyright (c) 2023 Malcolm Sherrington
. .
. . .
```

Also, the `src` folder contains an empty `Funky.jl`, module file, to be replaced or added to the sources, although developed.

In the test folder is a `runitests.jl` proforma file:

```
$> cat test/runitests.jl
using Funky
using Test
@testset "Funky.jl" begin
    # Write your tests here.
end
```

The first few lines of the `Project.toml` and `Manifest.toml` files look like this:

```
$> head Project.toml
name = "Funky"
uuid = "2caceddd-5cab-4f8b-892c-da06cd331579"
authors = ["Malcolm Sherrington"]
version = "1.0.0-DEV"
$> head Manifest.toml
# This file is machine-generated - editing it directly is not advised
julia_version = "1.9.3"
manifest_format = "2.0"
project_hash = "f060a2e64e479ab6465690e0158022ef99ef3465"
```

One task remaining is to create task files that will be executed via the package manager via `Pkg.test("Funky")` and then executed from the `runitests.jl` file.

If all goes well, testing declares no errors, after often quite extensive output.

Have a look at the `BenchmarkTools` package, one that should be already available to you; otherwise, add it like so:

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

This produces copious amounts of information, the important bits being that all tests are reported as having passed:

```
Testing BenchmarkGroup...
Test Summary: | Pass  Total  Time
benchmarkset |     3      3  0.3s
Testing serialization...
Test Summary: | Pass  Total  Time
Successful (de)serialization |     7      7  2.4s
Test Summary: | Pass  Total  Time
Deprecated behaviors |    11     11  1.1s
Test Summary: | Pass  Total  Time
```

```
Error checking | 2 2 0.0s
Test Summary: | Pass Total Time
Error checking | 2 2 0.0s
Testing BenchmarkTools tests passed
```

## Collaborating with Git

**Git** is a mnemonic for **Global Information Tracker** and is a popular source control system, like others such as **Source Code Control System (SCCS)** and **Revision Control System (RCS)**. The name is a little unfortunate since to a “Brit” git means something else; viz., a relatively nasty person, one who invariably identifies as male. The system was developed by Linus Torvalds, of Linux fame, way back in 2005, and some of the naming is down to him. Its usage has since become extremely widespread in the open source community, so inevitably you, the reader, will use it and probably are doing so already.

There are four different approaches I have identified when wishing to work with Git:

1. Use the command-line version.
2. Interact with the interface provided by the website of the system being accessed; for example, GitHub, but also GitLab, Bitbucket, and so on.
3. Utilize a function that is part of an editor or IDE, often one such as VS Code; other popular development systems such as IntelliJ IDEA and Eclipse are also possibilities.
4. Work with a Git client; there are a number of these depending on the operating system being used, although some are cross-platform. For a review of many of these, visit <https://www.hostinger.com/tutorials/best-git-gui-clients>.

There is a great amount of information available on Git, in terms of books, online tutorials, and videos.

My only advice is to get familiar with the command-line approach, even if not intending to use that on a regular basis, as it will be a good basis for understanding how Git works. There are over a hundred commands in Git, all of which are available this way to look for and download a good Git cheat sheet that will provide a comprehensive list.

## Quo Vadis, Julia?

What have I missed?, quite a lot you may be thinking. What I want to do in this final section is just to highlight some of these aspects and perhaps pull the threads together and see what I might conclude as to where Julia might be going from now (viz., Xmas 2023).

Julia’s original reason for existence over 10 years ago was that it compiled its source code into native code, hence providing a great reduction in execution times. But what one can create, another can emulate, and I have seen that coming out of the MIT stable, following on from Julia, is a system called Codon that purports to do the same to the ubiquitous Python language: <https://github.com/exaloop/codon>.

In a year when the **James Webb Space Telescope (JWST)** has detected supergalaxies where there should be nothing at all, implying the Big Bang (Alan Guth's) cosmic inflation theory, dark matter, and energy may all be incorrect, this may be a futile occupation, but I hope at least it will point to some interesting areas for the emerging Julia practitioner into which they might immerse themselves.

If you had posed the same question to me when the first edition of this book was written, I would have said that Julia would rival existing “new” languages for a piece of the data science pie, and like most current big-bang cosmologists, I have been totally wrong.

Julia was initially seen as being designed for technical computing, making it a natural choice for use in scientific problems.

At present, work involving Julia is especially rich in **machine learning (ML)**; the following topics were too large to include in this book:

- FluxML (<https://fluxml.ai>)
- Turing.jl (<https://turing.ml>)
- SciML (<https://sciml.ai>)

One problem that could be encountered by the general reader is the raw computing power required to tackle the use of these in meaningful projects, especially where GPUs can be utilized to their advantage.

In this situation, it is worth turning to the cloud. Many services are available on Amazon (AWS), Google (Colab), and Microsoft (Azure), but a standout solution is probably best offered by JuliaHub (<https://juliahub.com>), formerly known as Julia Box.

JuliaHub, at present, provides a reasonably generous free tier to the private user of 20 hours per month and can be relied on to be kept up to date.

To quote the hype on the website, it is designed with access to CPUs and GPUs for multithreading and parallel and distributed computing, via a supercomputing infrastructure that allows teams to model breakthrough science and technology.

## Summary

This final chapter discussed some additional topics that may be required as the reader progresses from a casual user to a serious Julia developer.

It began by looking at the REPL and how Julia can be configured and used to execute scripts, and then continued by describing some of the various development tools that Julia provides for debugging, profiling, and testing of code.

This was followed by a discussion of producing Julia modules and, as an example, bundling (some of) the code supporting this book into a single package.

Finally, I closed the book with some comments on my take on the future Julia might pursue, take them as you will.

# Index

## Symbols

**@rget macro** 184, 185  
**@rlibrary macro** 187

## A

**abstract syntax tree (AST)** 123  
**abstract types** 10  
**Amazon Web Services (AWS)** 2  
**application program interface (API)** 348  
**ArgParse**  
    using 443, 444  
    versus Getopt 444-446  
**arguments**  
    default and optional arguments 89-92  
    keyword arguments 93, 94  
    passing 89  
    variable argument list 92, 93  
**arithmetic operators** 44  
**array** 46, 47  
    broadcast() function, using 47, 48  
**Atomicity, Consistency, Isolation, and Durability (ACID)** 348  
**automatic differentiation (AD)** 280-284

## B

**backends** 324  
    GR 325  
    PlotlyJS 325  
    plots, saving with HDF5 327  
    PythonPlot 325, 326  
**Basel function** 171-173  
**Basel problem** 22-24  
**Berkeley Database (BDB)** 372  
**big integers** 45, 46  
**BigQuery** 379  
**binary files** 217, 218  
    text, processing 218-220  
**binding** 395  
**Booleans** 44  
**box plot of stock prices example** 328, 329  
**Broyden-Fletcher-Goldfarb-Shanno (BFGS) method**  
    working with 185  
 **BSON** 378  
**Buffon's needle** 422  
**Bulls and Cows** 64-67  
**byte array literal** 59, 60  
    rules 60

**C****C**

and Fortan 168, 169  
 Basel function 171-173  
 Horner function 171-173

**C++ 174-176****Calculus 278**

automatic differentiation 280-284  
 differentiation 278-280  
 quadratures 284

**catastrophic equations 273-275****chaos theory 275, 276****characters 55, 56**

strings 56-58

**closures 87, 88****Command Palette 21****comma-separated-values (CSV) 220**

and delimited (DLM) files 220  
 DLM file formats 223-226  
 file formats 220-222

**Common Lisp Object System (CLOS) 120****complex numbers 60, 61****compose() function 317-319****composite Vehicle data type 104-109****Comprehensive R Archive**

Network (CRAN) 236

**computing pi 113-115****concrete types 10****Condon**

reference link 471

**consistency, availability, and partition tolerance (CAP) 348****contour plot 330, 331****Convex.jl 290****co-routine 419****Couchbase 379****CouchDB 379****C types**

mapping 170

**D****data arrays 73, 74****database**

interfacing to 348, 349  
 preliminaries 347, 348

**database interface (DBI) 35****database management system (DBMS) 191, 348****database manipulation language (DML) 355****DataFrames 73, 74, 237**

Excel spreadsheets 237-239  
 R-Datasets 239-244

**data science 5****data types**

abstract 41  
 primitive 41

**default and optional arguments 89-92****delay DEs (DDEs) 268****delegate objects 120****DELETE request 402****dense 71****development tools 448**

debugging 454-458  
 document strings 449-452  
 performance tips 453  
 Revise.jl 458-461  
 software profiling 462-465  
 Traceur 453, 454

**dictionaries (dicts) 15, 74, 75****differential algebraic equations (DAEs) 268****differential equations (DEs) 268**

catastrophic equations 273-275  
 chaos theory 275, 276

---

ODEs 268-271  
 real pendulum, simulating 271, 272  
 stochastic DEs 277, 278  
**differentiation** 278, 279, 280  
**digital signal filters** 264, 265  
**discrete FT (DFT)** 261  
**Distributed Array** 424-427  
**distributed data sources** 428-432  
**document databases** 373-376  
**Document Object Model (DOM)** 413  
**do syntax** 82  
**Dual Number** 162

**E**

**editors and IDEs, Julia**  
 Juno 19  
 Jupyter 17-19  
 Pluto 21, 22  
 Visual Studio Code (VS Code) 20, 21  
**eigenvalues** 258, 259  
**eigenvectors** 258, 259  
 computing 259, 260  
**Elastic Compute Cloud (EC2)** 35  
**enumerations** 112, 113  
**explicit typing** 51-53

**F**

**fast FTs (FFTs)** 261  
**filesystem**  
 working with 204, 205  
**first in, first out (FIFO)** 78, 400  
**first in, last out (FILO)** 78  
**FluxML**  
 reference link 472

**Fortran routines**  
 calling 170, 171  
**Fourier transform (FT)** 261  
**frequency analysis, signal processing** 261  
 digital signal filters 264, 265  
 smoothing and filtering 262-264  
**functions** 82  
 and macros, selecting between 152  
 closure 87  
 do syntax 82  
 first-class objects 83-87

**G**

**Gadfly** 312-315  
 compose() function 317-319  
**Gaston** 319  
 plotting functions 315-317  
**Gaston** 319  
**General Purpose Simulation System (GPSS)** 290  
**generated function** 149-151  
**Genie** 415-419  
 URL 416  
**geometric Brownian trajectories**  
 computing 27-29  
**Getopt**  
 versus ArgParse 444, 445  
**GET requests** 402  
**GR** 325  
**graphic packages** 305  
 Gadfly 312  
 PyPlot 306  
 PythonPlot 306  
 Winston 309  
**Gumbo** 413

**H****Hierarchical Data Format v5 (HDF5)**

- plots, saving with 327

- and JLD files 226, 227

**higher dimensional vectors 116****high-order algebraic equations 260****Horner function 171-173****HTTP 379-381**

- body 380

- header 379

- status code 379

**HTTP methods 402-404****I****image processing 339**

- resizing 340-342

**Images(.jl) family 342, 343**

- TB1.jpg image 343-345

**implicit-explicit (IMEX) 268****implicit typing 51-53****index sequential access method (ISAM) 366****inline graphics**

- displaying 25-27

**in-memory databases (IMDBs) 366****integer type 40-42****interactive development**

- environments (IDEs) 1

**intermediate representations (IR) 7**

- LLVM code 123

- lowered code 123

- typed code 123

**internal representation (IR) 123****I/O 207, 208**

- terminal input 209-211

- terminal I/O 208

- terminal output 208

**J****James Webb Space Telescope (JWST) 472****Java 189, 190****Java Database Connectivity (JDBC) 349, 364****Java Virtual Machine (JVM) 364****JDBC databases 364, 365****Julia**

- .julia subdirectory 435

- accessing 179

- and OS, configuring 434

- comparison, with other languages 5, 7

- data science 5

- designing, considerations 4

- editors and IDEs 16, 17

- environments 436-438

- juliaup package 436

- overview 2, 3

- packages subfolder 436

- performance times 7-9

- philosophy 3, 4

- resources, obtaining 434

- script 12, 13

- starting up 438, 439

- TOML files 435

- usage, considerations 9-11

- working with 11, 12

**Julia 101 2****Julia Box 472****Julia code 22**

- Basel problem 22, 24

- geometric Brownian trajectories,  
computing 27-29

- inline graphics, displaying 25-27

**Julia data format (JLD) 227, 228****JuliaDB 428****JuliaDiff community group 165**

**Julia Editor Support group**  
reference link 17

**JuliaHub**  
reference link 472

**JuliaInterop group**  
supported languages 203

**Julia Micro-Benchmarks**  
URL 6

**Julia modules** 153  
base 153  
core 153  
main 153

**Julia packages**  
adding 30, 32  
database packages 35  
graphics 34, 35  
listing 30, 32  
machine learning 36  
management 29  
removing 30, 32  
selecting and exploring 33  
statistics and mathematics 34  
testing 32  
web and networking 35

**Julia set** 67-69

**JuliaWeb** 401

**JuMP**  
Convex.jl 290  
knapsack problem 286-288  
Optim 288, 289  
travelling salesman problem (TSP) 288  
using 285

**Juno** 19

**Jupyter** 17-19

## K

**kernel density** 246-249  
**keyword arguments** 93, 94  
**knapsack problem** 286, 287  
**KV datastores** 366  
Redis 367-372

## L

**lazy evaluation** 147, 148  
**Lightning Mapped Database (LMDB)** 366  
**Lightweight Directory Access Protocol (LDAP)** 347  
**Linear algebra** 254, 255  
eigenvalues 258, 259  
eigenvectors 258, 259  
high-order algebraic equations 260  
matrix decomposition 256  
simultaneous equations 257, 258  
**LINQ queries** 388  
**LMDB** 372, 373  
**localhost** 394  
**logical operators** 44  
**long-term support (LTS)** 2, 436  
**looking-glass world echo server** 396-399  
**Low-Level Virtual Machine (LLVM)** 1, 45  
**Luxor** 300-302

## M

**machine learning (ML)** 36, 472  
**macros** 134, 135  
and functions, selecting between 152  
expansions 139-142  
hygiene 138, 139  
lazy evaluation 147, 148

MacroTools 143  
reductions 145-147  
timing 136-138

**MacroTools 143-145****Makie 335-338**

backends 335, 336

**MAMP(OSX) 407****Map-Reduce 424-427****Massachusetts Institute of Technology (MIT) 2****matrices 46****matrix decomposition 256****matrix operations 53-55****memcache 372****metaprogramming 129**

code tree, manipulating 132, 133  
symbols and expressions 129-132

**methods 157, 158****middleware 411****model-view-controller (MVC) 419, 202****Modern Applied Statistics**

with S (MASS) 187

**modular integers 155-157****modularity 152-154****modularization 109, 110****module**

loading 154, 155

**moving average (MA) 263****multi-dimensional arrays 70**

sparse diagonal matrices 73

sparse matrices 71, 72

sparse vectors 72

**multidimensional vectors 113-115****multiple dispatch 83, 100, 119-122**

code generation 122-128

**multiple machines**

task, running on 428

**Mux 411-413**

branching 412  
dispatching 411  
stacking 412

**MySQL 355-358**

native MySQL 358-360  
ODBC 358  
PyCall, using 360, 361

**N****named pipes 395, 400, 401****needles 422-424****NoSQL databases 366**

document databases 373-376

KV datastores 366

**numbers**

and strings, converting between 98, 99

**O****object-oriented (OO) language 202****Open Database Connectivity (ODBC) 349, 358****Optim 288, 289****optimization 285**

JuMP, using 285

**ordered pair 160-164****ordinary DEs (ODEs) 268-271****OS and pipelines**

commands, running 191-193

large files, finding 195-197

text processing and pipes 194, 195

working with 191

**P****packages, creating**

- collaborating, with Git 471
  - funky module 466-468
  - layout, creating 469-471
  - software profiling 465, 466
- packages, with Python wrappings** 180-182

**parallel computing** 419**parameterization** 115, 116**partial DEs (PDEs)** 268**PATCH request** 402**Perl** 197

- examples 197-200

**Perl 6** 201**PGFPlots** 303-305**Pi**

- calculating 117, 118

**PI(ns)** 422, 423, 424**Pkg**

- used, for adding modules 15, 16

**Plotly** 328

- reference link 329

**PlotlyJS** 325, 328**Plots API** 319-321

- backends 324
- multiple plots, creating with layouts 321, 322
- recipes 322-324

**plotting functions** 315-317**Pluto** 21, 22**Posix-compliant** 208**PostgreSQL** 362, 363**POST requests** 402**primitive types** 42, 43**process I/O channels**

- using 200, 201

**Programming Language One (PL/I)** 64**PUT request** 402**PyPlot** 306-309**Python** 177, 178, 202, 203**PythonPlot** 306-326**Q****quadratures** 284**Quandl** 369**Queens problem** 97, 98**Queryverse** 385, 386

- LINQ queries 388

stocks dataset, querying 386-388

Vega-Lite 389-391

**query server** 407-409**QuestDB (QDB)** 379-385**queues** 78, 79**R****R** 185, 186**random DEs (RDEs)** 268**R API** 187**rational numbers** 62, 63**Rational type** 100-104**R-Datasets** 239-243**read-evaluate-print loop (REPL)** 1, 69**recursive functions**

- computing 49-51

explicit typing 51-53

implicit typing 51-53

**Redis** 367-372

download link 367

**regular expressions (regexes)** 58, 59

URL 59

**relational databases** 349

building 349-352

interfacing with 353

JDBC databases 364, 365

MySQL 355-358  
PostgreSQL 362, 363  
SQLite 353-355  
**release candidate (RC)** 436  
**remote procedures** 421, 422  
**REpresentational State Transfer (REST)**  
    interfacing with 377  
    JSON/BSON formats 377, 378  
    web database 379  
**Riak** 379  
**R (language)** 183  
    @rimport macro 187, 188  
    @rlibrary macro 187, 188  
    R 185, 186  
    R API 187  
    R REPL mode 183, 184  
    variables, exchanging with @  
        rget and @rput 184  
**routing** 409-411  
**R REPL mode** 183, 184  
**Ruby** 202

## S

**SciML**  
    reference link 472  
**scope** 94-97  
**script, Julia**  
    modules, adding with Pkg 15, 16  
    rules, scoping 14, 15  
**server** 393, 394  
**sets** 77, 78  
**Sieve of Eratosthenes algorithm** 63, 64  
**signal processing** 261  
    frequency analysis 261  
    image convolutions 266, 267

**SimJulia** 290  
    bank teller example 291-295  
**Simple Storage System (S3)** 35  
**simultaneous equations** 256, 258  
**single character** 55  
**socket** 393-395  
**software profiling** 462-465  
**sparse diagonal matrices** 73  
**sparse matrices** 71, 72  
**sparse vectors** 72  
**SQLite** 353-355  
**stacks** 78, 79  
**standalone Julia** 440  
    scripting 440-443  
    scripting, with ArgParse 443, 444  
    system images 446-448  
**standard input (stdin)** 207  
**standard output (stdout)** 207  
**static libraries** 168  
**statistics** 244-246  
    hypothesis, testing 249, 250  
    kernel densities 246,-249  
**StatsPlots** 331-335  
**stochastic DE (SDE)** 277, 278  
**stochastic ODEs (SODEs)** 268  
**stochastic PDEs (SPDEs)** 268  
**stochastic simulations** 290  
    SimJulia packages 290  
**strings** 56-58  
    regular expressions (regexes) 58, 59  
    version strings 59  
**structured datasets** 220  
    HDF5 226, 227  
    Julia data format (JLD) 227, 228  
    XML files 228-232  
**symbolic differentiation (SD)** 278

**T**

**tab-separated value (TSV)** 223  
**task** 419, 420  
  running, on multiple machines 428  
**TCP servers** 407  
**test harness**  
  adding 159  
**testing process** 159, 160  
**text files** 212, 213  
  text, processing 214-217  
**textual visualization** 298  
  inline displays 298, 299  
  Luxor 300-302  
  PGFPlots 303-305  
  turtle graphics 302, 303  
**time series** 232-236  
**Tom's Obvious Minimal Language (TOML) format** 30, 435  
**Traceur** 453, 454  
**Transmission Control Protocol (TCP) socket** 395, 396  
**travelling salesman problem (TSP)** 288  
**tuple** 83  
**Turing.jl**  
  reference link 472  
**turtle graphics** 302, 303  
**typealias mechanism** 110, 111

**U**

**Uniform Resource Indicator (URI)** 405  
**User Datagram Protocol (UDP)**  
  socket 395, 396  
**utility functions** 404-407

**V**

**variable** 40  
**vectors** 46  
**Vega-Lite** 389  
**version strings** 59  
**visualization frameworks** 328  
  basic image processing 339  
  image processing 339  
  Images(.jl) family 342, 343  
  Makie 335-338  
  Plotly/PlotlyJS 328  
  StatsPlots 331-335  
**Visual Studio Code (VS Code)** 20, 21

**W**

**WAMP (Windows)** 407  
**web**  
  working with 401, 402  
**web app, building in Julia with Genie.jl**  
  reference link 416  
**web crawlers** 413-415  
**well-known ports** 394  
**Winston** 309, 311, 312

**X**

**XAMPP (Linux)** 407  
**XML files** 228-232

**Y**

**Yet Another Markup Language (YAML)** 435