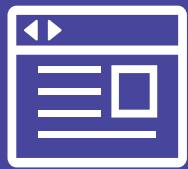
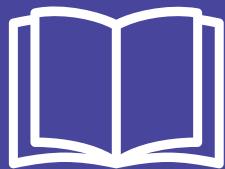


Ivo Balbaert, Avik Sengupta,
Malcolm Sherrington

Julia: High Performance Programming

Learning Path

Leverage the power of Julia to design and develop high performing programs



Packt

Julia: High Performance Programming

Leverage the power of Julia to design and develop high performing programs

A course in three modules

Packt

BIRMINGHAM - MUMBAI

Julia: High Performance Programming

Copyright © 2016 Packt Publishing

All rights reserved. No part of this course 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 course to ensure the accuracy of the information presented. However, the information contained in this course is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this course.

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

Published on: November 2016

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

ISBN 978-1-78712-570-4

www.packtpub.com

Credits

Authors

Ivo Balbaert
Avik Sengupta
Malcolm Sherrington

Content Development Editor

Priyanka Mehta
Graphics
Disha Haria

Reviewers

Pascal Bugnion
Michael Otte
Dustin Stansbury
Zhuo QL
Gururaghav Gopal
Dan Wlasiuk

Production Coordinator

Aparna Bhagat

Preface

Julia is a relatively young programming language. The initial design work on the Julia project began at MIT in August 2009, and by February 2012, it became open source. It is largely the work of three developers Stefan Karpinski, Jeff Bezanson, and Viral Shah. These three, together with Alan Edelman, still remain actively committed to Julia and MIT currently hosts a variety of courses in Julia, many of which are available over the Internet.

Initially, Julia was envisaged by the designers as a scientific language sufficiently rapid to make the necessity of modeling in an interactive language and subsequently having to redevelop in a compiled language, such as C or Fortran. At that time the major scientific languages were propriety ones such as MATLAB and Mathematica, and were relatively slow. There were clones of these languages in the open source domain, such as GNU Octave and Scilab, but these were even slower. When it launched, the community saw Julia as a replacement for MATLAB, but this is not exactly the case. Although the syntax of Julia is similar to MATLAB, so much so that anyone competent in MATLAB can easily learn Julia, it was not designed as a clone. It is a more feature-rich language with many significant differences that will be discussed in depth later.

The period since 2009 has seen the rise of two new computing disciplines: big data/cloud computing and data science. Big data processing on Hadoop is conventionally seen as the realm of Java programming, since Hadoop runs on the Java virtual machine. It is, of course, possible to process big data by using programming languages other than those that are Java-based and utilize the streaming-jar paradigm, and Julia can be used in a way similar to C++, C#, and Python.

The emergence of data science heralded the use of programming languages that were simple for analysts with some programming skills but who were not principally programmers. The two languages that stepped up to fill the breach have been R and Python. Both of these are relatively old with their origins back in the 1990s. However, the popularity of these two has seen a rapid growth, ironically from around the time when Julia was introduced to the world. Even so, with such estimated and staid opposition, Julia has excited the scientific programming community and continues to make inroads in this space.

The aim of this course is to cover all aspects of Julia that make it appealing to the data scientist. The language is evolving quickly. Binary distributions are available for Linux, Mac OS X, and Windows, but these will lag behind the current sources. So, to do some serious work with Julia, it is important to understand how to obtain and build a running system from source. In addition, there are interactive development environments available for Julia and the course will discuss both the Jupyter and Juno IDEs.

What this learning path covers

Module 1, Getting Started with Julia, a head start to tackle your numerical and data problems with Julia. Your journey will begin by learning how to set up a running Julia platform before exploring its various built-in types. You will then move on to cover the different functions and constructs in Julia. The module will then walk you through the two important collection types—arrays and matrices. Over the course of the module, you will also be introduced to homoiconicity, the meta-programming concept in Julia. Towards the concluding part of the module, you will also learn how to run external programs. This module will cover all you need to know about Julia to leverage its high speed and efficiency.

Module 2, Julia High Performance, will take you on a journey to understand the performance characteristics of your Julia programs, and enables you to utilize the promise of near C levels of performance in Julia. You will learn to analyze and measure the performance of Julia code, understand how to avoid bottlenecks, and design your program for the highest possible performance. In this module, you will also see how Julia uses type information to achieve its performance goals, and how to use multiple dispatch to help the compiler to emit high performance machine code. Numbers and their arrays are obviously the key structures in scientific computing – you will see how Julia's design makes them fast.

Module 3, Mastering Julia, you will compare the different ways of working with Julia and explore Julia's key features in-depth by looking at design and build. You will see how data works using simple statistics and analytics, and discover Julia's speed, its real strength, which makes it particularly useful in highly intensive computing tasks and observe how Julia can cooperate with external processes in order to enhance graphics and data visualization. Finally, you will look into meta-programming and learn how it adds great power to the language and establish networking and distributed computing with Julia.

What you need for this learning path

Developing in Julia can be done under any of the familiar computing operating systems: Linux, OS X, and Windows. To explore the language in depth, the reader may wish to acquire the latest versions and to build from source under Linux. However, to work with the language using a binary distribution on any of the three platforms, the installation is very straightforward and convenient. In addition, Julia now comes pre-packaged with the Juno IDE, which just requires expansion from a compressed (zipped) archive.

Who this learning path is for

This learning path is for data scientists and for all those who work in technical and scientific computation projects. It will be great for Julia developers who are interested in high-performance technical computing.

This learning path assumes that you already have some basic working knowledge of Julia's syntax and high-level dynamic languages such as MATLAB, R, Python, or Ruby.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this course – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail feedback@packtpub.com, and mention the title of the course in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to any of our product, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt course, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this course from your account at <http://www.packtpub.com>. If you purchased this course elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the course in the **Search** box.
5. Select the course for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

You can also download the code files by clicking on the **Code Files** button on the course's webpage at the Packt Publishing website. This page can be accessed by entering the course's name in the **Search** box. Please note that you need to be logged into your Packt account.

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

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

The code bundle for the course is also hosted on GitHub at <https://github.com/PacktPublishing/Julia-High-Performance-Programming>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books/courses – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this course. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your course, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book/course in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this course, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Module 1: Getting Started with Julia

The Rationale for Julia	3
The scope of Julia	3
Julia's place among the other programming languages	5
A comparison with other languages for the data scientist	7
Useful links	9
Summary	9
Chapter 1: Installing the Julia Platform	11
Installing Julia	12
Working with Julia's shell	17
Startup options and Julia scripts	19
Packages	20
Installing and working with Julia Studio	23
Installing and working with IJulia	24
Installing Sublime-IJulia	26
Installing Juno	27
Other editors and IDEs	27
How Julia works	28
Summary	29
Chapter 2: Variables, Types, and Operations	31
Variables, naming conventions, and comments	32
Types	34
Integers	35
Floating point numbers	36
Elementary mathematical functions and operations	37
Rational and complex numbers	38
Characters	39
Strings	39

Table of Contents

Regular expressions	42
Ranges and arrays	44
Dates and times	49
Scope and constants	50
Summary	53
Chapter 3: Functions	55
Defining functions	55
Optional and keyword arguments	58
Anonymous functions	60
First-class functions and closures	60
Recursive functions	63
Map, filter, and list comprehensions	64
Generic functions and multiple dispatch	66
Summary	68
Chapter 4: Control Flow	69
Conditional evaluation	69
Repeated evaluation	71
Exception handling	75
Scope revisited	78
Tasks	80
Summary	82
Chapter 5: Collection Types	83
Matrices	84
Tuples	90
Dictionaries	91
Sets	96
Example project – word frequency	97
Summary	99
Chapter 6: More on Types, Methods, and Modules	101
Type annotations and conversions	102
The type hierarchy – subtypes and supertypes	104
User-defined and composite types	108
Types and collections – inner constructors	113
Type unions	115
Parametric types and methods	116
Standard modules and paths	118
Summary	121

Table of Contents

Chapter 7: Metaprogramming in Julia	123
Expressions and symbols	123
Eval and interpolation	126
Defining macros	127
Built-in macros	131
Reflection capabilities	133
Summary	134
Chapter 8: I/O, Networking, and Parallel Computing	135
Basic input and output	135
Working with files	137
Using DataFrames	142
Working with TCP sockets and servers	148
Interacting with databases	151
Parallel operations and computing	153
Summary	163
Chapter 9: Running External Programs	165
Running shell commands	165
Calling C and FORTRAN	168
Calling Python	169
Performance tips	170
Summary	173
Chapter 10: The Standard Library and Packages	175
Digging deeper into the standard library	175
Julia's package manager	177
Publishing a package	178
Graphics in Julia	180
Using Gadfly on data	181
Summary	183
Appendix: List of Macros and Packages	185
Macros	185
List of packages	186

Module 2: Julia High Performance

Chapter 1: Julia is Fast	191
Julia – fast and dynamic	192
Designed for speed	194
How fast can Julia be?	197
Summary	199

Table of Contents

Chapter 2: Analyzing Julia Performance	201
Timing Julia code	201
The Julia profiler	203
Analyzing memory allocation	207
Statistically accurate benchmarking	208
Summary	209
Chapter 3: Types in Julia	211
The Julia type system	211
Type-stability	218
Kernel methods	229
Types in storage locations	231
Summary	234
Chapter 4: Functions and Macros – Structuring Julia Code for High Performance	235
Using globals	236
Inlining	240
Closures and anonymous functions	245
Using macros for performance	247
Generated functions	253
Summary	256
Chapter 5: Fast Numbers	257
Numbers in Julia	257
Trading performance for accuracy	262
Subnormal numbers	266
Summary	269
Chapter 6: Fast Arrays	271
Array internals in Julia	271
Bound checking	277
Allocations and in-place operations	279
Array views	282
SIMD parallelization	284
Yeppp!	287
Writing generic library functions with arrays	289
Summary	292
Chapter 7: Beyond the Single Processor	293
Parallelism in Julia	293
Programming parallel tasks	296
Shared arrays	300
Summary	301

Module 3: Mastering Julia

Chapter 1: The Julia Environment	305
Introduction	305
Getting started	310
A quick look at some Julia	315
Package management	324
What makes Julia special	330
Summary	332
Chapter 2: Developing in Julia	333
Integers, bits, bytes, and bools	333
Arrays	337
Char and strings	342
Real, complex, and rational numbers	348
Composite types	354
More about matrices	355
Data arrays and data frames	360
Dictionaries, sets, and others	361
Summary	365
Chapter 3: Types and Dispatch	367
Functions	367
Julia's type system	381
Enumerations (revisited)	394
Multiple dispatch	395
Summary	403
Chapter 4: Interoperability	405
Interfacing with other programming environments	405
Metaprogramming	416
Tasks	426
Executing commands	432
Summary	441
Chapter 5: Working with Data	443
Basic I/O	443
Structured datasets	452
DataFrames and RDatasets	463
Statistics	472
Selected topics	477
Summary	488

Table of Contents

Chapter 6: Scientific Programming	489
Linear algebra	490
Signal processing	497
Differential equations	503
Optimization problems	511
Stochastic problems	520
Summary	530
Chapter 7: Graphics	531
Basic graphics in Julia	532
Data visualization	540
Graphic engines	547
Using the Web	555
Raster graphics	559
Summary	565
Chapter 8: Databases	567
A basic view of databases	567
Relational databases	571
NoSQL datastores	590
RESTful interfacing	596
Summary	604
Chapter 9: Networking	605
Sockets and servers	605
Working with the Web	611
Messaging	621
Cloud services	627
Summary	637
Chapter 10: Working with Julia	639
Under the hood	639
Performance tips	646
Developing a package	655
Community groups	662
What's missing?	674
Summary	675
Bibliography	677

Module 1

Getting Started with Julia

Enter the exciting world of Julia, a high-performance language for technical computing

The Rationale for Julia

This introduction will present you with the reasons why Julia is quickly growing in popularity in the technical, data scientist, and high-performance computing arena. We will cover the following topics:

- The scope of Julia
- Julia's place among other programming languages
- A comparison with other languages for the data scientist
- Useful links

The scope of Julia

The core designers and developers of Julia (*Jeff Bezanson, Stefan Karpinski, and Viral Shah*) have made it clear that Julia was born out of a deep frustration with the existing software toolset in the technical computing disciplines. Basically, it boils down to the following dilemma:

- Prototyping is a problem in this domain that needs a high-level, easy-to-use, and flexible language that lets the developer concentrate on the problem itself instead of on low-level details of the language and computation.
- The actual computation of a problem needs maximum performance; a factor of 10 in computation time makes a world of difference (think of one day versus ten days), so the production version often has to be (re)written in C or FORTRAN.

- Before Julia, practitioners had to be satisfied with a "speed for convenience" trade-off, use developer-friendly and expressive, but decades-old interpreted languages such as MATLAB, R, or Python to express the problem at a high level. To program the performance-sensitive parts and speed up the actual computation, people had to resort to statically compiled languages such as C or FORTRAN, or even the assembly code. Mastery on both the levels is not evident: writing high-level code in MATLAB, R, or Python for prototyping on the one hand, and writing code that does the same thing in C, which is used for the actual execution.

Julia was explicitly designed to bridge this gap. It gives you the possibility of writing high-performance code that uses CPU and memory resources as effectively as can be done in C, but working in pure Julia all the way down, reduces the need for a low-level language. This way, you can rapidly iterate using a simple programming model from the problem prototype to near-C performance. The Julia developers have proven that working in one environment that has the expressive capabilities as well as the pure speed is possible using the recent advances in **Low Level Virtual Machine Just in Time (LLVM JIT)** compiler technologies (for more information, see <http://en.wikipedia.org/wiki/LLVM>).

In summary, they designed Julia to have the following specifications:

- Julia is open source and free with a liberal (MIT) license.
- It is designed to be an easy-to-use and learn, elegant, clear and dynamic, interactive language by reducing the development time. To that end, Julia almost looks like the pseudo code with an obvious and familiar mathematical notation; for example, here is the definition for a polynomial function, straight from the code:

```
x -> 7x^3 + 30x^2 + 5x + 42
```

Notice that there is no need to indicate the multiplications.

- It provides the computational power and speed without having to leave the Julia environment.
- Metaprogramming and macro capabilities (due to its homoiconicity (refer to *Chapter 7, Metaprogramming in Julia*), inherited from Lisp), to increase its abstraction power.
- Also, it is usable for general programming purposes, not only in pure computing disciplines.
- It has built-in and simple to use concurrent and parallel capabilities to thrive in the multicore world of today and tomorrow.

Julia unites this all in one environment, something which was thought impossible until now by most researchers and language designers.



The Julia logo

Julia's place among the other programming languages

Julia reconciles and brings together the technologies that before were considered separate, namely:

- The dynamic, untyped, and interpreted languages on the one hand (Python, Ruby, Perl, MATLAB/Octave, R, and so on)
- The statically typed and compiled languages on the other (C, C++, Fortran, and Fortress)

How can Julia have the flexibility of the first and the speed of the second category?

Julia has no static compilation step. The machine code is generated just-in-time by an LLVM-based JIT compiler. This compiler, together with the design of the language, helps Julia to achieve maximal performance for numerical, technical, and scientific computing. The key for the performance is the *type information*, which is gathered by a fully automatic and intelligent *type inference engine*, that deduces the type from the data contained in the variables. Indeed, because Julia has a *dynamic type system*, declaring the type of variables in the code is optional. Indicating types is not necessary, but it can be done to document the code, improve tooling possibilities, or in some cases, to give hints to the compiler to choose a more optimized execution path. This optional typing discipline is an aspect it shares with Dart. Typeless Julia is a valid and useful subset of the language, similar to traditional dynamic languages, but it nevertheless runs at statically compiled speeds. Julia applies *generic programming* and *polymorphic functions* to the limit, writing an algorithm just once and applying it to a broad range of types. This provides common functionality across drastically different types, for example: `size` is a generic function with 50 concrete method implementations. A system called **dynamic multiple dispatch** efficiently picks the optimal method for all of a function's arguments from tens of method definitions. Depending on the actual types very specific and efficient native code implementations of the function are chosen or generated, so its type system lets it align closer with primitive machine operations.



In summary, data flow-based type inference implies multiple dispatch choosing specialized execution code.



However, do keep in mind that types are not statically checked. Exceptions due to type errors can occur at runtime, so thorough testing is mandatory. As to categorizing Julia in the programming language universe, it embodies multiple paradigms, such as procedural, functional, metaprogramming, and also (but not fully) object oriented. It is by no means an exclusively class-based language such as Java, Ruby, or C#. Nevertheless, its *type system* offers a kind of inheritance and is very powerful. Conversions and promotions for numeric and other types are elegant, friendly, and swift, and user-defined types are as fast and compact as built-in types. As for functional programming, Julia makes it very easy to design programs with pure functions and has no side effects; functions are first-class objects, as in mathematics.

Julia also supports a multiprocessing environment based on a message passing model to allow programs to run via multiple processes (local or remote) using distributed arrays, enabling distributed programs based on any of the models for parallel programming.

Julia is equally suited for general programming as is Python. It has as good and modern (Unicode capable) string processing and regular expressions as Perl or other languages. Moreover, it can also be used at the shell level, as a glue language to synchronize the execution of other programs or to manage other processes.

Julia has a standard library written in Julia itself, and a built-in package manager based on GitHub, which is called **Metadata**, to work with a steadily growing collection of external libraries called **packages**. It is *cross platform*, supporting GNU/Linux, Darwin/OS X, Windows, and FreeBSD for both x86/64 (64-bit) and x86 (32-bit) architectures.

A comparison with other languages for the data scientist

Because speed is one of the ultimate targets of Julia, a benchmark comparison with other languages is displayed prominently on the Julia website (<http://julialang.org/>). It shows that Julia's rivals C and Fortran, often stay within a factor of two of fully optimized C code, and leave the traditional dynamic language category far behind. One of Julia's explicit goals is to have sufficiently good performance that you never have to drop down into C. This is in contrast to the following environments, where (even for NumPy) you often have to work with C to get enough performance when moving to production. So, a new era of technical computing can be envisioned, where libraries can be developed in a high-level language instead of in C or FORTRAN. Julia is especially good at running MATLAB and R-style programs. Let's compare them somewhat more in detail.

MATLAB

Julia is instantly familiar to MATLAB users; its syntax strongly resembles that of MATLAB, but Julia aims to be a much more general purpose language than MATLAB. The names of most functions in Julia correspond to the MATLAB/Octave names, and not the R names. Under the covers, however, the way the computations are done, things are extremely different. Julia also has equally powerful capabilities in *linear algebra*, the field where MATLAB is traditionally applied. However, using Julia won't give you the same license fee headaches. Moreover, the benchmarks show that it is from 10 to 1,000 times faster depending on the type of operation, also when compared to Octave (the open source version of MATLAB). Julia provides an interface to the MATLAB language with the package `MATLAB.jl` (<https://github.com/lindahua/MATLAB.jl>).

R

R was until now the chosen development language in the *statistics* domain. Julia proves to be as usable as R in this domain, but again with a performance increase of a factor of 10 to 1,000. Doing statistics in MATLAB is frustrating, as is doing linear algebra in R, but Julia fits both the purposes. Julia has a much richer type system than the vector-based types of R. Some statistics experts such as *Douglas Bates* heavily support and promote Julia as well. Julia provides an interface to the R language with the package `Rif.jl` (<https://github.com/lgautier/Rif.jl>).

Python

Again, Julia has a performance head start of a factor of 10 to 30 times as compared to Python. However, Julia compiles the code that reads like Python into machine code that performs like C. Furthermore, if necessary you can call Python functions from within Julia using the `PyCall` package (<https://github.com/stevengj/PyCall.jl>).

Because of the huge number of existing libraries in all these languages, any practical data scientist can and will need to mix the Julia code with R or Python when the problem at hand demands it.

Julia can also be applied to *data analysis and big data*, because these often involve predictive analysis, modeling problems that can often be reduced to linear algebra algorithms, or graph analysis techniques, all things Julia is good at tackling.

In the field of **High Performance Computing (HPC)**, a language such as Julia has long been lacking. With Julia, domain experts can experiment and quickly and easily express a problem in such a way that they can use modern HPC hardware as easily as a desktop PC. In other words, a language that gets users started quickly without the need to understand the details of the underlying machine architecture is very welcome in this area.

Useful links

The following are the links that can be useful while using Julia:

- The main Julia website can be found at <http://julialang.org/>
- For documentation, refer to <http://docs.julialang.org/en/latest>
- View the packages at <http://pkg.julialang.org/index.html>
- Subscribe to the mailing lists at <http://julialang.org/community/>
- Get support at an IRC channel from <http://webchat.freenode.net/?channels=julia>

Summary

In this introduction, we gave an overview of Julia's characteristics and compared them to the existing languages in its field. Julia's main advantage is its ability to generate specialized code for different input types. When coupled with the compiler's ability to infer these types, this makes it possible to write the Julia code at an abstract level while achieving the efficiency associated with the low-level code. Julia is already quite stable and production ready. The learning curve for Julia is very gentle; the idea being that people who don't care about fancy language features should be able to use it productively too and learn about new features only when they become useful or needed.

1

Installing the Julia Platform

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

- Installing Julia
- Working with Julia's shell
- Start-up options and Julia scripts
- Packages
- Installing and working with Julia Studio
- Installing and working with IJulia
- Installing Sublime-IJulia
- Installing Juno
- Other editors and IDEs
- Working of Julia

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

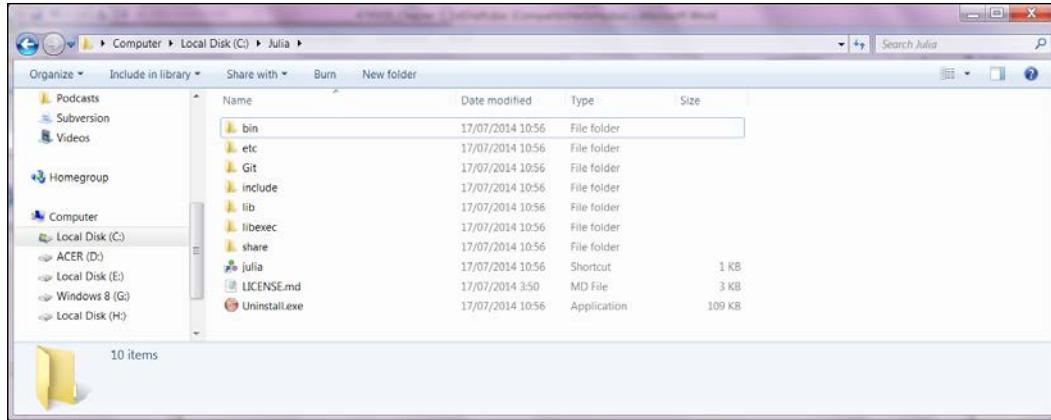
Installing Julia

The Julia platform in binary (that is, executable) form can be downloaded from <http://julialang.org/downloads/>. It exists for three major platforms (Windows, Linux, and OS X) in 32- and 64-bit format, and is delivered as a package or in an archive format. You should use the current official stable release when doing serious professional work with Julia (at the time of writing, this is Version 0.3). If you would like to investigate the latest developments, install the upcoming version (which is now Version 0.4). The previous link contains detailed and platform-specific instructions for the installation. We will not repeat these instructions here completely, but we will summarize some important points.

Windows version – usable from Windows XP SP2 onwards

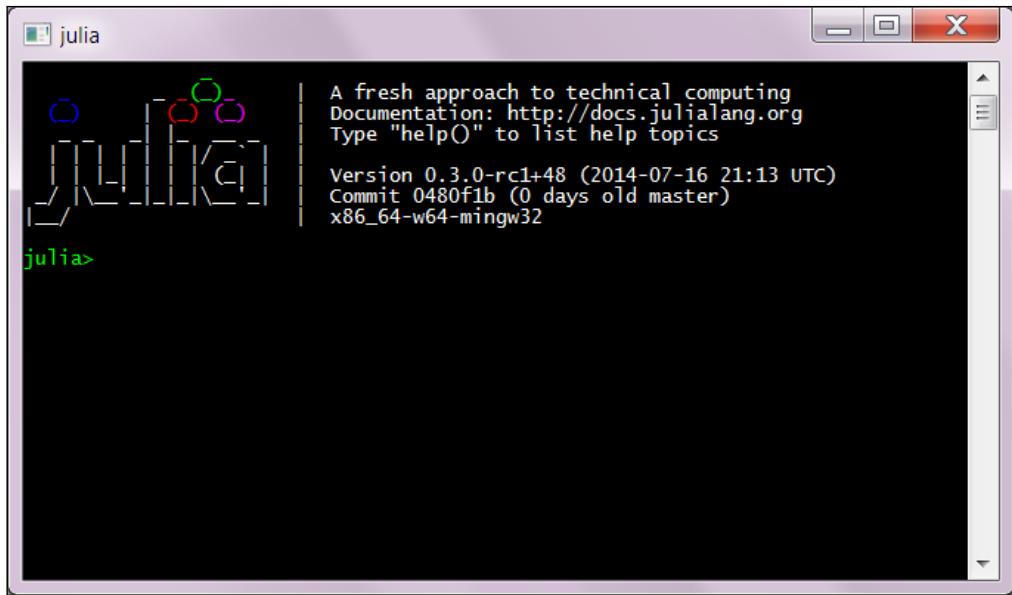
You need to keep the following things in mind if you are using the Windows OS:

1. As a prerequisite, you need the 7zip extractor program, so first download and install <http://www.7-zip.org/download.html>.
2. Now, download the `julia-n.m.p-win64.exe` file to a temporary folder (`n.m.p` is the version number, such as `0.2.1` or `0.3.0`; `win32/win64` are respectively the 32- and 64-bit version; a release candidate file looks like `julia-0.4.0-rc1-nnnnnnnn-win64` (`nnnnnnnn` is a checksum number such as `0480f1b`).
3. Double-click on the file (or right-click, and select **Run as Administrator** if you want Julia installed for all users on the machine). Clicking **OK** on the security dialog message, and then choosing the installation directory (for example, `c:\julia`) will extract the archive into the chosen folder, producing the following directory structure, and taking some 400 MB of disk space:



The Julia folder structure in Windows

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



The Julia REPL

5. On Windows, if you have chosen C:\Julia as your installation directory, this is the C:\Julia\bin\julia.exe file. Add C:\Julia\bin to your PATH variable if you want the REPL to be available on any Command Prompt. The default installation folder on Windows is: C:\Users\UserName\AppData\Local\Julia-n.m.p (where n.m.p is the version number, such as 0.3.2).
6. More information on Julia in the Windows OS can be found at <https://github.com/JuliaLang/julia/blob/master/README.windows.md>.

Ubuntu version

For Ubuntu systems (Version 12.04 or later), there is a **Personal Package Archive (PPA)** for Julia (can be found at <https://launchpad.net/~staticfloat/+archive/ubuntu/juliareleases>) that makes the installation painless. All you need to do to get the stable version is to issue the following commands in a terminal session:

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

If you want to be at the bleeding edge of development, you can download the nightly builds instead of the stable releases. The nightly builds are generally less stable, but will contain the most recent features. To do so, replace the first of the preceding commands with:

```
sudo add-apt-repository ppa:staticfloat/julianightlies
```

This way, you can always upgrade to a more recent version by issuing the following commands:

```
sudo apt-get update  
sudo apt-get upgrade
```

The Julia executable lives in /usr/bin/julia (given by the JULIA_HOME variable or by the which julia command) and the standard library is installed in /usr/share/julia/base, with shared libraries in /usr/lib/x86_64-linux-gnu/Julia.

For other Linux versions, the best way to get Julia running is to build from source (refer to the next section).

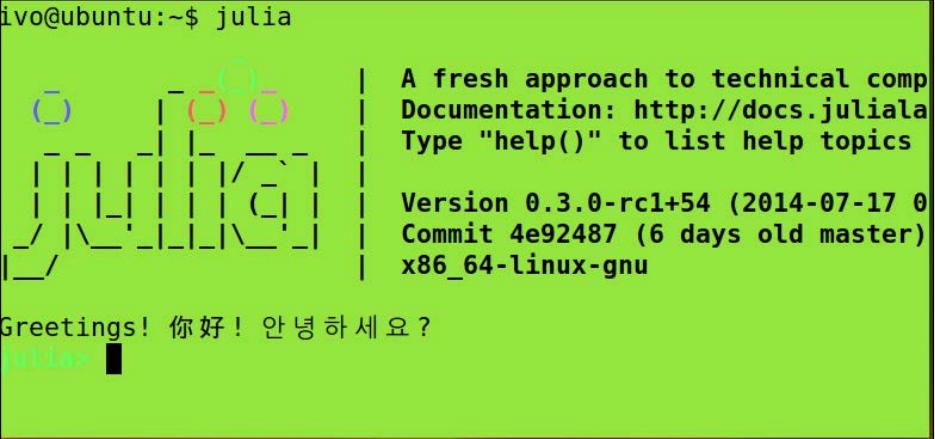
OS X

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

If you want code to be run whenever you start a Julia session, put it in `/home/.juliarc.jl` on Ubuntu, `~/.juliarc.jl` on OS X, or `c:\Users\username\.juliarc.jl` on Windows. For instance, if this file contains the following code:

```
println("Greetings! 你好! 안녕하세요?")
```

Then, Julia starts up in its shell (or REPL as it is usually called) with the following text in the screenshot, which shows its character representation capabilities:



```
ivo@ubuntu:~$ julia
A fresh approach to technical comp
Documentation: http://docs.juliala
Type "help()" to list help topics

Version 0.3.0-rc1+54 (2014-07-17 0
Commit 4e92487 (6 days old master)
x86_64-linux-gnu

Greetings! 你好! 안녕하세요?
julia> ■
```

Using `.juliarc.jl`

Building from source

Perform the following steps to build Julia from source:

1. Download the source code, rather than the binaries, if you intend to contribute to the development of Julia itself, or if no Julia binaries are provided for your operating system or particular computer architecture. Building from source is quite straightforward on Ubuntu, so we will outline the procedure here. The Julia source code can be found on GitHub at <https://github.com/JuliaLang/julia.git>.
2. Compiling these will get you the latest Julia version, not the stable version (if you want the latter, download the binaries, and refer to the previous section).

3. Make sure you have `git` installed; if not, issue the command:

```
sudo apt-get -f install git
```

4. Then, clone the Julia sources with the following command:

```
git clone git://github.com/JuliaLang/julia.git
```

This will download the Julia source code into a `julia` directory in the current folder.

5. The Julia building process needs the GNU compilation tools `g++`, `gfortran`, and `m4`, so make sure that you have installed them with the following command:

```
sudo apt-get install gfortran g++ m4
```

6. Now go to the Julia folder and start the compilation process as follows:

```
cd julia
```

```
make
```

7. After a successful build, Julia starts up with the `./julia` command.

8. Afterwards, if you want to download and compile the newest version, here are the commands to do this in the Julia source directory:

```
git pull
```

```
make clean
```

```
make
```

For more information on how to build Julia on Windows, OS X, and other systems, refer to <https://github.com/JuliaLang/julia/>.



Using parallelization

If you want Julia to use n concurrent processes, compile the source with `make -j n`.

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

Working with Julia's shell

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

```
C:\Users\CVDo\julia
julia> 6 * 7
42
julia> ans
42
julia> 8 * 5;
julia> ans
40
julia> ans + 10
50
```

Working with the REPL (1)

If, for some reason, you don't need to see the result, end the expression with a `;` (semicolon) such as `6 * 7`. In both the cases, the resulting value is stored, for convenience, in a variable named `ans` that can be used in expressions, but only inside the REPL. You can bind a value to a variable by entering an assignment as `a = 3`. Julia is dynamic, and we don't need to enter a type for `a`, but we do need to enter a value for the variable, so that Julia can infer its type. Using a variable `b` that is not bound to the `a` value, results in the `ERROR: b not defined` message. Strings are delineated by double quotes (" "), as in `b = "Julia"`. The following screenshot illustrates these workings with the REPL:

```
julia> a = 3
3
julia> b
ERROR: b not defined
julia> b = "Julia"
"Julia"
julia> b
"Julia"
```

Working with the REPL (2)

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

- To clear or interrupt a current command, press *CTRL + C*
- To clear the screen (but variables are kept in memory), press *CTRL + L*
- To reset the session so that variables are cleared, enter the command `workspace()` in the REPL

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

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

```
Base.println(x)
Base.enumerate(Iter)
Base.cartesianmap(f, dims)
```

Thus, we can see that it is defined in the `Base` module, and is used in two other functions. Different complete expressions on the same line have to be separated by a `;` (semicolon) and only the last result is shown. You can enter multi-line expressions as shown in the following screenshot. If the shell detects that the statement is syntactically incomplete, it will not attempt to evaluate it. Rather, it will wait for the user to enter additional lines until the multi-line statement can be evaluated.

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

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

Julia> ■
```

Working with the REPL (3)

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

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

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

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

The preceding command prints the output as follows:

```
Hello, Julia World!
```

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



You can get more information at <http://docs.julialang.org/en/latest/manual/interacting-with-julia/#key-bindings>.



Startup options and Julia scripts

Without any options, the `julia` command starts up the REPL environment. A useful option to check your environment is `julia -v`. This shows Julia's version, for example, `julia-version 0.3.2+2`. (The `versioninfo()` function in REPL is more detailed, the `VERSION` constant gives you only the version number: `v"0.3.2+2"`). An option that lets you evaluate expressions on the command line itself is `-e`, for example:

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

The preceding commands print out 42 (on Windows, use `"` instead of the `'` character).

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

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

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

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

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

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

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



Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Packages

Most of the standard library in Julia (can be found in /share/julia/base relative to where Julia was installed) is written in Julia itself. The rest of Julia's code ecosystem is contained in packages that are simply Git repositories. They are most often authored by external contributors, and already provide functionality for such diverse disciplines such as bioinformatics, chemistry, cosmology, finance, linguistics, machine learning, mathematics, statistics, and high-performance computing. A searchable package list can be found at <http://pkg.julialang.org/>. Official Julia packages are registered in the METADATA.jl file in the Julia Git repository, available on GitHub at <https://github.com/JuliaLang/METADATA.jl>.

Julia's installation contains a built-in package manager `Pkg` for installing additional Julia packages written in Julia. The downloaded packages are stored in a cache ready to be used by Julia given by `Pkg.dir()`, which are located at `c:\users\username\.julia\vn.m\cache`, `/home/$USER/.julia/vn.m/cache`, or `~/julia/vn.m/cache`. If you want to check which packages are installed, run the `Pkg.status()` command in the Julia REPL, to get a list of packages with their versions, as shown in the following screenshot:

```
julia> Pkg.status()
4 required packages:
- IJulia          0.1.12
- Jewel           0.6.2
- Nettle          0.1.4
- ZMQ             0.1.12
11 additional packages:
- BinDeps         0.2.14
- HTTPClient      0.1.4
- JSON            0.3.7
- Lazy             0.4.1
- LibCURL          0.1.3
- LibExpat         0.0.4
- REPLCompletions 0.0.1
- URIParser        0.0.2
- URLParse         0.0.0
- WinRPM           0.0.14
- zlib             0.1.7
```

Packages list

The `Pkg.installed()` command gives you the same information, but in a dictionary form and is usable in code. Version and dependency management is handled automatically by `Pkg`. Different versions of Julia can coexist with incompatible packages, each version has its own package cache.



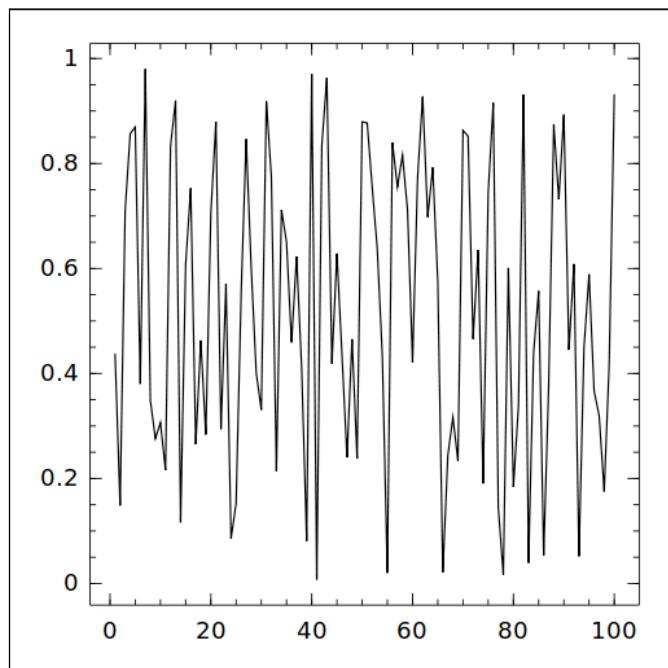
If you get an error with `Pkg.status()` such as `ErrorException("Unable to read directory METADATA.")`, issue a `Pkg.init()` command to create the package repository folders, and clone `METADATA` from Git. If the problem is not easy to find or the cache becomes corrupted somehow, you can just delete the `.julia` folder, enter `Pkg.init()`, and start with an empty cache. Then, add the packages you need.

Adding a new package

Before adding a new package, it is always a good idea to update your package database for the already installed packages with the `Pkg.update()` command. Then, add a new package by issuing the `Pkg.add("PackageName")` command, and execute `using PackageName` in code or in the REPL. For example, to add 2D plotting capabilities, install the Winston package with `Pkg.add("Winston")`. To make a graph of 100 random numbers between 0 and 1, execute the following commands:

```
using Winston  
plot(rand(100))
```

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

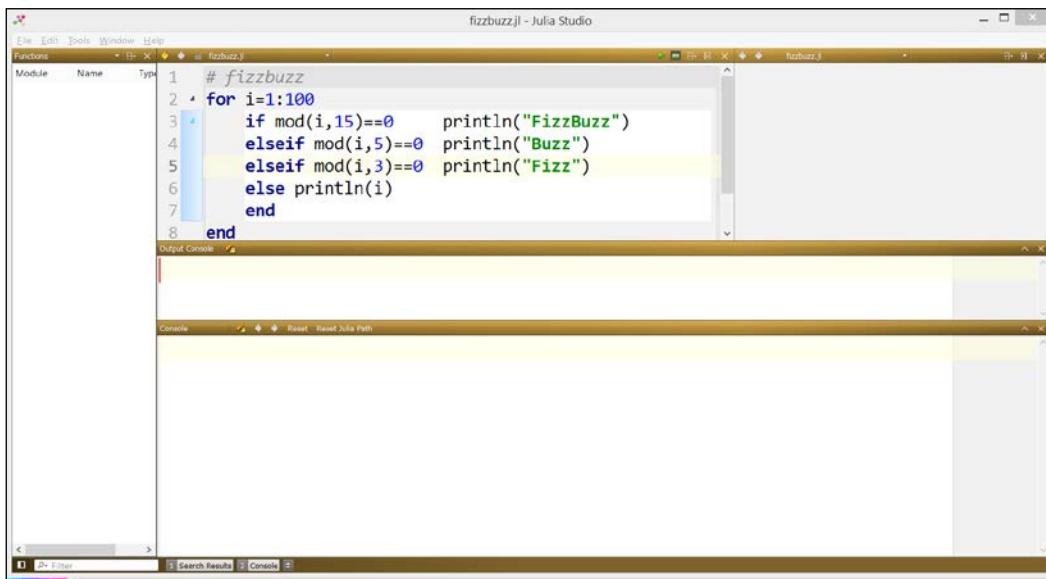


A plot of white noise with Winston

After installing a new Julia version, update all the installed packages by running `Pkg.update()` in the REPL. For more detailed information, you can refer to <http://docs.julialang.org/en/latest/manual/packages/>.

Installing and working with Julia Studio

Julia Studio is a free desktop app for working with Julia that runs on Linux, Windows, and OS X (<http://forio.com/labs/julia-studio/>). It works with the 0.3 release on Windows (Version 0.2.1 for Linux and OS X, at this time, if you want Julia Studio to work with Julia v0.3 on Linux and OS X, you have to do the compilation of the source code of the Studio yourself). It contains a sophisticated editor and integrated REPL, version control with Git, and a very handy side pane with access to the command history, filesystem, packages, and the list of edited documents. It is created by *Forio*, a company that makes software for simulations, data explorations, interactive learning, and predictive analytics. In the following screenshot, you can see some of Julia Studio's features, such as the **Console** section and the green **Run** button (or *F5*) in the upper-right corner. The simple program `fizzbuzz.jl` prints for the first 100 integers for "fizz" if the number is a multiple of 3, "buzz" if a multiple of 5, and "fizzbuzz" if it is a multiple of 15.



Julia Studio

Notice the # sign that indicates the beginning of comments, the elegant and familiar for loop and if elseif construct, and how they are closed with end. The 1:100 range is a range; mod returns the remainder of the division; the function mod(i, n) can also be written as an i % n operator. Using four spaces for indentation is a convention. Recently, Forio also developed Epicenter, a computational platform for hosting the server-side models (also in Julia), and building interactive web interfaces for these models.

Installing and working with IJulia

IJulia (<https://github.com/JuliaLang/IJulia.jl>) is a combination of the IPython web frontend interactive environment (<http://ipython.org/>) with a Julia-language backend. It allows you to work with IPython's powerful graphical notebook (which combines code, formatted text, math, and multimedia in a single document) with qtconsole and regular REPL. Detailed instructions for installation are found at the GitHub page for IJulia (<https://github.com/JuliaLang/IJulia.jl>) and in the *Julia at MIT* notes (<https://github.com/stevengj/julia-mit/blob/master/README.md>). Here is a summary of the steps:

1. Install Version 1.0 or later of IPython via easy_install or pip (on OS X and Windows, this is included in the Anaconda Python installation). On Linux, use apt-get install ipython. (For more information, refer to the IPython home page).
2. Install PyQt4 or PySide for qtconsole.
3. Install the IJulia package from the REPL with Pkg.add("IJulia").
4. Install the PyPlot package with Pkg.add("PyPlot").

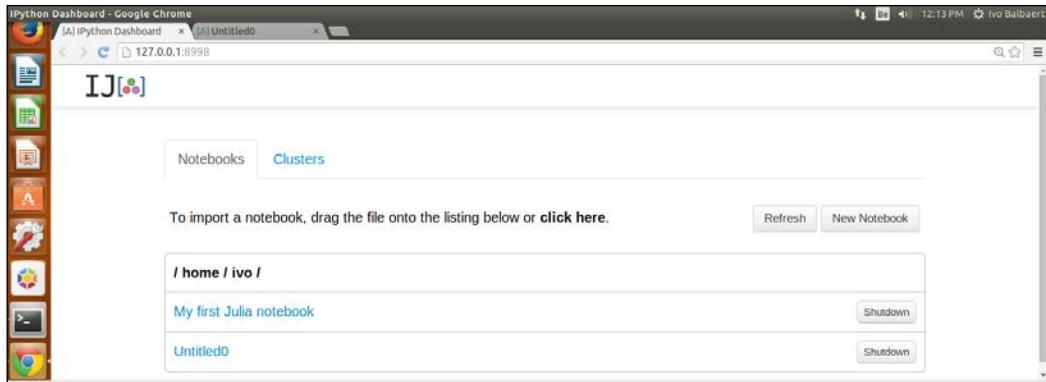
You can work with IJulia in either of two ways:

- Start an IPython notebook in your web browser by typing the following command in a console:

```
ipython notebook --profile julia
```

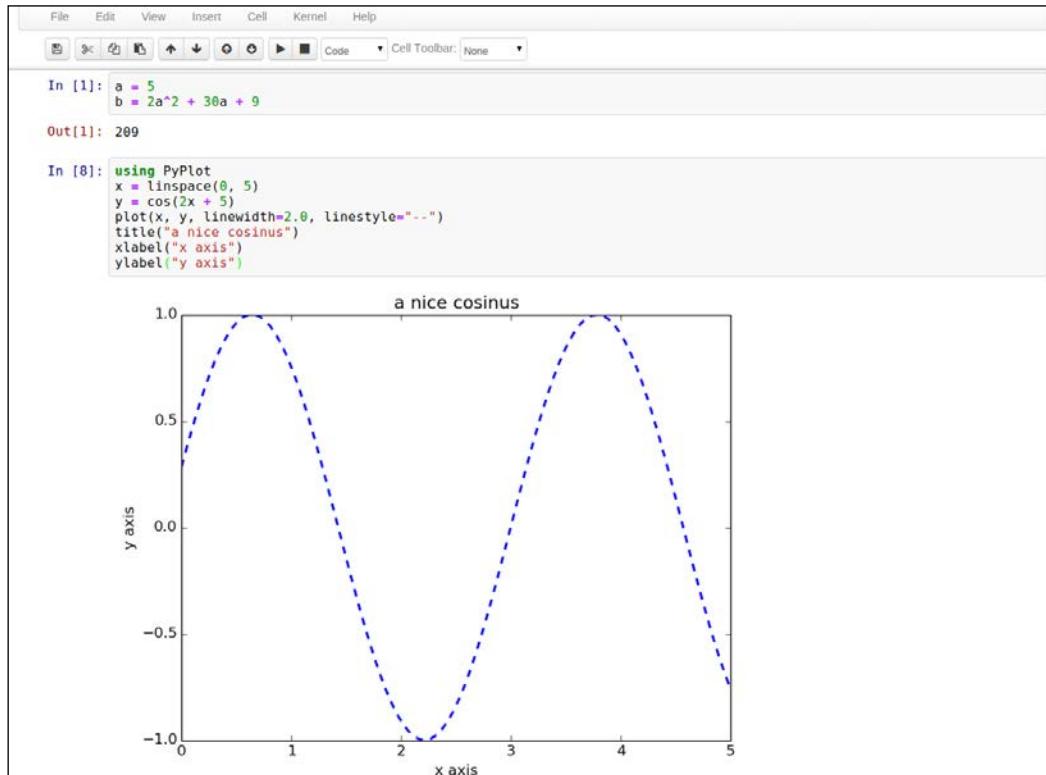
- Start qtconsole with:

```
ipython qtconsole --profile Julia
```



The IJulia dashboard on Ubuntu

Verify that you have started IJulia. You must see IJ and the Julia logo in the upper-left corner of the browser window. Julia code is entered in the input cells (input can be multiline) and then executed with *Shift + Enter*. Here is a small example:



An IJulia session example

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

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

In the second input cell, we use `PyPlot` (this requires the installation of `matplotlib`; for example, on Linux, this is done by `sudo apt-get install python-matplotlib`).

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

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

Save a notebook in file format (with the extension `.ipynb`) by downloading it from the menu. If working in an IPython notebook is new for you, you can take a look at the demo at <http://ipython.org/notebook.html> to get started. After installing a new Julia version, always run `Pkg.build("IJulia")` in the REPL in order to rebuild the IJulia package with this new version.

Installing Sublime-IJulia

The popular Sublime Text editor (<http://www.sublimetext.com/3>) now has a plugin based on IJulia (<https://github.com/quinnj/Sublime-IJulia>) authored by Jacob Quinn. It gives you syntax highlighting, autocompletion, and an in-editor REPL, which you basically just open like any other text file, but it runs Julia code for you. You can also select some code from a code file and send it to the REPL with the shortcut `CTRL + B`, or send the entire file there. Sublime-IJulia provides a frontend to the IJulia backend kernel, so that you can start an IJulia frontend in a Sublime view and interact with the kernel. Here is a summary of the installation, for details you can refer to the preceding URL:

1. From within the Julia REPL, install the `ZMQ` and `IJulia` packages.
2. From within Sublime Text, install the `Package Control` package (<https://sublime.wbond.net/installation>).

3. From within Sublime Text, install the `IJulia` package from the Sublime command palette.
4. *Ctrl + Shift + P* opens up a new IJulia console. Start entering commands, and press *Shift + Enter* to execute them. The *Tab* key provides command completion.

Installing Juno

Another promising IDE for Julia and a work in progress by Mike Innes and Keno Fisher is **Juno**, which is based on the Light Table environment. The docs at <http://junolab.org/docs/installing.html> provides detailed instructions for installing and configuring Juno. Here is a summary of the steps:

1. Get LightTable from <http://lighttable.com>.
2. Start LightTable, install the `Juno` plugin through its plugin manager, and restart LightTable.

Light Table works extensively with a command palette that you can open by typing *Ctrl + SPACE*, entering a command, and then selecting it. Juno provides an integrated console, and you can evaluate single expressions in the code editor directly by typing *Ctrl + Enter* at the end of the line. A complete script is evaluated by typing *Ctrl + Shift + Enter*.

Other editors and IDEs

For terminal users, the available editors are as follows:

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

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

The **SageMath** project (<https://cloud.sagemath.com/>) runs Julia in the cloud within a terminal and lets you work with IPython notebooks. You can also work and teach with Julia in the cloud using the **JuliaBox** platform (<https://juliabox.org/>).

How Julia works

(You can safely skip this section on a first reading.)

Julia works with an LLVM JIT compiler framework that is used for just-in-time generation of machine code. The first time you run a Julia function, it is parsed and the types are inferred. Then, LLVM code is generated by the **JIT (just-in-time)** compiler, which is then optimized and compiled down to native code. The second time you run a Julia function, the native code already generated is called. This is the reason why, the second time you call a function with arguments of a specific type, it takes much less time to run than the first time (keep this in mind when doing benchmarks of Julia code). This generated code can be inspected. Suppose, for example, we have defined a $f(x) = 2x + 5$ function in a REPL session. Julia responds with the message, **f (generic function with 1 method)**; the code is *dynamic* because we didn't have to specify the type of x or f . Functions are by default *generic* because they are ready to work with different data types for their variables. The `code_llvm` function can be used to see the JIT bytecode, for example, the version where the x argument is of type `Int64`:

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

define i64 @"julia_f;1065"(i64) {
top:
    %1 = shl i64 %0, 1, !dbg !3248
    %2 = add i64 %1, 5, !dbg !3248
    ret i64 %2, !dbg !3248
}
```

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

```
julia> code_native(f, (Int64,))
.text
Filename: none
Source line: 1
    push    RBP
    mov     RBP, RSP
Source line: 1
    lea     RAX, QWORD PTR [RCX + RCX + 5]
    pop    RBP
    ret
```

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

```
julia> code_native(f, (Float64,))
.text
Filename: none
Source line: 1
    push    RBP
    mov     RBP, RSP
Source line: 1
    vaddsd XMM0, XMM0, XMM0
    movabs RAX, 48532256
    vaddsd XMM0, XMM0, QWORD PTR [RAX]
    pop    RBP
    ret
```

Julia code is fast because it generates specialized versions of functions for each data type. Julia implements automatic memory management. The user doesn't have to worry about allocating and keeping track of the memory for specific objects. Automatic deletion of objects that are not needed any more (and hence, reclamation of the memory associated with those objects) is done using a garbage collector (GC). The garbage collector runs at the same time as your program. Exactly when a specific object is garbage collected is unpredictable. In Version 0.3, the GC is a simple mark-and-sweep garbage collector; this will change to an incremental mark-and-sweep GC in Version 0.4. You can start garbage collection yourself by calling `gc()`, or if it runs in the way you can disable it by calling `gc_disable()`.

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

Summary

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

2

Variables, Types, and Operations

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

The following topics are covered in this chapter:

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

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

Variables, naming conventions, and comments

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

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

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

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

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

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

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

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

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

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

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

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

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

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

In a REPL, the value of the last expression is automatically displayed each time a statement is evaluated (unless it ends with a ; sign). In a standalone script, Julia will not display anything unless the script specifically instructs it to. This is achieved with a `print` or `println` statement. To display any object in the way the REPL does in code, use `display(object)`.

Types

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

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

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

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

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

```
julia> (2+3)::ASCIIString
ERROR: typeassert: expected ASCIIString, got Int64
```

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

A lot of types exist, in fact, a whole type hierarchy is built in in Julia. If you don't specify the type of a function argument, it has the type `Any`, which is effectively the root or parent of all types. Every object is at least of the universal type `Any`. At the other end of the spectrum, there is type `None` that has no values. No object can have this type, but it is a subtype of every other type. While running the code, Julia will infer the type of the parameters passed in a function, and with this information, it will generate optimal machine code.

You can define your own custom types as well, for instance, a `Person` type. By convention, the names of types begin with a capital letter, and if necessary, the word separation is shown with CamelCase, such as `BigFloat` or `AbstractArray`.

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

Everything in Julia has a type, including types themselves, which are of type `DataType`: `typeof(Int64)` returns `DataType`. Conversion of a variable `var` to a type `Type1` can be done using the type name (lower-cased) as a function `type1(var)`, for example, `int64(3.14)` returns `3`.

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

```
julia> int64("hello")
ERROR: invalid base 10 digit 'h' in "hello"
```

Integers

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

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

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

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

For computations needing arbitrary-precision integers, Julia has a `BigInt` type. These values can be constructed as `BigInt("number")`, and support the same operators as normal integers. Conversions between numeric types are automatic, but not between the primitive types and the `Big-` types. The normal operations of addition (+), subtraction (-), and multiplication (*) apply for integers. A division (/) always gives a floating point number. If you only want integer divisor and remainder, use `div` and `rem`. The symbol ^ is used to obtain the power of a number. The logical values, `true` and `false`, of type `Bool` are also integers with 8 bits. 0 amounts to `false`, and 1 (in fact, also all values other than 0) to `true`; for example, `bool(-56)` returns `true`. Negation can be done with the ! operator; for example, `!true` is `false`. Comparing numbers with == (equal), != or < and > return a `Bool` value, and comparisons can be chained after one another (as in `0 < x < 3`).

Floating point numbers

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

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

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

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

```
julia> 0.1 + 0.2
0.30000000000000004
```

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

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

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

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

Elementary mathematical functions and operations

You can view the binary representation of any number (integer or float) with the `bits` function, for example, `bits(3)` returns "00000000000000000000000000000000".

To round a number, use the `round()` or `iround()` functions: the first returns a floating point number, and the last returns an integer. All standard mathematical functions are provided, such as `sqrt()`, `cbrt()`, `exp()`, `log()`, `sin()`, `cos()`, `tan()`, `erf()` (the error function), and many more (refer to the following URL). To generate a random number, use `rand()`.

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

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

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

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

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

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

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

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

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

Rational and complex numbers

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

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

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

If `a` and `b` are two variables that contain a number, use `complex(a, b)` to form a complex number with them. Rational numbers are useful when you want to work with exact ratios of integers, for example, `3//4`, which is of type `Rational{Int64}`. Again, comparisons and standard operations are defined: `float()` converts to a floating point number, and `num()` and `den()` gives the numerator and denominator. Both types work together seamlessly with all the other numeric types.

Characters

Like C or Java, but unlike Python, Julia implements a type for a single character, the `Char` type. A character literal is written as '`A`', `typeof('A')` returns `Char`. A `Char` type is, in fact, a 32-bit integer whose numeric value is a Unicode code point, and they range from '`\0`' to '`\Ufffffff`'. Convert this to its code point with `int()`: `int('A')` returns 65, `int('α')` returns 945, so this takes two bytes.

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

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

Strings

Literal strings are always ASCII (if they only contain ASCII letters) or UTF8 (if they contain characters that cannot be represented in ASCII), as in this example:

```
julia> typeof("hello")
ASCIIString
julia> typeof("Güdrun")
UTF8String
```

UTF16 and UTF32 are also supported. Strings are contained in double quotes (" ") or triple quotes ('''). They are immutable, which means that they cannot be altered once they have been defined:

```
julia> s = "Hello, Julia"
julia> s[2] = "z"
ERROR: 'setindex!' has no method matching setindex!...
```

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

Using an index less than one or greater than the index of the last byte gives `BoundsError`. In general, strings can contain Unicode characters, which can take up to four bytes, so not every index is a valid character index. For example, for `str2 = "I am the α: the beginning"`, we have `str2[10]` that returns '`\u3b1`' (the two-byte character representing α), `str2[11]` returns `ERROR: invalid UTF-8 character index` (because this is the second byte of the α character) and `str2[12]` returns colon `(:)`.

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

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

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

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

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

Strings prefixed with `:` are of type `Symbol`, such as `:green`; we already used it in the `print_with_color` function. They are more efficient than strings and are used for IDs or keys. Symbols cannot be concatenated. They should only be used if they are expected to remain constant over the course of the execution of the program. The `String` type is very rich, and it has 354 functions defined on it, given by `methodswith(String)`. Some useful methods include:

- `search(string, char)`: This returns the index of the first matching char in `string`, or the range of a substring: `search("Julia", 'l')` returns 3.
- `replace(string, str1, str2)`: This changes substrings `str1` to `str2` in `string`, for example, `replace("Julia", "u", "o")` returns `"Jolia"`.

- `split(string, char or [chars]):` This splits a string on the specified char or chars, for example, `split("34,Tom Jones,Pickwick Street 10,Aberdeen", ',')` returns the four strings in an array: `["34", "Tom Jones", "Pickwick Street 10", "Aberdeen"]`; if char is not specified, the split is done on space characters (spaces, tabs, newlines, and so on).

Formatting numbers and strings

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

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

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

The following `chapter 2\formatting.jl` script shows the most common formats (`show` is a basic function that prints a text representation of an object, often more specific than `print`):

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

The following output is obtained on running the preceding script:

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

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

```
if v"0.3" <= VERSION < v"0.4-"
# do something specific to 0.3 release series
end
```

Regular expressions

To search for and match patterns in text and other data, regular expressions are an indispensable tool for the data scientist. Julia adheres to the Perl syntax of regular expressions. For a complete reference, refer to <http://www.regular-expressions.info/reference.html>. Regular expressions are represented in Julia as a double (or triple) quoted string preceded by `r`, such as `r"..."` (optionally, followed by one or more of the `i`, `s`, `m`, or `x` flags), and they are of type `Regex`. The chapter `2\regexp.jl` script shows some examples.

In the first example, we will match the e-mail addresses (#> shows the result):

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

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

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

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

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

The `RegexMatch` object has the following properties:

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

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

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

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

```
email_pattern = r"(.+)@(.+)"
input = "john.doe@mit.edu"
m = match(email_pattern, input)
println(m.captures) #> ["john.doe", "mit.edu"]
```

Here is another example:

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

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

```
str = "The sky is blue"
reg = r"\w\{3,\}" # matches words of 3 chars or more
r = matchall(reg, str)
show(r) #> ["The", "sky", "blue"]
iter = eachmatch(reg, str)
for i in iter
    println("\\"$(i.match)\\" ")
end
```

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

Ranges and arrays

When we execute `search("Julia", "uli")`, the result is not an index number, but a range `2:4` that indicates the index interval for the searched substring. This comes in handy when you have to work with an interval of numbers, for example, one up to thousand: `1:1000`. The type of this object `typeof(1:1000)` is `UnitRange{Int64}`. By default, the step is `1`, but this can also be specified as the second number; `0:5:100` gives all multiples of 5 up to 100. You can iterate over a range as follows:

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

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

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

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

Julia's arrays are very efficient, powerful, and flexible. The general type format for an array is `Array{Type, n}`, with `n` number of dimensions (we will discuss multidimensional arrays or matrices in *Chapter 5, Collection Types*). As with the complex type, we can see that the `Array` type is generic, and all the elements have to be of the same type. A one-dimensional array (also called a **Vector** in Julia) can be initialized by separating its values by commas and enclosing them in square brackets, for example, `arr = [100, 25, 37]` is a 3-element `Array{Int64,1}`; the type is automatically inferred with this notation. If you want explicitly the type to be `Any`, then define it as follows: `arra = Any[100, 25, "ABC"]`. Notice that we don't have to indicate the number of elements. Julia takes care of that and lets an array grow dynamically when needed.

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

```
arr2 = Array(Int64,5) # is a 5-element Array{Int64,1}  
show(arr2) #> [0,0,0,0,0]
```

When making an array like this, you cannot be sure that it will be initialized to 0 values (refer to the *Other ways to create arrays* section to learn how to initialize an array).

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

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

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

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

Arrays can also be initialized from a range:

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

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

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

```
arr[6] #> ERROR: BoundsError()
```

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

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

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

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

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

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

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

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

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

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

Other ways to create arrays

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

`linspace(start, stop, n)` creates a vector of `n` equally spaced numbers from `start` to `stop`, for example:

```
eqa = linspace(0, 10, 5)
show(eqa) #> [0.0, 2.5, 5.0, 7.5, 10.0]
```

You can use `cell` to create an array with undefined values: `cell(4)` creates a four element array `{Any,1}` with four `#undef` values as shown:

```
{#undef, #undef, #undef, #undef}
```

To fill an array `arr` with the same value for all the elements, use `fill!(arr, 42)`, which returns `[42, 42, 42]`. To create a five-element array with random `Int32` numbers, execute the following:

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

To convert this to an array of `Int64` elements, just execute `int64(v1)`.

Some common functions for arrays

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

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

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



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

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

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

If you want to do the same operations on the front of the array, use `shift!` and `unshift!:`

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

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

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

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

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

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

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

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

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

If a dot (.) precedes operators such as `+` or `*`, the operation is done element wise, that is, on the corresponding elements of the arrays. For example, if `a1 = [1, 2, 3]` and `a2 = [4, 5, 6]`, then `a1 .* a2` returns the array `[4, 10, 18]`. On the other hand, if you want the dot (or scalar) product of vectors, use the `dot(a1, a2)` function, which returns 32, so `dot(a1, a2)` gives the same result as `sum(a1 .* a2)`.

Other functions such as `sin()` simply work by applying the operation to each element, for example, `sin(arr)`. Lots of other useful methods exist, such as `repeat([1, 2, 3], inner = [2])`, which produces `[1, 1, 2, 2, 3, 3]`.

The `methodswith(Array)` function returns 358 methods. You can use `help` in the REPL or search the documentation for more information.

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

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

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

This happens because they point to the same object in memory. If you don't want this, you have to make a copy of the array. Just use `b = copy(a)` or `b = deepcopy(a)` if some elements of `a` are arrays that have to be copied recursively.

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

```
a = [1,2,3]
function change_array(arr)
    arr[2] = 25
end
change_array(a)
println(a) #> [ 1, 25, 3]
```

How to convert an array of chars to a string

Suppose you have an array, `arr = ['a', 'b', 'c']`. Which function on `arr` do we need to return all characters in one string? The function `join` will do the trick: `join(arr)` returns the string "abc"; `utf32(arr)` does the same thing.

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

Dates and times

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

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

Most useful function is `strftime(time())` that returns a string in "22/08/2014 17:06:13" format.

If you want more functionality greater than equal to 0.3 when working in Julia, take a look at the `Dates` package. Add this to the environment with `Pkg.add("Dates")` (it provides a subset of the functionality of the `Dates` module mentioned next). There is also the `Time` package by Quinn Jones. Take a look at the docs to see how to use it (<https://github.com/quinnj/Datetime.jl/wiki/Datetime-Manual>).

Starting from Julia Version 0.4, you should use the `Dates` module built into the standard library, with `Date` for days and `DateTime` for times down to milliseconds. Additional time zone functionality can be added through the `Timezones.jl` package.

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

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

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

Scope and constants

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

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

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

```
# code in chapter 2\scope.jl
x = 1.0 # x is Float64
x = 1 # now x is Int
# y::Float64 = 1.0 # LoadError: "y is not defined"
```

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

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

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

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

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

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

```
x = (a = 5; 2 * a)
```

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

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

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

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

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

```
julia> GC = 3.14
      Warning: redefining constant GC
julia> GC = 10
      ERROR: invalid redefinition of constant GC
```

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

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

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

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

```
# a newspaper headline:
str = "The Gold and Blue Loses a Bit of Its Luster"
println(str)
nchars = length(str)
println("The headline counts $nchars characters") # 43
str2 = replace(str, "Blue", "Red")
# strings are immutable
println(str) # The Gold and Blue Loses a Bit of Its Luster
println(str2)
println("Here are the characters at position 25 to 30:")
subs = str[25:30]
print("-$(lowercase(subs))-") # "-a bit -"
println("Here are all the characters:")
for c in str
    println(c)
```

```
end
arr = split(str, ' ')
show(arr)
#[ "The", "Gold", "and", "Blue", "Loses", "a", "Bit", "of", "Its", "Luster"]
nwords = length(arr)
println("The headline counts $nwords words") # 10
println("Here are all the words:")
for word in arr
    println(word)
end
arr[4] = "Red"
show(arr) # arrays are mutable
println("Convert back to a sentence:")
nstr = join(arr, ' ')
println(nstr) # The Gold and Red Loses a Bit of Its Luster

# working with arrays:
println("arrays: calculate sum, mean and standard deviation ")
arr = [1:100]
typeof(arr) #>
println(sum(arr)) #> 5050
println(mean(arr)) #> 50.5
println(std(arr)) #> 29.011491975882016
```

Summary

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

3

Functions

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

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

Defining functions

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

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

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

Here is a simple example:

```
# code in chapter 3\functions101.jl
function mult(x, y)
    println("x is $x and y is $y")
    return x * y
end
```

Function names such as `mult` are by convention in lower case, without underscores. They can contain Unicode characters, which are useful in mathematical notations. The `return` keyword in the last line is optional; we could have written the line as `x * y`. In general, the value of the last expression in the function is returned, but writing `return` is mostly a good idea in multi-line functions to increase the readability.

The function is called with `n = mult(3, 4)` returns 12, and assigns the `return` value to a new variable `n`. You can also execute a function just by calling `fname(arglist)` if you only need its side effects (that is, how the function affects the program state; for instance, by changing the global variables). The `return` keyword can also be used within a condition in other parts of the function body to exit the function earlier, as in this example:

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

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

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

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

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

We can also have a variable with number of arguments using so-called the `varargs` function. They are coded as follows:

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

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

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

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

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

For example, consider the following code:

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

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

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

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

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

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

There is also a more compact, one-line function syntax (the assignment form) for short functions, for example, `mult(x, y) = x * y`. Use this preferably for simple one-line functions, as it will lend the code greater clarity. Because of this, mathematical functions can also be written in an intuitive form: `f(x, y) = x^3 - y + x * y; f(3, 2) #=> 31`.

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

Optional and keyword arguments

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

```
# code in chapter 3\arguments.jl:  
f(a, b = 5) = a + b
```

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

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

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

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

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

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

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

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

Calling this with `varargs2(k1="name1", k2="name2", k3=7)` returns a 3-element `Array{Any,1}` with the elements: `(:k1, "name1")` `(:k2, "name2")` `(:k3, 7)`. Now, `args` is a collection of the `(key, value)` tuples, where each key comes from the name of the keyword argument, and it is also a symbol (refer to the *Strings* section of *Chapter 2, Variables, Types, and Operations*) because of the colon (`:`) as prefix.

Anonymous functions

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

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

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

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

First-class functions and closures

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

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

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

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

```
(anonymous function)
julia> plustwo(3)
5
```

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

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

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

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

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

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

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

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

Functions

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

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

We can assign the returned functions to variables:

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

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

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

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

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

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

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

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

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

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

Recursive functions

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

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

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

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

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

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

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

When using recursion, care should be taken to define a base case to stop the calculation. Also, although Julia can nest very deep, watch out for stack overflow because until now, Julia does not do tail call optimization automatically. If you run into this problem, *Zach Allau*n suggests a nice workaround in the blog at <http://blog.zachallaun.com/post/jumping-julia>.

Map, filter, and list comprehensions

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

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

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

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

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

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

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

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

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

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

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

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

2	3	4
3	4	5
- `table10 = [x * y for x=1:10, y=1:10]` creates a 10 x 10 Array{Int64,2}, and returns the multiplication table of 10
- `arrany = Any[i * 2 for i in 1:5]` creates 5-element Array{Any,1} with elements 2, 4, 6, 8, and 10

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

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

Generic functions and multiple dispatch

We already saw that functions are inherently defined as generic, that is, they can be used for different types of their arguments. The compiler will generate a separate version of the function each time it is called with arguments of a new type. A concrete version of a function for a specific combination of argument types is called a **method** in Julia. To define a new method for a function (also called **overloading**), just use the same function name but a different signature, that is, with different argument types. A list of all the methods is stored in a virtual method table (`vtable`) on the function itself; methods do not belong to a particular type. When a function is called, Julia will do a lookup in that `vtable` at runtime to find which concrete method it should call based on the types of all its arguments; this is Julia's mechanism of **multiple dispatch**, which neither Python, nor C++ or Fortran implements. It allows open extensions where normal object-oriented code would have forced you to change a class or subclass an existing class and thus change your library. Note that only the positional arguments are taken into account for multiple dispatch, and not the keyword arguments.

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

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

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

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

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

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

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

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

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

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

To get a quick overview of all the versions of a function, type `methods(fname)` in the REPL. For example, `methods(+)` shows a listing of 149 methods for a generic function `+`:

```
+ (x::Bool) at bool.jl:36
+ (x::Bool, y::Bool) at bool.jl:39
...
+ (a,b,c) at operators.jl:82
+ (a,b,c, xs...) at operators.jl:83
```

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

As a second example, `methods(sort)` shows # 4 methods for generic function "sort".

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

```
354-element Array{Method,1}:
 write(io::IO,s::String) at string.jl:68
 download(url::String,filename::String) at interactiveutil.jl:322
 ...

```

In the source for the `write` method, it is defined as follows:

```
write(io::IO, s::String) = for c in s write(io, c) end
```

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

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

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

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

In Julia, the first call of a method invokes the **Low Level Virtual Machine Just In Time (LLVM JIT)** compiler backend (refer to the *How Julia works* section in *Chapter 1, Installing the Julia Platform*), to emit machine code for it, so this *warm-up call* will take a bit longer. Start timing or benchmarking from the second call onwards, after doing a dry run.

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

Summary

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

4

Control Flow

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

We will cover the following topics:

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

Conditional evaluation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Julia has no switch/case statement, and the language provides no built-in pattern matching (although one can argue that multiple dispatch is a kind of pattern matching, which is based not on value, but on type). If you need pattern matching, take a look at the `PatternDispatch` and `Match` packages that provide this functionality.

Repeated evaluation

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

The `for` loop

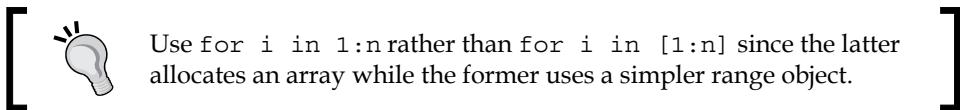
We already encountered the `for` loop when iterating over the elements `e` of a collection `coll` (refer to the *Strings and Ranges and Arrays* sections in *Chapter 2, Variables, Types, and Operations*). This takes the general form:

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

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

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

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



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

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

A more elegant way to accomplish this is using the `enumerate` function as follows:

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

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

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

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

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

The while loop

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

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

In the body of the loop, something has to change the value of `a` so that the initial condition becomes false and the loop ends. If the initial condition is false at the start, the body of the `while` loop is never executed.

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

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

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

The break statement

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

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

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

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

A possible output might be as follows:

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

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

The continue statement

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

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

This prints out, 1 2 7 8 9 10, skipping the numbers 3 to 6, using a chained comparison.

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

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

Exception handling

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

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

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

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

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

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

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

Some other useful functions that do not cause the program flow to stop and that can help you with testing and debugging are as follows:

- `warn("Something is not right here")`: This prints or writes to the standard error output (in red in the REPL), `WARNING: Something is not right here`
- `info("Did you know this?")`: This prints (in blue in the REPL), `INFO: Did you know this?`

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

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

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

Here is an example:

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

This example prints the output as follows:

```
ErrorException
array must be non-empty
```

Popping an empty array generates an exception (as does `push!(a, 1)`, but then this is because `a` is not typed). The variable, `ex`, contains the exception object, but a plain `catch` without a variable can also be used. The `showerror` function is a handy function; its first argument can be any I/O stream, so it could be a file.

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

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

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

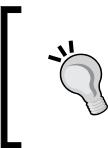
```
ret = try
  a = 4 * 2
  catch ex
  end
```

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

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

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

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



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

Scope revisited

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

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

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

The following example will clarify this as follows:

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

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

This prints out the following result:

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

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

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

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

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



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



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

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

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

```
anon = cell(2)
for i = 1:2
    let i = i
        anon[i] = () -> println(i)
    end
    i += 1
end
```

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

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

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

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

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

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

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

Tasks

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

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

```
# code in Chapter 4\tasks.jl
function fib_producer(n)
    a, b = (0, 1)
    for i = 1:n
        produce(b)
        a, b = (b, a + b)
    end
end
```

If you call this function as `fib_producer(5)`, it waits indefinitely. Instead you have to envelop it as a task that takes a function with no arguments:

```
tsk1 = Task( () -> fib_producer(10) )
```

This gives the following output as `Task (runnable) @0x000000005696180`. The tasks' state is runnable. To get the Fibonacci numbers, start consuming them until nothing returns, and the task is finished (state is done):

```
consume(tsk1) #=> 1
consume(tsk1) #=> 1
consume(tsk1) #=> 2
consume(tsk1) #=> 3
consume(tsk1) #=> 5
consume(tsk1) #=> 8
consume(tsk1) #=> 13
consume(tsk1) #=> 21
consume(tsk1) #=> 34
consume(tsk1) #=> 55
consume(tsk1) #=> nothing # Task (done) @0x000000005696180
```

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

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

```
for n in tsk1
    println(n)
end
```

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

The Task constructor argument must be a function with 0 arguments, that's why it is written as an anonymous function, `() -> fib_producer(10)`.

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

```
tsk1 = @task fib_producer(10)
```

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

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

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

Summary

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

5

Collection Types

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

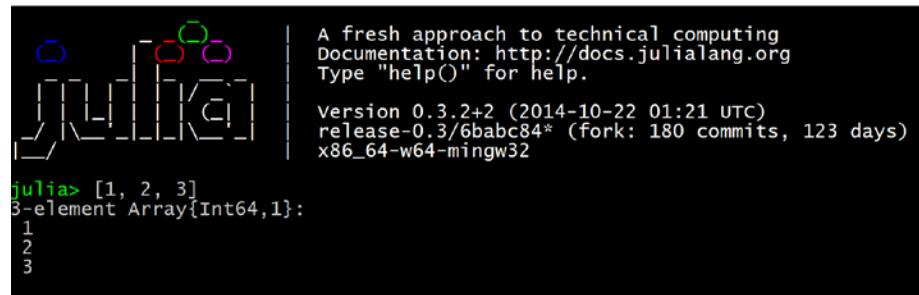
Collections are also iterable types, the types over which we can loop with `for` or an iterator producing each element of the collection successively. The iterable types include string, range, array, tuple, dict, and set.

So, the following are the topics for this chapter:

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

Matrices

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



A screenshot of a terminal window showing a Julia session. The terminal has a black background with white text. At the top, there is some decorative text and version information. Below that, a command is entered and its output is displayed.

```
A fresh approach to technical computing
Documentation: http://docs.julialang.org
Type "help()" for help.

version 0.3.2+2 (2014-10-22 01:21 UTC)
release-0.3/6babcc84* (fork: 180 commits, 123 days)
x86_64-w64-mingw32

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

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

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

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

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

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

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

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

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

Using the same notation, one can calculate products of matrices such as `[1 2] * [3 ; 4]` is calculated as `[1 2] * [3 4]`, which returns the value 11 (which is equal to $1*3 + 2*4$).

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

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

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

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

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

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

The number of columns (5) is given by:

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

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

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

```
idm = eye(3)
3x3 Array{Float64,2}:
1.0  0.0  0.0
0.0  1.0  0.0
0.0  0.0  1.0
```

You can easily work with parts of a matrix, known as *slices* that are similar to those used in Python and NumPy as follows:

- `idm[1:end, 2]` or shorter `idm[:, 2]` returns the entire second column
- `idm[2, :]` returns the entire second row
- `idmc = idm[2:end, 2:end]` returns the output as follows:
`2x2 Array{Float64,2}`
1.0 0.0
0.0 1.0
- `idm[2, :] = 0` sets the entire second row to 0
- `idm[2:end, 2:end] = [5 7 ; 9 11]` will change the entire matrix as follows:

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

All these slicing operations return copies of the original matrix in Julia v0.3. For instance, a change to `idmc` from the previous example will not change `idm`. To obtain a view of the matrix `idm`, rather than a copy, use the `sub` function (see `?sub` for details). From v0.4 onwards, slicing will create views into the original array rather than copying the data.

To make an array of arrays (a *jagged array*), use `jarr = fill(Array{Int64,1}, 3)` and then start initializing every element as an array, for example:

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

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

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

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

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

```
5      11
11     25
```

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

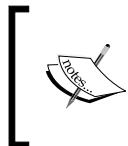
```
1      6
6     16
```

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

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

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

```
1.0   0.0
0.0   1.0
```



Trying to take the inverse of a singular matrix (a matrix that does not have a well-defined inverse) will result in `LAPACKException` or `SingularException`, depending on the matrix type.



Suppose you want to solve the `ma1 * X = ma2` equation, where `ma1`, `X`, and `ma2` are matrices. The obvious solution is `X = inv(ma1) * ma2`. However, this is actually not that good. It is better to use the built-in solver, where `X = ma1 \ ma2`. If you have to solve the `X * ma1 = ma2` equation, use the solution `X = ma2 / ma1`. The solutions that use `/` and `\` are much more numerically stable and also much faster.

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

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

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

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

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

a	B	c = [a b]	c = [a; b]	c = [a, b]
1 2	5 6	1 2 5 6	1 2	1 2
3 4	7 8	3 4 7 8	3 4 5 6 7 8	3 4 5 6 7 8

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

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

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

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

```
0.355623
0.967591
0.809577

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

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

The following example makes this clear:

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

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

As to performance, there is a consensus that using fixed-size arrays can offer a real speed boost. If you know the size, your array `arr` will reach from the start (say 75), then indicate this with `sizehint` to the compiler so that the allocation can be optimized as follows:

```
sizehint(arr, 75) (from v0.4 onwards use sizehint!(arr, 75))
```

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

Tuples

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

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

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

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

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

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

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

```
typeof(t1) #> (Int64,Float64,ASCIIString,Char)
```

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

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

```
t3 = (5, 6, 7, 8)
t3[1] #> 5
t3[end] #> 8
t3[2:3] #> (6, 7)
t3[5] #> BoundsError
t3[3] = 9 #> Error: 'setindex' has no matching ...
author = ("Ivo", "Balbaert", 59)
author[2] #> "Balbaert"
```

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

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

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

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

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

Dictionaries

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

```
// code in Chapter 5\dicts.jl:
d1 = [1 => 4.2, 2 => 5.3]
```

It returns `Dict{Int64,Float64}` with 2 entries: `2 => 5.3` `1 => 4.2`, so there are two key-value tuples here, `(1, 4.2)` and `(2, 5.3)`; the key appears before the `=>` symbol and the value appears after it, and the tuples are separated by commas. The `[]` indicates a typed dictionary; all the keys must have the same type, and the same is true for the values. A dynamic version of a dictionary can be defined with `{ }`:

- `d1 = {1 => 4.2, 2 => 5.3} is Dict{Any,Any}`
- `d2 = {"a" => 1, (2,3) => true} is Dict{Any,Any}`

Any is also inferred when a common type among the keys or values cannot be detected. In general, dictionaries that have type `{Any, Any}` tend to lead to lower performance since the JIT compiler does not know the exact type of the elements. Dictionaries used in performance-critical parts of the code should therefore be explicitly typed. Notice that the (key, value) pairs are not returned (or stored) in the key order. If the keys are of type `Char` or `String`, you can also use `Symbol` as the key type, which could be more appropriate since `Symbols` are immutable. For example, `d3 = [:A => 100, :B => 200]`, which is `Dict{Symbol, Int64}`.

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

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

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

To test if a (key, value) tuple is present, you can use `in` as follows:

- `in((:Z, 999), d3)` or `(:Z, 999) in d3` returns false
- `in((:A, 100), d3)` or `(:A, 100) in d3` returns true

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

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

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

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

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

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

Keys and values – looping

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

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

This prints out `A` and `B`. This gives us also an easier way to test if a key exists with `in`, for example, `:A` in `keys(d3)` returns `true` and `:Z` in `keys(d3)` returns `false`.

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

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

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

Creating a dictionary from arrays with `keys` and `values` is trivial because we have a `Dict` constructor that can use these. For example:

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

Collection Types

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

```
d5 = [ "J.S. Bach" => 1685, "Woody Allen" => 1935,
      "Barack Obama" => 1961 ]
```

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

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

This will print the following output:

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

Or alternatively, using an index in every (key,value) tuple of `d5`:

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

If the key-value pairs are arranged in a single array, like this:

```
dpairs = ["a", 1, "b", 2, "c", 3]
```

Then, you can build a dictionary from this array with the following comprehension:

```
d6 = [dpairs[i] => dpairs[i+1] for i in 1:2:length(dpairs)]
```

Here, `1:2:length(dpairs)` iterates over the array in steps of two: `i` will therefore take on values 1, 3, and 5.

If you want it typed, prefix it with `(String => Int64)` like this:

```
d6 = (String => Int64)[dpairs[i] => dpairs[i+1] for i in
1:2:length(dpairs)]
```

Here is a nice example of the use of a dictionary with the built-in function `factor`. The function `factor` takes an integer and returns a dictionary with the prime factors as the keys, and the number of times each prime appears in the product as values:

```
function showfactor(n)
    d = factor(n)
    println("factors for $n")
    for (k, v) in d
        print("$k^$v\t")
    end
end
```

@time showfactor(3752) outputs the following result:

```
factors for 3752
7^1      2^3      67^1      elapsed time: 0.000458578 seconds
(2472 bytes allocated)
```

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

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

```
arrkey = [key for (key, value) in dict]
This is the same as collect(keys(dict)).
```

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

```
arrval = [value for (key, value) in dict]
This is the same as collect(values(dict))
```

- Make an array with the first n values of a dictionary when the keys are the integers from 1 to n and beyond:

```
arrn = [dict[i] for i = 1:n]
```

This can also be written as a `map`, `arrn = map((i) -> dict[i], [1:n])`

From Julia v0.4 onwards, the following literal syntaxes or the `Dict` constructors are deprecated:

```
d1 = [1 => 4.2, 2 => 5.3]
d2 = {"a"=>1, (2,3)=>true}
capitals = (String => String) ["France"=> "Paris",
"China"=>"Beijing"]
d5 = Dict(keys1, values1)
```

They take the new forms as follows:

```
d1 = Dict(1 => 4.2, 2 => 5.3)
d2 = Dict{Any,Any}("a"=>1, (2,3)=>true)
capitals = Dict{String, String}("France"=> "Paris",
    "China"=>"Beijing") # from v0.4 onwards
d5 = Dict(zip(keys1, values1))
```

 This is indicated for all the examples in the accompanying code file, `dicts.jl`. It can be difficult to make packages to work on both the versions. The `Compat` package (<https://github.com/JuliaLang/Compat.jl>) was created to help with this, as it provides compatibility constructs that will work in both the versions without warnings.

Sets

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

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

The `Set()` function creates an empty set. The preceding line returns `Set{Int64}({7,14,13,11})`, where the duplicates have been eliminated. From v0.4 onwards, the `{}` notation with sets is deprecated; you should use `s = Set(Any[11, 14, 13, 7, 14, 11])`. In the accompanying code file, the latest version is used.

The operations from the set theory are also defined for `s1 = Set({11, 25})` and `s2 = Set({25, 3.14})` as follows:

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

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

Be careful if you want to define a set of arrays: `Set([1, 2, 3])` produces a set of integers `Set{Int64}({2, 3, 1})`; to get a set of arrays, use `Set({[1, 2, 3]})` that returns `Set{Any}({[1, 2, 3]})`.

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

```
x = Set([1:100])
@time 2 in x # elapsed time 4.888e-6 seconds
x2 = Set([1:1000000])
@time 2 in x2 # elapsed time 5.378e-6 seconds
```

Both the tests take approximately the same time, despite the fact that `x2` is four orders of magnitude larger than `x`.

Making a set of tuples

You can start by making an empty set as this: `st = Set{ (Int, Int) }()`.

Then, you can use `push!` to start filling it up: `push!(st, (1, 2))`, which returns a `Set{ (Int64, Int64) }({(1, 2)})`, and so on. Another possibility is to use `[]`, for example, `st2 = Set([(1, 2), (5, 6)])` produces a set with the two tuples `(1, 2)` and `(5, 6)`.

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

Example project – word frequency

A lot of the concepts and techniques that we have seen so far in the book come together in this little project. Its aim is to read a text file, remove all the characters that are not used in words, and count the frequencies of the words in the remaining text. This can be useful, for example, when counting the word density on a web page, the frequency of DNA sequences, or the number of hits on a website that came from various IP addresses. This can be done in some 10 lines of code. For example, when `words1.txt` contains the sentence `to be, or not to be, that is the question!`, then this is the output of the program:

Collection Types

Word : frequency

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

Here is the code with comments:

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

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

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

Summary

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

6

More on Types, Methods, and Modules

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

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

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

- Type annotations and conversions
- The type hierarchy – subtypes and supertypes
- Concrete and abstract types
- User-defined and composite types
- Type unions
- Parametric types
- Parametric and constructor methods
- Standard modules and paths

Type annotations and conversions

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

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

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

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

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

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

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

Type conversions and promotions

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

In general, `convert(Type, x)` will attempt to put the `x` value in an instance of `Type`. In most cases, `type(x)` will also do the trick, as in `int64(7.0)`.

The conversion, however, doesn't always work:

- When precision is lost: `convert(Int64, 7.01)` returns an `ERROR: InexactError()` error message, however, `int64(7.01)` rounds off and converts to the nearest integer, producing the output as 7
- When the target type is incompatible with the source value: `convert(Int64, "CV")` returns an `ERROR: `convert` has no method matching convert(::Type{Int64}, ::ASCIIString)` error message

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

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

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

- `promote(1, 2.5, 3//4)` returns `(1.0, 2.5, 0.75)`
- `promote(1.5, im)` returns `(1.5 + 0.0im, 0.0 + 1.0im)`
- `promote(true, 'c', 1.0)` returns `(1.0, 99.0, 1.0)`

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

It basically says that first, promote the arguments to a common type, and then perform the addition. A number is a common supertype for all values of numeric types. To determine the common promotion type of the two types, use `promote_type(Int8, UInt16)` to find whether it returns `Int64`.

This is because somewhere in the standard library the following `promote_rule` function was defined as `promote_rule(::Type{Int8}, ::Type{UInt16}) = Int64`.

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

The type hierarchy – subtypes and supertypes

(Follow along with the code in Chapter 6\type_hierarchy.jl.)

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

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

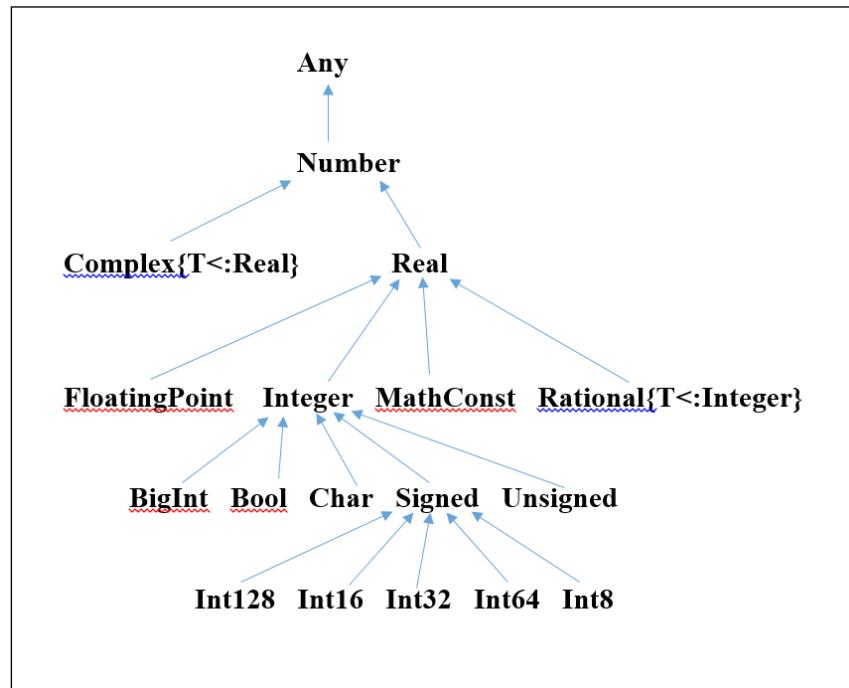
- `super(Int64)` returns `Signed`
- `super(Signed)` returns `Integer`
- `super(Integer)` returns `Real`
- `super(Real)` returns `Number`
- `super(Number)` returns `Any`
- `super(Any)` returns `Any`

A type can have a lot of children or subtypes as follows:

- `subtypes(Integer)` form 5-element `Array{Any, 1}` that contains `BigInt`, `Bool`, `Char`, `Signed`, and `Unsigned`
- `subtypes(Signed)` form 5-element `Array{Any, 1}` that contain `Int128`, `Int16`, `Int32`, `Int64`, and `Int8`
- `subtypes(Int64)` is 0-element `Array{Any, 1}`, and it has no subtypes

To indicate the subtype relationship, the operator `<` is used: `Bool <: Integer` and `Bool <: Any` return true, while `Bool <: Char` is false. An equivalent form uses the `issubtype` function: `issubtype(Bool, Integer)` is true, but `issubtype(Float64, Integer)` returns false.

Here is a visualization of part of this type tree:



Concrete and abstract types

In this hierarchy, some types, such as `Number`, `Integer`, and `Signed`, are abstract, which means that they have no concrete objects or values of their own. Instead, objects or values are of concrete types given by the result of applying `typeof(value)`, such as `Int8`, `Float64`, and `UTF8String`. For example, the concrete type of the value `5` is `Int64` given by `typeof(5)` (on a 64-bit machine). However, a value has also the type of all of its supertypes, for example, `isa(5, Number)` returns true (we introduced the `isa` function in the *Types* section of *Chapter 2, Variables, Types, and Operations*).

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

Type	Instantiate	Subtypes
concrete	Y	N
abstract	N	Y

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

Julia's type tree can be graphically visualized by running the following command:

```
julia julia_types.jl > tree.txt
```

Here is a little fragment of its output:

```

. . . +- Integer << abstract immutable size:0 >>
. . . . +- Signed << abstract immutable size:0 >>
. . . . . +- FileOffset = Int64 << concrete immutable
pointerfree size:8 >>
. . . . . +- Cssize_t = Int64 << concrete immutable
pointerfree size:8 >>
. . . . . +- Clonglong = Int64 << concrete immutable
pointerfree size:8 >>
. . . . . +- Cchar = Int8 << concrete immutable pointerfree
size:1 >>
. . . . . +- Clong = Int32 << concrete immutable pointerfree
size:4 >>
. . . . . +- Cint = Int32 << concrete immutable pointerfree
size:4 >>
. . . . . +- Int8 << concrete immutable pointerfree
size:1 >>
. . . . . +- Integer64 =
Union(Uint16,Uint8,Int8,Uint32,Int16,Int64,Int32,Uint64)
. . . . . . +- SmallSigned = Union(Int8,Int16,Int32)
. . . . . . +- Signed64 = Union(Int8,Int16,Int64,Int32)
. . . . . . +- Int16 << concrete immutable pointerfree
size:2 >>
. . . . . . +- Integer64 =
Union(Uint16,Uint8,Int8,Uint32,Int16,Int64,Int32,Uint64)
. . . . . . +- SmallSigned = Union(Int8,Int16,Int32)
. . . . . . +- Signed64 = Union(Int8,Int16,Int64,Int32)
. . . . . . +- Coff_t = Int64 << concrete immutable
pointerfree size:8 >>
. . . . . . +- Int128 << concrete immutable pointerfree
size:16 >>
. . . . . . +- CommonReduceResult =
Union(Float32,Int128,Float64,Int64,Uint128,Uint64)
. . . . . . +- Cptrdiff_t = Int64 << concrete immutable
pointerfree size:8 >>
. . . . . . +- Int64 << concrete immutable pointerfree
size:8 >>
. . . . . . +- Integer64 =
Union(Uint16,Uint8,Int8,Uint32,Int16,Int64,Int32,Uint64)

```

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

At the other end is None; all types are supertypes of None and no object is an instance of None. The None type has no values and no subtypes, it is unlikely that you will ever have to use this type.

Different from `None` is the type `Nothing`; this has one value named `nothing`. When a function is only used for its side effects, convention dictates that it returns `nothing`. We have seen this with the `println` function, where the printing is the side effect, for instance:

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

From v0.4 onwards, these types are named differently: `None` becomes `Union()`, and `Nothing` becomes `Void`.

User-defined and composite types

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

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

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

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

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

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

The names of the fields that belong to a composite type can be obtained with the `names` function of the type or an object: `names(Point)` or `names(p1)` returns 3-element `Array{Symbol,1}: :x :y :z`.

A user-defined type has two default implicit *constructors* that have the same name as the type and take an argument for each field. You can see this by asking for the methods of `Point`: `methods(Point)` returns 2 methods for generic function "`Point`": `Point(x::Float64, y::Float64, z::Float64)` and `Point(x,y,z)`. Here, the field values can be of type `Any`.

You can now make objects simply like this:

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

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

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

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

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

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

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



Immutable types enhance the performance, because Julia can optimize the code for them. Another big advantage of immutable types is thread safety: an immutable object can be shared between threads without needing synchronization.

If, however, such an immutable type contains a mutable field such as an array, the contents of that field can be changed. So, define your immutable types without mutable fields.

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

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

When are two values or objects equal or identical?

To check whether two values are equal or not can be decided by the `==` operator, for example, `5 == 5` and `5 == 5.0` are both `true`. To see whether the two objects `x` and `y` are identical, they must be compared with the `is` function, and the result is a Boolean value, `true` or `false`:

```
is(x, y) -> Bool
```

The `is(x, y)` function can also be written with the three `=` signs as `x === y`.

Objects such as numbers are immutable and they are compared at the bits level: `is(5, 5)` returns `true` and `is(5, 5.0)` returns `false`.

For objects that are more complex, such as strings, arrays, or objects that are constructed from composite types, the addresses in memory are compared to check whether they point to the same memory location. For example, if `q = Vector3D(4.0, 3.14, 2.71)`, and `r = Vector3D(4.0, 3.14, 2.71)`, then `is(q, r)` returns `false`.

Multiple dispatch example

Let's now explore an example about people working in a company to show multiple dispatch in action. Let's define an abstract type `Employee` and a type `Developer` that is a subtype:

```
abstract Employee
type Developer <: Employee
    name::String
    iq
    favorite_lang::String
end
```

We cannot make objects from an abstract type: calling `Employee()` only returns an `ERROR: type cannot be constructed` error message.

The type `Developer` has two implicit constructors, but we can define another *outer constructor* that uses a default constructor as follows:

```
Developer(name, iq) = Developer(name, iq, "Java")
```

Outer constructors provide additional convenient methods to construct objects. Now, we can make the following two developer objects:

- `devel1 = Developer("Bob", 110)` that returns `Developer("Bob", 110, "Java")`
- `devel2 = Developer("William", 145, "Julia")` that returns `Developer("William", 145, "Julia")`

Similarly, we can define a type `Manager` and an instance of it as follows:

```
type Manager
    name::String
    iq
    department::String
end
man1 = Manager("Julia", 120, "ICT")
```

Concrete types, such as `Developer` or `Manager`, cannot be subtyped:

```
type MobileDeveloper <: Developer
    platform
end
```

This returns `ERROR: invalid subtyping in definition of MobileDeveloper.`

If we now define a function `cleverness` as `cleverness(emp::Employee) = emp.iq`, then `cleverness(devel1)` returns 110, but `cleverness(man1)` returns an `ERROR: `cleverness` has no method matching cleverness(::Manager)` error message; the function has no method for a manager.

Suppose we introduce a function `cleverer` with the following argument types:

```
function cleverer(d::Developer, e::Employee)
    println("The developer $(d.name) is cleverer I think!")
end
```

The `cleverer(devel1, devel2)` function will now print "The developer Bob is cleverer I think!" (Clearly, the function isn't yet coded right, we are biased in thinking that developers are always more intelligent). It matches a method because `devel2` is also an employee. However, `cleverer(devel1, man1)` will give an `ERROR: `cleverer` has no method matching cleverer(::Developer, ::Manager)` error message, as a manager is not an employee, and a method with this signature was not defined.

We now define another method for `cleverer` as follows:

```
function cleverer(e::Employee, d::Developer)
    if e.iq <= d.iq
        println("The developer $(d.name) is cleverer!")
    else
        println("The employee $(e.name) is cleverer!")
    end
end
```

Now an ambiguity arises; Julia detects a problem in the definitions and gives us the following warning:

```
Warning: New definition
cleverer(Employee,Developer) at none:2
is ambiguous with:
cleverer(Developer,Employee) at none:2.
To fix, define
cleverer(Developer,Developer)
before the new definition.
```

The ambiguity is that if `cleverer` is called with `e` being a `Developer`, which of the two defined methods should be chosen? Julia takes a pragmatic standpoint and `cleverer(devel1, devel2)` still gives the same outcome. However, now we will define the more specific (and correct) method as follows:

```
function cleverer(d1::Developer, d2::Developer)
    if d1.iq <= d2.iq
        println("The developer $(d2.name) is cleverer!")
    else
        println("The developer $(d1.name) is cleverer!")
    end
end
```

Now, `cleverer(devel1, devel2)` prints "The developer William is cleverer!" as well as `cleverer(devel2, devel1)`. This illustrates multiple dispatching. When defined, the more specific method definition (here, the second method `cleverer`) is chosen. More specific means the method with more specialized type annotations for its arguments. More specialized doesn't only mean subtypes, it can also mean using type aliases.



Always avoid method ambiguities by specifying an appropriate method for the intersection case.



Types and collections – inner constructors

Here is another type with only default constructors:

```
# see the code in Chapter 6\inner_constructors.jl
type Person
    firstname::String
    lastname::String
    sex::Char
    age::Float64
    children)::Array{String, 1}
end
p1 = Person("Alan", "Bates", 'M', 45.5, ["Jeff", "Stephan"])
```

This example demonstrates that an object can contain collections such as arrays or dictionaries. Custom types can also be stored in a collection, just like built-in types, for example:

```
people = Person[]
```

This returns 0-element Array{Person,1}.

```
push!(people, p1)
push!(people, Person("Julia", "Smith", 'F', 27, ["Viral"]))
```

The show(people) function now returns the following output:

```
[Person("Alan", "Bates", 'M', 45.5, String["Jeff", "Stephan"]),
 Person("Julia", "Smith", 'F', 27.0, String["Viral"])]
```

Now, we can define a function fullname on type Person. You notice that the definition stays outside the type's code:

```
fullname(p::Person) = "$(p.firstname) $(p.lastname)"
```

Or, slightly more performant:

```
fullname(p::Person) = string(p.firstname, " ", p.lastname)
```

Now, print(fullname(p1)) returns Alan Bates.

If you need to include error checking or transformations as part of the type construction process, you can use inner constructors (so-called because they are defined inside the type itself), as shown in the following example:

```
type Family
    name::String
    members::Array{String, 1}
    big::Bool
    Family(name::String) = new(name, String[], false)
    Family(name::String, members) = new(name, members,
        length(members) > 4)
end
```

We can make a Family object as follows:

```
fam = Family("Bates-Smith", ["Alan", "Julia", "Jeff", "Stephan",
    "Viral"])
```

Then the output is as follows:

```
Family("Bates-Smith",String["Alan","Julia","Jeff","Stephan","Viral"],true)
```

The keyword `new` can only be used in an inner constructor to create an object of the enclosing type. The first constructor takes one argument and generates a default for the other two values. The second constructor takes two arguments and infers the value of `big`. Inner constructors give you more control over how values of the type can be created. Here, they are written with the short function notation, but if they are multiline, they would use the normal function syntax.

Note that when you use inner constructors, there are no default constructors anymore. Outer constructors calling a limited set of inner constructors is often the best practice.

Type unions

In geometry, a two-dimensional point and a vector are not the same, even if they both have an `x` and `y` component. In Julia, we can also define them as different types as follows:

```
# see the code in Chapter 6\unions.jl
type Point
    x::Float64
    y::Float64
end

type Vector2D
    x::Float64
    y::Float64
end
```

Here are the two objects:

- `p = Point(2, 5)` that returns `Point(2.0, 5.0)`
- `v = Vector2D(3, 2)` that returns `Vector2D(3.0, 2.0)`

Suppose we want to define the sum for these types as a point that has coordinates as the sum of the corresponding coordinates:

```
+ (p, v)
```

This results in an ERROR: `+` has no method matching `+(::Point, ::Vector2D)` error message.

Even after defining the following, `+(p, v)` still returns the same error because of multiple dispatch (Julia has no way of knowing that `+(p, v)` should be the same as `+ (v, p)`):

```
+ (p::Point,     q::Point) = Point(p.x + q.x, p.y + q.y)
+ (u::Vector2D,  v::Vector2D) = Point(u.x + v.x, u.y + v.y)
+ (u::Vector2D,  p::Point) = Point(u.x + p.x, u.y + p.y)
```

Only when we define the type matching method as `+ (p::Point, v::Vector2D) = Point(p.x + v.x, p.y + v.y)`, we get a result `+ (p, v)` that returns `Point(5.0, 7.0)`.

Now, you can ask the question: doesn't multiple dispatch and many types give rise to code duplication as in the case here?

However, this is not so because in such a case, we can define a union type `VecOrPoint`:

```
VecOrPoint = Union(Vector2D, Point)
```

If `p` is a point, it is also of type `VecOrPoint`, and the same is true for a `Vector2D` `v`: `isa(p, VecOrPoint)` and `isa(v, VecOrPoint)` both return `true`.

Now, we can define one `+` method that works for any of the preceding four cases:

```
+ (u::VecOrPoint, v::VecOrPoint) = VecOrPoint(u.x + v.x, u.y +
v.y)
```

So, now we only need one method instead of four.

Parametric types and methods

An array can take elements of different types, so, we can have, for example, arrays of the following types: `Array{Int64, 1}`, `Array{Int8, 1}`, `Array{Float64, 1}`, or `Array{ASCIIString, 1}`, and so on. That is why an Array is a **parametric type**; its elements can be of any arbitrary type `T`, written as `Array{T, 1}`.

In general, types can take **type parameters**, so that type declarations actually introduce a whole family of new types. Returning to the Point example of the previous section, we can generalize it to the following:

```
# see the code in Chapter 6\parametric.jl
type Point{T}
    x::T
    y::T
end
```

(This is conceptually similar to generic types in Java or templates in C++).

This abstract type creates a whole family of new possible concrete types (but they are only compiled as needed at runtime), such as `Point{Int64}`, `Point{Float64}`, and `Point{String}`.

These are all subtypes of `Point`: `issubtype(Point{String}, Point)` that return `true`. However, this is not the case when comparing different `Point` types, whose parameter types are subtypes of one another: `issubtype(Point{Float64}, Point{Real})` returns `false`.

To construct objects, you can indicate the type `T` in the constructor, as in `p = Point{Int64}(2, 5)`, but this can be shortened to `p = Point(2, 5)`, or let's consider another example: `p = Point("London", "Great-Britain")`.

If you want to restrict the parameter type `T` to only the subtypes of `Real`, this can be written as follows:

```
type Point{T <: Real}
    x::T
    y::T
end
```

Now, the statement `p = Point("London", "Great-Britain")` results in an ERROR: ``Point{T<:Real}` has no method matching Point{T<:Real}(::ASCIIString, ::ASCIIString)` error message, because `String` is not a subtype of `Real`.

Much in the same way, methods also optionally can have type parameters immediately after their name and before the tuple of arguments, for example, to constrain two arguments to be of the same type `T`, run the following command:

```
add{T}(x::T, y::T) = x + y
```

Now, `add(2, 3)` returns 5 and `add(2, 3.0)` returns an ERROR: ``add` has no method matching add(::Int64, ::Float64)` error message.

Here, we restrict T to be a subtype of Number in `add` as follows:

```
add{T <: Number}(x::T, y::T) = x + y
```

As another example, here is how to check whether a `vecfloat` function only takes a vector of floating point numbers as the input. Simply, define it with a type parameter T as follows:

```
function vecfloat{T <: FloatingPoint}(x::Vector{T})  
    # code  
end
```

Inner constructors can also take type parameters in their definition.

Standard modules and paths

The code of Julia packages (also called **libraries**) is contained in a module, whose name starts with an uppercase letter by convention like this:

```
# see the code in Chapter 6\modules.jl  
module Package1  
    # code  
end
```

This serves to separate all its definitions from those in the other modules, so that no name conflicts occur. Name conflicts are solved by qualifying the function by the module name. For example, the packages `Winston` and `Gadfly` both contain a function `plot`. If needed these two versions in the same script, we would write this as follows:

```
import Winston  
import Gadfly  
Winston.plot(rand(4))  
Gadfly.plot(x=[1:10], y=rand(10))
```

All variables defined in the `global` scope are automatically added to the `Main` module. Thus, when you write `x = 2` in the REPL, you are adding the variable `x` to the `Main` module.

Julia starts with `Main` as the current top-level module. The module `Core` contains all built-in identifiers, and it is always available. The standard library is also available. All of its code (the contents of `/base`) is contained in the modules `Base`, `Pkg`, `Collections`, `Graphics`, `Test`, and `Profile`.

The type of a module is `Module`: `typeof(Base)` that returns `Module`. If we call `names(Main)`, we get, for example, 6-element `Array{Symbol,1}`: `:ans, :a, :vecfloat, :Main, :Core, :Base`, depending on what you have defined in the REPL.

All the top-level defined variables and functions, together with the default modules are stored as symbols. The `whos()` function lists these objects with their types:

Base	Module
Core	Module
Main	Module
a	Int64
ans	6-element <code>Array{Symbol,1}</code>
vecfloat	Function

This can also be used for another module, for example, `whos(Winston)` lists all the exported names from the module `Winston`.

A module can make some of its internal definitions such as constants, variables, types, functions, and so on visible to other modules (make them `public`) by declaring them with `export`, for example:

```
export Type1, perc
```

If a module `LibA` (among others) is defined in a `modules_ext.jl` file, then the statement `require("modules_ext.jl")` will load this in the current code. Using `LibA`, will make all its exported names available in the current namespace; this is what we did in the REPL to load a package.

For the preceding example, using `Package1` will make the type `Type1` and function `perc` available in other modules that import them through this statement. All the other definitions remain invisible (or `private`).

Here is a more concrete example. Suppose we define a `TemperatureConverter` module as follows:

```
#code in Chapter 6\temperature_converter.jl
module TemperatureConverter

    export as_celsius

    function as_celsius(temperature, unit)
        if unit == :Celsius
            return temperature
        elseif unit == :Kelvin
```

```
        return kelvin_to_celsius(temperature)
    end
end

function kelvin_to_celsius(temperature)
    # 'private' function
    return temperature + 273
end

end
```

We can now use this module in another program as follows:

```
#code in Chapter 6\using_module.jl
require("temperature_converter.jl")

using TemperatureConverter

println!("$(as_celsius(100, :Celsius))") #> 100
println!("$(as_celsius(100, :Kelvin))") #> 373
# println!("$(kelvin_to_celsius(0))") #> ERROR: kelvin_to_celsius
#       not defined
```

The function `kelvin_to_celsius` was not exported, and so is not known in the program `using_module.jl`.

In general, there are different ways of importing definitions from another module `LibA` in the current module:

- First, make use of `using LibA`, then `LibA` will be searched for exported definitions if needed. A function from `LibA` can then be used without qualifying it with the module name.
- If you want to be more selective, you can execute `using LibB.varB` or the shorthand, `using LibC: varC, funcC`.
- The `import LibD.funcD` statement only imports one name and can also be used if `funcD` was not exported; the function `funcD` must be used as `LibD.funcD`.
- Use `importall LibE` to import all the exported names in `LibE`.

Imported variables are read-only, and the current module cannot create variables with the same names as the imported ones. A source file can contain many modules, or one module can be defined in several source files. If a module contains a function `__init__()`, this will be executed when the module is first loaded.

As we saw in *Chapter 1, Installing the Julia Platform*, a module can also include other source files in their entirety with `include("file1.jl")`, but then, the included files are not modules. Using `include("file1.jl")` is, to the compiler, no different to copying `file1.jl` and pasting it directly in the current file or the REPL.

The variable `LOAD_PATH` contains a list of directories where Julia looks for (module) files when running the `using`, `import`, or `include` statements. It can be set up at the operating system level: in a start-up script such as `.bashrc` or `.profile`, or in `Environment Variables` on Windows. You can extend this variable in the code using `push!:`

```
push!(LOAD_PATH, "new/path/to/search")
```

Modules are compiled on load, which slows down Julia's start-up time in the current version. This will improve considerably once precompiling of modules is possible, which is planned for Version 0.4.

Summary

In this chapter, we delved into types and type hierarchy in Julia. We got a much better understanding of types and how functions work on them through multiple dispatch. The next chapter will reveal another power tool in Julia: **metaprogramming and macros**.

7

Metaprogramming in Julia

Everything in Julia is an expression that returns a value when executed. Every piece of the program code is internally represented as an ordinary Julia data structure, also called an **expression**. In this chapter, we will see that by working on expressions, how a Julia program can transform and even generate the new code, which is a very powerful characteristic, also called **homoiconicity**. It inherits this property from **Lisp**, where code and data are just lists, and where it is commonly referred to with the phrase: "**code is data and data is code**". We will explore this metaprogramming power by covering the following topics:

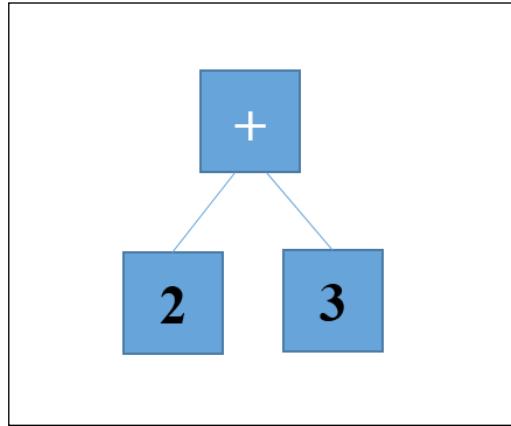
- Expressions and symbols
- Eval and interpolation
- Defining macros
- Built-in macros
- Reflection capabilities

Expressions and symbols

An **abstract syntax tree (AST)** is a tree representation of the abstract syntactic structure of the source code written in a programming language. When Julia code is parsed by its LLVM JIT compiler, it is internally represented as an abstract syntax tree. The nodes of this tree are simple data structures of type expression `Expr`.

(For more information on abstract syntax trees, refer to http://en.wikipedia.org/wiki/Abstract_syntax_tree).

An expression is simply an object that represents Julia code. For example, `2 + 3` is a piece of code, which is an expression of type `Int64` (follow along with the code in Chapter 7\expressions.jl). Its syntax tree can be visualized as follows:



To make Julia see this as an expression and block its evaluation, we have to quote it, that is, precede it by a colon (:) as in `: (2 + 3)`. When you evaluate `: (2 + 3)` in the REPL, it just returns `: (2 + 3)`, which is of type `Expr`: `typeof (: (2 + 3))` returns `Expr`. In fact, the `:` operator (also called the **quote** operator) means to treat its argument as data, not as code.

If this code is more than one line, enclose them between the `quote` and `end` keywords to turn the code into an expression, for example, this expression just returns itself:

```
quote
    a = 42
    b = a^2
    a - b
end
```

In fact, this is the same as `: (a = 42; b = a^2; a - b)`. `quote ... end` is just another way to convert blocks of code into expressions.

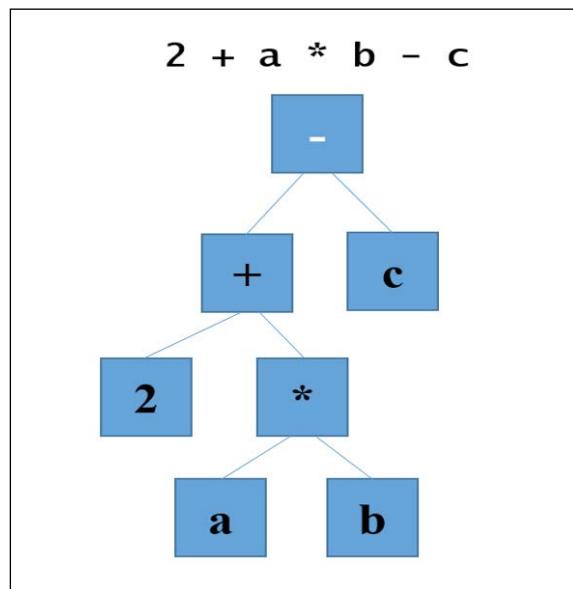
We can give such an expression a name, such as `e1 = : (2 + 3)`. We can then inquire if `e1` has fields with names `(e1)`, which returns 3-element `Array{Symbol,1}`: `:head, :args, :typ`.

These return the following information:

- `e1.head` returns `:call`, indicating the kind of expression, which here is a function call
- `e1.args` returns 3-element `Array{Any,1}`: `:+ 2 3`
- `e1.typ` returns `Any`; it is used by the type inference mechanism to store type annotations

Indeed the expression `2 + 3` is, in fact, a call of the `+` function with the argument `2` and `3`: `2 + 3 == + (2, 3)` returns `true`. The `args` argument consists of a symbol `:+` and two literal values `2` and `3`. Expressions are made of symbols and literals. More complicated expressions will consist of literal values, symbols, and sub- or nested expressions, which can, in turn, be reduced to symbols and literals.

For example, consider the expression `e2 = : (2 + a * b - c)`, which can be visualized by the following syntax tree:



`e2` consists of `e2.args`, which is a 3-element `Array{Any,1}` that contains `:-` and `:c`, which are symbols, and `:(2 + a * b)`, which is also an expression. This last expression, in turn, is itself an expression with `args :+, 2`, and `:(a * b); :(a * b)` is an expression with arguments and symbols: `:*`, `:a`, and `:b`. We can see that this works recursively; we can simplify every subexpression in the same way until we end up with elementary symbols and literals.

In the context of an expression, *symbols are used to indicate access to variables*; they represent the variable in the tree structure of the code. In fact, the "**prevent evaluation**" character of the quote operator (:) is already at work with the symbols: after `x = 5`, `x` returns 5, but `:x` returns `:x`.

With this knowledge, we can conclude that the definition of the type `Expr` in Julia goes as follows:

```
type Expr
    head::Symbol
    args::Array{Any,1}
    typ
end
```

The `dump` function presents the abstract syntax tree for its argument in a nice way. For example, `dump(:((2 + a * b - c)))` returns the output, as shown in the following screenshot:

```
julia> dump(:((2 + a * b - c)))
Expr
  head: Symbol call
  args: Array{Any,(3,)})
    1: Symbol -
    2: Expr
      head: Symbol call
      args: Array{Any,(3,)})
        1: Symbol +
        2: Int64 2
        3: Expr
          head: Symbol call
          args: Array{Any,(3,)})
            typ: Any
          3: Symbol c
            typ: Any
```

Eval and interpolation

With the definition of type `Expr` from the preceding section, we can also build expressions directly from the constructor for `Expr`, for example: `e1 = Expr(:call, *, 3, 4)` returns `:((*)(3, 4))` (follow along with the code in Chapter 7\eval.jl).

The result of an expression can be computed with the `eval` function, `eval(e1)`, which returns 12 in this case. At the time an expression is constructed, not all the symbols have to be defined, but they have to be at the time of evaluation, otherwise an error occurs.

For example, `e2 = Expr(:call, *, 3, :a)` returns `:((*)(3, a))` and `eval(e2)` then, gives `ERROR: a not defined`. Only after we say, for example, `a = 4` does `eval(e2)` and returns `12`.

Expressions can also change the state of the execution environment, for example, the expression `e3 = :(b = 1)` assigns a value to `b` when evaluated, and even defines `b` if it doesn't exist already.

To make writing expressions a bit simpler, we can use the `$` operator to do **interpolation** in expressions; as with `$` in strings, and this will evaluate immediately when the expression is made. The expressions `a = 4` and `b = 1`, `e4 = :(a + b)` return `:(a + b)` and `e5 = :($a + b)` returns `:(4 + b)`; both the expressions evaluate to `5`. So, there are two kinds of evaluations here:

- Expression interpolation (with `$`) evaluates when the expression is constructed (at parse time)
- Quotation (with `:` or `quote`) evaluates only when the expression is passed to `eval` at runtime

We now have the capability to build the code programmatically; inside a Julia program, we can construct the arbitrary code while it is running, and then evaluate this with `eval`. So, Julia can generate the code from inside itself during the normal program execution.

This happens all the time in Julia and it is used, for example, to do things such as to generate bindings for external libraries, to reduce the repetitive boilerplate code needed to bind big libraries, or to generate lots of similar routines in other situations. Also, in the field of robotics, the ability to generate another program and then, run it is very useful. For example: a surgical robot learns how to move by perceiving a human surgeon demonstrate a procedure. Then, the robot generates the program code from that perception, so that it is able to perform the procedure by itself.

One of the most powerful Julia tools emerging from what we discussed until now is **macros**, which exist in all the languages of the Lisp family.

Defining macros

In the previous chapters, we already used macros, such as `@printf` in *Chapter 2, Variables, Types, and Operations*, and `@time` in *Chapter 3, Functions*. Macros are like functions, but instead of values, they take expressions (which can also be symbols or literals) as input arguments. When a macro is evaluated, the input expression is expanded, that is, the macro returns a modified expression. This expansion occurs at parse time when the syntax tree is being built, not when the code is actually executed.

The following highlights the difference between macros and functions when they are called or invoked:

- **Function:** It takes the input values and returns the computed values at runtime
- **Macro:** It takes the input expressions and returns the modified expressions at parse time

In other words, a macro is a custom program transformation. Macros are defined with the keyword as follows:

```
macro mname
    # code returning expression
end
```

It is invoked as @mname exp1 exp2 or @mname(exp1, exp2) (the @ sign distinguishes it from a normal function call). The macro block defines a new scope. Macros allow us to control when the code is executed.

Here are some examples:

- A first simple example is a `macint` macro, which does the interpolation of its argument expression `ex`:

```
# see the code in Chapter 7\macros.jl)
macro macint(ex)
    quote
        println("start")
        $ex
        println("after")
    end
end
```

@`macint` `println("Where am I?")` will result in:

```
start
Where am I?
after
```

- The second example is an `assert` macro that takes an expression `ex` and tests whether it is true or not, in the last case, an error is thrown:

```
macro assert(ex)
    :($ex ? nothing : error("Assertion failed: ",
                           $(string(ex))))
end
```

For example: @assert 1 == 1.0 returns nothing. @assert 1 == 42 returns
ERROR: Assertion failed: 1 == 42.

The macro replaces the expression with a ternary operator expression, which is evaluated at runtime. To examine the resulting expression, use the macroexpand function as follows:

```
macroexpand(:(@assert 1 == 42))
```

This returns the following expression:

```
: (if 1 == 42
    nothing
  else
    error("Assertion failed: ", "1 == 42")
  end)
```

This assert function is just a macro example, use the built-in assert function in the production code (refer to the *Testing* subsection of the *Built-in macro's* section).

- The third example mimics an unless construct, where branch is executed if the condition test is not true:

```
macro unless(test, branch)
  quote
    if !$test
      $branch
    end
  end
end
```

Suppose arr = [3.14, 42, 'b'], then @unless 42 in arr
println("arr does not contain 42") returns nothing, but @unless 41 in arr println("arr does not contain 41") prints out the following command:

```
arr does not contain 41
```

Here, macroexpand(:(@unless 41 in arr println("arr does not contain 41"))) returns the following output:

```
quote # none, line 3:
  if !(41 in arr) # line 4:
    println("arr does not contain 41")
  end
end
```

- The fourth example shows how to convert an array of strings to an array of type T with a `convarr` macro.

Suppose the array is `arr = ["a", "b", "c"]`, then, we define a macro as follows:

```
macro convarr(arr, T)
    : (reshape($T[$arr...], size($arr)...))
end
```

Notice how the destination type T is general. The `reshape` function is used here to redimension the array (this is its signature: `Base.reshape(arr, dims)`) so, this macro will work for arrays of arbitrary dimensions.

For example, calling `@convarr arr Symbol` returns 3-element
`Array{Symbol,1} :a :b :c.`

Unlike functions, macros inject the code directly in the namespace in which they are called, possibly this is also in a different module than in which they were defined. It is therefore important to ensure that this generated code does not clash with the code in the module in which the macro is called. When a macro behaves appropriately like this, it is called a **hygienic macro**. The following rules are used when writing hygienic macros:

- Declare the variables used in the macro as `local`, so as not to conflict with the outer variables
- Use the escape function `esc` to make sure that an interpolated expression is not expanded, but instead is used literally
- Don't call `eval` inside a macro (because it is likely that the variables you are evaluating don't even exist at that point)

These principles are applied in the following `timeit` macro that times the execution of an expression `ex` (like the built-in macro `@time`):

```
macro timeit(ex)
    quote
        local t0 = time()
        local val = $(esc(ex))
        local t1 = time()
        print("elapsed time in seconds: ")
        @printf "%.3f" t1 - t0
        val
    end
end
```

The expression is executed through `$`, and `t0` and `t1` are respectively the start and end times.

```
@timeit factorial(10) returns elapsed time in seconds: 0.0003628800.
```

```
@timeit a^3 returns elapsed time in seconds: 0.0013796416.
```

Hygiene with macros is all about differentiating between the macro context and the calling context.

Macros are valuable tools to save you a lot of tedious work and with the quoting and interpolation mechanism, they are fairly easy to create. You will see them being used everywhere in Julia for lots of different tasks. Ultimately, they allow you to create domain-specific languages (DSLs). To get a better idea of this concept, we suggest you experiment with the other examples in the accompanying code file.

Built-in macros

Needless to say the Julia team has put macros to good use. To get the help information about a macro, enter `a ?` in the REPL, and type `@macroName` after the `help>` prompt, or type `help ("@macroName")`. Apart from the built-in macros we encountered in the examples in the previous chapters, here are some other very useful ones (refer to the code in `Chapter 7\built_in_macros.jl`).

Testing

The `@assert` macro actually exists in the standard library. The standard version also allows you to give your own error message, which is printed after `ERROR: assertion failed`.

The `Base.Test` library contains some useful macros to compare the numbers:

```
using Base.Test  
@test 1 == 3
```

This returns `ERROR: test failed: 1 == 3.`

`@test_approx_eq` tests whether the two numbers are approximately equal. `@test_approx_eq 1 1.1` returns `ERROR: assertion failed: |1 - 1.1| <= 2.220446049250313e-12` because they are not equal within the machine tolerance. However, you can give the interval as the last argument within which they should be equal to `@test_approx_eq_eps 1 1.1 0.2`, which returns nothing, so `1` and `1.1` are within `0.2` from each other.

Debugging

If you want to look up in the source code where and how a particular method is defined, use `@which`, for example: if `arr = [1, 2]` then `@which sort(arr)` returns `sort(v::AbstractArray{T,1}) at sort.jl:334`.

`@show` shows the expression and its result, which is handy for checking the embedded results: `456 * 789 + (@show 2 + 3) gives 2 + 3 => 5 359789.`

Benchmarking

For benchmarking purposes, we already know `@time` and `@elapsed`; `@timed` gives you the `@time` results as a tuple:

```
@time [x^2 for x in 1:1000] prints elapsed time: 3.911e-6 seconds (8064 bytes allocated) and returns 1000-element Array{Int64,1}: ....  
@timed [x^2 for x in 1:1000] returns ([1, 4, 9, 16, 25, 36, 49, 64, 81, 100 ... 982081, 984064, 986049, 988036, 990025, 992016, 994009, 996004, 998001, 1000000], 3.911e-6, 8064, 0.0).  
@elapsed [x^2 for x in 1:1000] returns 3.422e-6.
```

If you are specifically interested in the allocated memory, use `@allocated [x^2 for x in 1:1000]` which returns 8064.

To time code execution, call `tic()` to start timing, execute the function, and then use `toc()` or `toq()` to end the timer:

```
tic()  
[x^2 for x in 1:1000]
```

The `toc()` function prints elapsed time: 0.024395069 seconds.

Starting a task

Tasks (refer to the *Tasks* section in *Chapter 4, Control Flow*) are independent units of code execution. Often, we want to start executing them, and then continue executing the main code without waiting for the task result. In other words, we want to start the task *asynchronously*. This can be done with the `@async` macro:

```
a = @async 1 + 2 # Task (done) @0x000000002d70faf0  
consume(a) # 3
```

For a list of the built-in macros we encountered in this book, consult the list of macros in *Appendix, List of Macros and Packages*.

Reflection capabilities

We saw in this chapter that the code in Julia is represented by expressions that are data structures of type `Expr`. The structure of a program and its types can therefore be explored programmatically just like any other data. This means that a running program can dynamically discover its own properties, which is called **reflection**. We already have encountered many of these functions before:

- `typeof` and `subtypes` to query the type hierarchy (refer to *Chapter 6, More on Types, Methods, and Modules*)
- `methods(f)` to see all the methods of a function `f` (refer to *Chapter 3, Functions*)
- names and types: given a type `Person`:

```
type Person
    name:: String
    height::Float64
end
```

Then, `names(Person)` returns the field names as symbols: 2-element
`Array{Symbol,1}:` `:name` `:height`.

`Person.types` returns a tuple with the field types (`String`, `Float64`).

- To inspect how a function is represented internally, you can use `code_lowered`:

```
code_lowered(+, (Int, Int))
```

This returns the following output:

```
1-element Array{Any,1}:
:(($(Expr(:lambda, { :x, :y}, { {},{{{:x,:Any,0},{:y,:Any,0}}},{}},
:(begin # int.jl
, line 33:
return box(Int64,add_int(unbox(Int64,x),unbox(Int64,y)))
end))))
```

Or, you can use `code_typed` to see the type-inferred form:

```
code_typed(+, (Int, Int))
```

This returns the following:

```
1-element Array{Any,1}:
:(($(Expr(:lambda, { :x, :y}, { {},{{{:x,Int64,0},{:y,Int64,0}}},{}},
:(begin # int.
```

```
jl, line 33:  
    return box(Int64,add_int(x::Int64,y::Int64))::Int64  
end)::Int64)))
```



Using `code_typed` can show you whether your code is type optimized for performance: if the `Any` type is used instead of an appropriate specific type you would expect, then type indication in your code can certainly be improved, leading most likely to speed up the program's execution.

- To inspect the code generated by the LLVM engine, use `code_llvm`, and to see the assembly code generated, use `code_native` (refer to the *How Julia works* section in *Chapter 1, Installing the Julia Platform*).

While reflection is not necessary for many of the programs that you will write, it is very useful for IDEs to be able to inspect the internals of an object as well as for the tools generating the automatic documentation and for profiling tools. In other words, reflection is indispensable for tools that need to inspect the internals of the code objects programmatically.

Summary

In this chapter, we explored the expression format in which Julia is parsed. Because this format is a data structure, we can manipulate this in the code, and this is precisely what macros can do. We explored a number of them, and also some of the built-in ones that can be useful. In the next chapter, we will extend our vision to the network environment in which Julia runs, and we will explore its powerful capabilities for parallel execution.

8

I/O, Networking, and Parallel Computing

In this chapter, we will explore how Julia interacts with the outside world, reading from standard input and writing to standard output, files, networks, and databases. Julia provides asynchronous networking I/O using the `libuv` library. We will see how to handle data in Julia. We will also discover the parallel processing model of Julia.

In this chapter, the following topics are covered:

- Basic input and output
- Working with files (including the CSV files)
- Using DataFrames
- Working with TCP sockets and servers
- Interacting with databases
- Parallel operations and computing

Basic input and output

Julia's vision on input/output (I/O) is **stream-oriented**, that is, reading or writing streams of bytes. We will introduce different types of streams, such as file streams, in this chapter. **Standard input (stdin)** and **standard output (stdout)** are constants of the type `TTY` (an abbreviation for the old term, `Teletype`) that can be used in the Julia code to read from and write to (refer to the code in Chapter 8\io.jl):

- `read(STDIN, Char)`: This command waits for a character to be entered, and then returns that character; for example, when you type in `J`, this returns '`J`'

- `write(STDOUT, "Julia")`: This command types out **Julia5** (the added 5 is the number of bytes in the output stream; it is not added if the command ends in a semicolon (:))

`STDIN` and `STDOUT` are simply streams and can be replaced by any stream object in the read/write commands. `readbytes` is used to read a number of bytes from a stream into a vector:

- `readbytes(STDIN, 3)`: This command waits for an input, for example, abe reads 3 bytes from it, and then returns 3-element `Array{UInt8,1}`: `0x61 0x62 0x65`
- `readline(STDIN)`: This command reads all the inputs until a newline character `\n` is entered, for example, type `Julia` and press *ENTER*, this returns "`Julia\r\n`" on Windows and "`Julia\n`" on Linux

If you need to read all the lines from an input stream, use the `eachline` method in a `for` loop, for example:

```
stream = STDIN
for line in eachline(stream)
    print("Found $line")
    # process the line
end
```

For example:

```
First line of input
Found First line of input
2nd line of input
Found 2nd line of input
3rd line...
Found 3rd line...
```

To test whether you have reached the end of an input stream, use `eof(stream)` in combination with a `while` loop as follows:

```
while !eof(stream)
    x = read(stream, Char)
    println("Found: $x")
    # process the character
end
```

We can experiment with replacing `stream` by `STDIN` in these examples.

Working with files

To work with files, we need the `IOStream` type. `IOStream` is a type with the supertype `IO` and has the following characteristics:

- The fields are given by names (`IOStream`)

```
4-element Array{Symbol,1}: :handle    :ios      :name     :mark
```

- The types are given by `IOStream.types`

```
(Ptr{None}, Array{UInt8,1}, String, Int64)
```

The file handle is a pointer of the type `Ptr`, which is a reference to the file object.

Opening and reading a line-oriented file with the name `example.dat` is very easy:

```
// code in Chapter 8\io.jl
fname = "example.dat"
f1 = open(fname)
```

`fname` is a string that contains the path to the file, using escaping of special characters with `\` when necessary; for example, in Windows, when the file is in the `test` folder on the `D:` drive, this would become `d:\\test\\example.dat`. The `f1` variable is now an `IOStream(<file example.dat>)` object.

To read all lines one after the other in an array, use `data = readlines(f1)`, which returns 3-element `Array{Union(ASCIIString,UTF8String),1}`:

```
"this is line 1.\r\n"
>this is line 2.\r\n"
>this is line 3."
```

For processing line by line, now only a simple loop is needed:

```
for line in data
    println(line) # or process line
end
close(f1)
```

Always close the `IOStream` object to clean and save resources. If you want to read the file into one string, use `readall` (for example, see the program `word_frequency` in *Chapter 5, Collection Types*). Use this only for relatively small files because of the memory consumption; this can also be a potential problem when using `readlines`.

There is a convenient shorthand with the `do` syntax for opening a file, applying a function process, and closing it automatically. This goes as follows (`file` is the `iostream` object in this code):

```
open(fname) do file
    process(file)
end
```

As you can recall, in the *Map, filter, and list comprehensions* section in *Chapter 3, Functions*, `do` creates an anonymous function, and passes it to `open`. Thus, the previous code example would have been equivalent to `open(process, fname)`. Use the same syntax for processing a file `fname` line by line without the memory overhead of the previous methods, for example:

```
open(fname) do file
    for line in eachline(file)
        print(line) # or process line
    end
end
```

Writing a file requires first opening it with a "`w`" flag, then writing strings to it with `write`, `print`, or `println`, and then closing the file handle that flushes the `iostream` object to the disk:

```
fname = "example2.dat"
f2 = open(fname, "w")
write(f2, "I write myself to a file\n")
# returns 24 (bytes written)
println(f2, "even with println!")
close(f2)
```

Opening a file with the "`w`" option will clear the file if it exists. To append to an existing file, use "`a`".

To process all the files in the current folder (or a given folder as an argument to `readdir()`), use this `for` loop:

```
for file in readdir()
    # process file
end
```

Reading and writing CSV files

A CSV file is a comma-separated file. The data fields in each line are separated by commas "," or another delimiter such as semicolons ";" . These files are the de-facto standard for exchanging small and medium amounts of tabular data. Such files are structured so that one line contains data about one *data object*, so we need a way to read and process the file line by line. As an example, we will use the data file Chapter 8\winequality.csv that contains 1,599 sample measurements, 12 data columns, such as pH and alcohol per sample, separated by a semicolon. In the following screenshot, you can see the top 20 rows:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality	
1	7.4	0.7		0.19	0.076		11	34	0.9978	3.51	0.56	9.4	5
2	7.8	0.88		0.26	0.098		25	67	0.9968	3.2	0.68	9.8	5
3	7.8	0.76	0.04	2.3	0.092		15	54	0.997	3.26	0.65	9.8	5
4	11.2	0.28	0.56	1.9	0.075		17	60	0.998	3.16	0.58	9.8	6
5	7.4	0.7		0.19	0.076		11	34	0.9978	3.51	0.56	9.4	5
6	7.4	0.66		0.18	0.075		13	40	0.9978	3.51	0.56	9.4	5
7	7.9	0.6	0.06	1.6	0.069		15	59	0.9964	3.3	0.46	9.4	5
8	7.3	0.65		0.12	0.065		15	21	0.9946	3.39	0.47		10
9	7.8	0.58	0.02		2.073		9	18	0.9968	3.36	0.57	9.5	7
10	7.5	0.5	0.36	6.1	0.071		17	102	0.9978	3.35	0.8	10.5	5
11	6.7	0.58	0.08	1.8	0.097		15	65	0.9959	3.28	0.54	9.2	5
12	7.5	0.5	0.36	6.1	0.071		17	102	0.9978	3.35	0.8	10.5	5
13	5.6	0.615		0.16	0.089		16	59	0.9943	3.58	0.52	9.9	5
14	7.8	0.61	0.29	1.6	0.114		9	29	0.9974	3.26	1.56	9.1	5
15	8.9	0.62	0.18	3.8	0.176		52	145	0.9986	3.16	0.88	9.2	5
16	8.9	0.62	0.19	3.9	0.17		51	148	0.9986	3.17	0.93	9.2	5
17	8.5	0.28	0.56	1.8	0.092		35	103	0.9969	3.3	0.75	10.5	7
18	8.1	0.56	0.28	1.7	0.368		16	56	0.9968	3.11	1.28	9.3	5
19	7.4	0.59	0.08	4.4	0.086		6	29	0.9974	3.38	0.5		9
20													4

In general, the `readdlm` function is used to read in the data from the CSV files:

```
# code in Chapter 8\csv_files.jl:
fname = "winequality.csv"
data = readdlm(fname, ',')
```

The second argument is the delimiter character (here, it is ,). The resulting data is a 1600x12 `Array{Any, 2}` array of the type Any because no common type could be found:

"fixed acidity"	"volatile acidity"	"alcohol"	"quality"
7.4	0.7	9.4	5.0
7.8	0.88	9.8	5.0
7.8	0.76	9.8	5.0
...			

If the `data` file is comma separated, reading it is even simpler with the following command:

```
data2 = readcsv(fname)
```

The problem with what we have done until now is that the headers (the column titles) were read as part of the data. Fortunately, we can pass the argument `header=true` to let Julia put the first line in a separate array. It then naturally gets the correct datatype, `Float64`, for the data array. We can also specify the type explicitly, such as this:

```
data3 = readdlm(fname, ';', Float64, '\n', header=true)
```

The third argument here is the type of data, which is a numeric type, `String` or `Any`. The next argument is the line separator character, and the fifth indicates whether or not there is a header line with the field (column) names. If so, then `data3` is a tuple with the data as the first element and the header as the second, in our case, `(1599x12 Array{Float64,2}, 1x12 Array{String,2})` (There are other optional arguments to define `readdlm`, see the help option). In this case, the actual data is given by `data3[1]` and the header by `data3[2]`.

Let's continue working with the variable `data`. The data forms a matrix, and we can get the rows and columns of data using the normal array-matrix syntax (refer to the *Matrices* section in *Chapter 5, Collection Types*). For example, the third row is given by `row3 = data[3, :]` with `data: 7.8 0.88 0.0 2.6 0.098 25.0 67.0 0.9968 3.2 0.68 9.8 5.0,` representing the measurements for all the characteristics of a certain wine.

The measurements of a certain characteristic for all wines are given by a `data` column, for example, `col3 = data[:, 3]` represents the measurements of citric acid and returns a column vector `1600-element Array{Any,1}: "citric acid" 0.0 0.0 0.04 0.56 0.0 0.0 ... 0.08 0.08 0.1 0.13 0.12 0.47.`

If we need columns 2-4 (`volatile acidity` to `residual sugar`) for all wines, extract the data with `x = data[:, 2:4]`. If we need these measurements only for the wines on rows 70-75, get these with `y = data[70:75, 2:4]`, returning a `6 x 3 Array{Any,2}` output as follows:

```
0.32 0.57 2.0  
0.705 0.05 1.9  
...  
0.675 0.26 2.1
```

To get a matrix with the data from columns 3, 6, and 11, execute the following command:

```
z = [data[:,3] data[:,6] data[:,11]]
```

It would be useful to create a type `Wine` in the code.

For example, if the data is to be passed around functions, it will improve the code quality to encapsulate all the data in a single data type, like this:

```
type Wine
    fixed_acidity::Array{Float64}
    volatile_acidity::Array{Float64}
    citric_acid::Array{Float64}
    # other fields
    quality::Array{Float64}
end
```

Then, we can create objects of this type to work with them, like in any other object-oriented language, for example, `wine1 = Wine(data[1, :]...)`, where the elements of the row are splatted with the `...` operator into the `Wine` constructor.

To write to a CSV file, the simplest way is to use the `writecsv` function for a comma separator, or the `writedlm` function if you want to specify another separator. For example, to write an array `data` to a file `partial.dat`, you need to execute the following command:

```
writedlm("partial.dat", data, ',')
```

If more control is necessary, you can easily combine the more basic functions from the previous section. For example, the following code snippet writes 10 tuples of three numbers each to a file:

```
// code in Chapter 8\tuple_csv.jl
fname = "savetuple.csv"
 csvfile = open(fname, "w")
# writing headers:
write(csvfile, "ColName A, ColName B, ColName C\n")
for i = 1:10
    tup(i) = tuple(rand(Float64,3)...)
    write(csvfile, join(tup(i), ", "), "\n")
end
close(csvfile)
```

Using DataFrames

If you measure n variables (each of a different type) of a single object of observation, then you get a table with n columns for each object row. If there are m observations, then we have m rows of data. For example, given the student grades as data, you might want to know "compute the average grade for each socioeconomic group", where grade and socioeconomic group are both columns in the table, and there is one row per student.

The DataFrame is the most natural representation to work with such a $(m \times n)$ table of data. They are similar to pandas DataFrames in Python or `data.frame` in R. A DataFrame is a more specialized tool than a normal array for working with tabular and statistical data, and it is defined in the `DataFrames` package, a popular Julia library for statistical work. Install it in your environment by typing in `Pkg.add("DataFrames")` in the REPL. Then, import it into your current workspace with `using DataFrames`. Do the same for the packages `DataArrays` and `RDatasets` (which contains a collection of example datasets mostly used in the R literature).

A common case in statistical data is that data values can be missing (the information is not known). The `DataArrays` package provides us with the unique value `NA`, which represents a missing value, and has the type `NAtype`. The result of the computations that contain the `NA` values mostly cannot be determined, for example, `42 + NA` returns `NA`. (Julia v0.4 also has a new `Nullable{T}` type, which allows you to specify the type of a missing value). A `DataArray{T}` array is a data structure that can be n -dimensional, behaves like a standard Julia array, and can contain values of the type `T`, but it can also contain the missing (**Not Available**) values `NA` and can work efficiently with them. To construct them, use the `@data` macro:

```
// code in Chapter 8\dataarrays.jl
using DataArrays
using DataFrames
dv = @data([7, 3, NA, 5, 42])
```

This returns 5-element `DataArray{Int64,1}`: 7 3 NA 5 42.

The sum of these numbers is given by `sum(dv)` and returns `NA`. One can also assign the `NA` values to the array with `dv[5] = NA`; then, `dv` becomes [7, 3, NA, 5, NA]. Converting this data structure to a normal array fails: `convert(Array, dv)` returns `ERROR: NAException`.

How to get rid of these NA values, supposing we can do so safely? We can use the `dropna` function, for example, `sum(dropna(dv))` returns 15. If you know that you can replace them with a value `v`, use the `array` function:

```
repl = -1
sum(array(dv, repl)) # returns 13
```

A DataFrame is a kind of an in-memory database, versatile in the ways you can work with the data. It consists of columns with names such as `Col1`, `Col2`, `Col3`, and so on. Each of these columns are DataArrays that have their own type, and the data they contain can be referred to by the column names as well, so we have substantially more forms of indexing. Unlike two-dimensional arrays, columns in a DataFrame can be of different types. One column might, for instance, contain the names of students and should therefore be a string. Another column could contain their age and should be an integer.

We construct a DataFrame from the program data as follows:

```
// code in Chapter 8\dataframes.jl
using DataFrames
# constructing a DataFrame:
df = DataFrame()
df[:Col1] = 1:4
df[:Col2] = [e, pi, sqrt(2), 42]
df[:Col3] = [true, false, true, false]
show(df)
```

Notice that the column headers are used as symbols. This returns the following 4×3 DataFrame object:

show(df)			
4x3 DataFrame			
Row	Col1	Col2	Col3
1	1	2.71828	true
2	2	3.14159	false
3	3	1.41421	true
4	4	42.0	false

We could also have used the full constructor as follows:

```
df = DataFrame(Col1 = 1:4, Col2 = [e, pi, sqrt(2), 42],
               Col3 = [true, false, true, false])
```

You can refer to the columns either by an index (the column number) or by a name, both of the following expressions return the same output:

```
show(df[2])
show(df[:Col2])
```

This gives the following output:

```
[2.718281828459045, 3.141592653589793, 1.4142135623730951, 42.0]
```

To show the rows or subsets of rows and columns, use the familiar splice (:) syntax, for example:

- To get the first row, execute `df[1, :]`. This returns `1x3 DataFrame`.

Row	Col1	Col2	Col3
1	1	2.71828	true

- To get the second and third row, execute `df[2:3, :]`
- To get only the second column from the previous result, execute `df[2:3, :Col2]`. This returns `[3.141592653589793, 1.4142135623730951]`.
- To get the second and third column from the second and third row, execute `df[2:3, [:Col2, :Col3]]`, which returns the following output:

Row	Col2	Col3
1	3.14159	false
2	1.41421	true

The following functions are very useful when working with `DataFrames`:

- The `head(df)` and `tail(df)` functions show you the first six and the last six lines of data respectively.
- The `names` function gives the names of the columns `names(df)`. It returns 3-element `Array{Symbol,1}: :Col1 :Col2 :Col3`.
- The `eltypes` function gives the data types of the columns `eltypes(df)`. It gives the output as 3-element `Array{Type{T<:Top},1}: Int64 Float64 Bool`.

- The `describe` function tries to give some useful summary information about the data in the columns, depending on the type, for example, `describe(df)` gives for column 2 (which is numeric) the min, max, median, mean, number, and percentage of NAs:

```
Col2
Min      1.4142135623730951
1st Qu.  2.392264761937558
Median   2.929937241024419
Mean     12.318522011105483
3rd Qu.  12.856194490192344
Max     42.0
NAs      0
NA%     0.0%
```

To load in data from a local CSV file, use the method `readtable`. The returned object is of type `DataFrame`:

```
// code in Chapter 8\dataframes.jl
using DataFrames
fname = "winequality.csv"
data = readtable(fname, separator = ';')
typeof(data) # DataFrame
size(data) # (1599,12)
```

Here is a fraction of the output:

1599x12 DataFrame				
Row	fixed_acidity	volatile_acidity	citric_acid	residual_sugar
1	7.4	0.7	0.0	1.9
2	7.8	0.88	0.0	2.6
3	7.8	0.76	0.04	2.3
⋮				
1596	5.9	0.55	0.1	2.2
1597	6.3	0.51	0.13	2.3
1598	5.9	0.645	0.12	2.0
1599	6.0	0.31	0.47	3.6

The `readtable` method also supports reading in gzipped CSV files.

Writing a DataFrame to a file can be done with the `writetable` function, which takes the filename and the DataFrame as arguments, for example, `writetable("dataframe1.csv", df)`. By default, `writetable` will use the delimiter specified by the filename extension and write the column names as headers.

Both `readtable` and `writetable` support numerous options for special cases. Refer to the docs for more information (refer to <http://dataframesjl.readthedocs.org/en/latest/>). To demonstrate some of the power of DataFrames, here are some queries you can do:

- Make a vector with only the quality information `data[:quality]`
- Give the wines with alcohol percentage equal to 9.5, for example, `data[data[:alcohol] .== 9.5, :]`

Here, we use the `.==` operator, which does element-wise comparison.

`data[:alcohol] .== 9.5` returns an array of Boolean values (true for datapoints, where `:alcohol` is 9.5, and false otherwise). `data[boolean_array, :]` selects those rows where `boolean_array` is true.

- Count the number of wines grouped by quality with `by(data, :quality, data -> size(data, 1))`, which returns the following:

6x2 DataFrame		
Row	quality	x1
1	3	10
2	4	53
3	5	681
4	6	638
5	7	199
6	8	18

The `DataFrames` package contains the `by` function, which takes in three arguments:

- A `DataFrame`, here it takes `data`
- A column to split the `DataFrame` on, here it takes `quality`
- A function or an expression to apply to each subset of the `DataFrame`, here `data -> size(data, 1)`, which gives us the number of wines for each quality value

Another easy way to get the distribution among quality is to execute the histogram `hist` function `hist(data[:quality])` that gives the counts over the range of quality `(2.0:1.0:8.0, [10, 53, 681, 638, 199, 18])`. More precisely, this is a tuple with the first element corresponding to the edges of the histogram bins, and the second denoting the number of items in each bin. So there are, for example, 10 wines with quality between 2 and 3, and so on.

To extract the counts as a variable `count` of type `Vector`, we can execute `_ , count = hist(data[:quality])`; the `_` means that we neglect the first element of the tuple. To obtain the quality classes as a `DataArray` class, we will execute the following:

```
class = sort(unique(data[:quality]))
```

We can now construct a `df_quality` `DataFrame` with the `class` and `count` columns as `df_quality = DataFrame(qual=class, no=count)`. This gives the following output:

Row	qual	no
1	3	10
2	4	53
3	5	681
4	6	638
5	7	199
6	8	18

In the *Using Gadfly on data* section of *Chapter 10, The Standard Library and Packages*, we will see how to visualize `DataFrames`.

To deepen your understanding and learn about the other features of Julia `DataFrames` (such as joining, reshaping, and sorting), refer to the documentation available at <http://dataframesjl.readthedocs.org/en/latest/>.

Other file formats

Julia can work with other human-readable file formats through specialized packages:

- For JSON, use the `JSON` package. The `parse` method converts the JSON strings into Dictionaries, and the `json` method turns any Julia object into a JSON string.
- For XML, use the `LightXML` package
- For YAML, use the `YAML` package

- For HDF5 (a common format for scientific data), use the `HDF5` package
- For working with Windows `INI` files, use the `IniFile` package

Working with TCP sockets and servers

To send data over a network, the data has to conform to a certain format or protocol. The **Transmission Control Protocol (TCP/IP)** is one of the core protocols to be used on the Internet. The following screenshot shows how to communicate over TCP/IP between a Julia Tcp server and a client (see the code in Chapter 8\tcpserver.jl):

The screenshot displays two windows. The top-left window is a Julia REPL session showing code execution and responses. The bottom-right window is a terminal window showing the output of the netcat tool.

Julia Session (Top Left):

```
A fresh approach to technical computing
Documentation: http://docs.julialang.org
Type "help()" for help.

Version 0.3.0 (2014-08-20 20:43 UTC)
Official http://julialang.org release
x86_64-linux-gnu

Greetings! 你好! 안녕하세요?
55> server = listen(8080)
TcpServer(active)

56> conn = accept(server)
TcpSocket(open, 0 bytes waiting)

57> line = readline(conn)
"hello Julia server!\n"

58> write(conn, "Hello back from server to client, what can I do for you?")
59>
60> close(conn)
61>
```

Terminal Window (Bottom Right):

```
ivo@ubuntu:~$ nc localhost 8080
hello Julia server!
Hello back from server to client, what can I do for you?ivo@ubuntu:~$
```

The server (in the upper-left corner) is started in a Julia session with `server = listen(8080)` that returns a `TcpServer` object listening on the port 8080. The line `conn = accept(server)` waits for an incoming client to make a connection. Now, in a second terminal (in the lower-right corner), start the **netcat (nc)** tool at the prompt to make a connection with the Julia server on port 8080, for example, `nc localhost 8080`. Then, the `accept` function creates a `TcpSocket` object on which the server can read or write.

Then, the server issues the command `line = readline(conn)`, blocking the server until it gets a full line (ending with a newline character) from the client. The client types "hello Julia server!" followed by ENTER, which appears at the server console. The server can also write text to the client over the TCP connection with the `write(conn, "message")` function, which then appears at the client side. The server can, when finished, close the `TcpSocket` connection to close the TCP connection with `close(conn)`; this also closes the netcat session.

Of course, a normal server must be able to handle multiple clients. Here, you can see the code for a server that echoes back to the clients everything they send to the server:

```
// code in Chapter8\echoserver.jl
server = listen(8081)
while true
    conn = accept(server)
    @async begin
        try
            while true
                line = readline(conn)
                println(line) # output in server console
                write(conn, line)
            end
        catch ex
            print("connection ended with error $ex")
        end
    end # end coroutine block
end
```

To achieve this, we place the `accept()` function within an infinite `while` loop, so that each incoming connection is accepted. The same is true for reading and writing to a specific client; the server only stops listening to that client when the client disconnects. Because the network communication with the clients is a possible source of errors, we have to surround it within a `try/catch` expression. When an error occurs, it is bound to the `ex` object. For example, when a client terminal exits, you get the `connection ended with errorErrorException("stream is closed or unusable")` message.

However, we also see a `@async` macro here, what is its function? The `@async` macro starts a new coroutine (refer to the *Tasks* section in *Chapter 4, Control Flow*) in the local process to handle the execution of the `begin-end` block that starts right after it. So, the macro `@async` handles the connection with each particular client in a separate coroutine. Thus, the `@async` block returns immediately, enabling the server to continue accepting new connections through the outer `while` loop. Because coroutines have a very low overhead, making a new one for each connection is perfectly acceptable. If it weren't for the `async` block, the program would block it until it was done with its current client before accepting a new connection.

On the other hand, the `@sync` macro is used to enclose a number of `@async` (or `@spawn` or `@parallel` calls, refer to the *Parallel operations and computing* section), and the code execution waits at the end of the `@sync` block until all the enclosed calls are finished.

Start this server example by typing the following command:

```
julia echoserver.jl
```

We can experiment with a number of netcat sessions in separate terminals. Client sessions can also be made by typing in a Julia console:

```
conn = connect(8081) #> TcpSocket(open, 0 bytes waiting)
write(conn, "Do you hear me?\n")
```

The `listen` function has some variants, for example, `listen(IPv6(0), 2001)` creates a TCP server that listens on port 2001 on all IPv6 interfaces. Similarly, instead of `readline`, there are also simpler `read` methods:

- `read(conn, UInt8)`: This method blocks until there is a byte to read from `conn`, and then returns it. Use `convert(Char, n)` to convert a `UInt8` value into `Char`. This will let you see the ASCII letter for `UInt8` you read in.
- `read(conn, Char)`: This method blocks until there is a byte to read from `conn`, and then returns it.

The important aspect about the communication API is that the code looks like the synchronous code executing line by line, even though the I/O is actually happening asynchronously through the use of tasks. We don't have to worry about writing callbacks as in some other languages. For more details about the possible methods, refer to the *I/O and Network* section at <http://docs.julialang.org/en/latest/stdlib/base/>.

Interacting with databases

Open Database Connectivity (ODBC) is a low-level protocol for establishing connections with the majority of databases and datasources (for more details, refer to http://en.wikipedia.org/wiki/Open_Database_Connectivity).

Julia has an ODBC package that enables Julia scripts to talk to ODBC data sources. Install the package through `Pkg.add("ODBC")`, and at the start of the code, run using ODBC.

The package can work with a system **Data Source Name (DSN)** that contains all the concrete connection information, such as server name, database, credentials, and so on. Every operating system has its own utility to make DSNs. In Windows, the ODBC administrator can be reached by navigating to **Configuration | System Administration | ODBC Data Sources**; on other systems, you have IODBC or Unix ODBC.

For example, suppose we have a database called `pubs` running in a SQL Server or a MySQL Server, and the connection is described with a DSN `pubsODBC`. Now, I can connect to this database as follows:

```
// code in Chapter 8\odbc.jl
using ODBC
ODBC.connect("pubsODBC")
```

This returns an output as follows:

```
ODBC Connection Object
-----
Connection Data Source: pubsODBC
pubsODBC Connection Number: 1
Contains resultset? No
```

You can also store this connection object in a variable `conn` as follows:

```
conn = ODBC.connect("pubsODBC")
```

This way, you are able to close the connection when necessary through `disconnect(conn)` to save the database resources, or handle multiple connections.

To launch a query on the `titles` table, you only need to use the `query` function as follows:

```
results = query("select * from titles")
```

The result is of the type `DataFrame` and dimensions 18×10 , because the table contains 18 rows and 10 columns, for example, here are some of the columns:

Row	title					
Row	_type	pub_id	price	advance	royalty	ytd_sales
1	"The Busy Executive's Database Guide"					
2	"Cooking with Computers: Surreptitious Balance Sheets"					
3	"You Can Combat Computer Stress!"					
4	"Straight Talk About Computers"					
5	"Silicon Valley Gastronomic Treats"					
6	"The Gourmet Microwave"					
:						
1	"business"	"1389"	19.99	5000.0	10	4095
2	"business"	"1389"	11.95	5000.0	10	3876
3	"business"	"0736"	2.99	10125.0	24	18722
4	"business"	"1389"	19.99	5000.0	10	4095
5	"mod_cook"	"0877"	19.99	0.0	12	2032
6	"mod_cook"	"0877"	2.99	15000.0	24	22246

If you haven't stored the query results in a variable, you can always retrieve them from `conn.resultset`, where `conn` is an existing connection. Now we have all the functionalities of `DataFrames` at our disposal to work with this data. Launching data manipulation queries works in the same way:

```
updsql = "update titles set type = 'psychology' where  
        title_id='BU1032'"  
query(updsql)
```

When successful, the result is a 0×0 `DataFrame`. In order to see which ODBC drivers are installed on the system, ask for `listdrivers()`. The already available DSNs are listed with `listdsns()`.

Julia already has database drivers for Memcache, FoundationDB, MongoDB, Redis, MySQL, SQLite, and PostgreSQL (for more information, refer to <https://github.com/svaksha/Julia.jl/blob/master/Database.md#postgresql>).

Parallel operations and computing

In our multicore CPU and clustered computing world, it is imperative for a new language to have excellent parallel computing capabilities. This is one of the main strengths of Julia, providing an environment based on message passing between multiple processes that can execute on the same machine or on remote machines. In that sense, it implements the actor model (as Erlang, Elixir, and Dart do), but we'll see that the actual coding happens on a higher level than receiving and sending messages between processes, or workers (processors) as Julia calls them. The developer only needs to manage explicitly the main process from which all other workers are started. The message send and receive operations are simulated by higher-level operations that look like function calls.

Creating processes

Julia can be started as a REPL or as a separate application with a number of workers n available. The following command starts n processes on the local machine:

```
// code in Chapter 8\parallel1.jl
julia -p n    # starts REPL with n workers
```

These workers are different processes, not threads, so they do not share memory.

To get the most out of a machine, set n equal to the number of processor cores. For example, when n is 8, then you have, in fact, 9 workers: one for the REPL shell itself, and eight others that are ready to do parallel tasks. Every worker has its own integer identifier, which we can see by calling the `workers` function `workers()`. This returns the following:

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

Process 1 is the REPL worker. We can now iterate over the workers with the following command:

```
for pid in workers()
    # do something with each process (pid = process id)
end
```

Each worker can get its own process ID with the function `myid()`. If at a certain moment, you need more workers, adding new ones is easy:

```
addprocs(5)
```

This returns 5-element `Array{Any, 1}` that contains their process identifiers 10 11 12 13 14. The default method adds workers on the local machine, but the `addprocs` method accepts arguments to start processes on remote machines via SSH. This is the secure shell protocol that enables you to execute commands on a remote computer via a shell in a totally encrypted manner.

The number of available workers is given by `nprocs()`, in our case, this is 14. A worker can be removed by calling `rmprocs()` with its identifier, for example, `rmprocs(3)` stops the worker with ID 3.

All these workers communicate via TCP ports and run on the same machine, which is why it is called a local cluster. To activate workers on a cluster of computers, start Julia as follows:

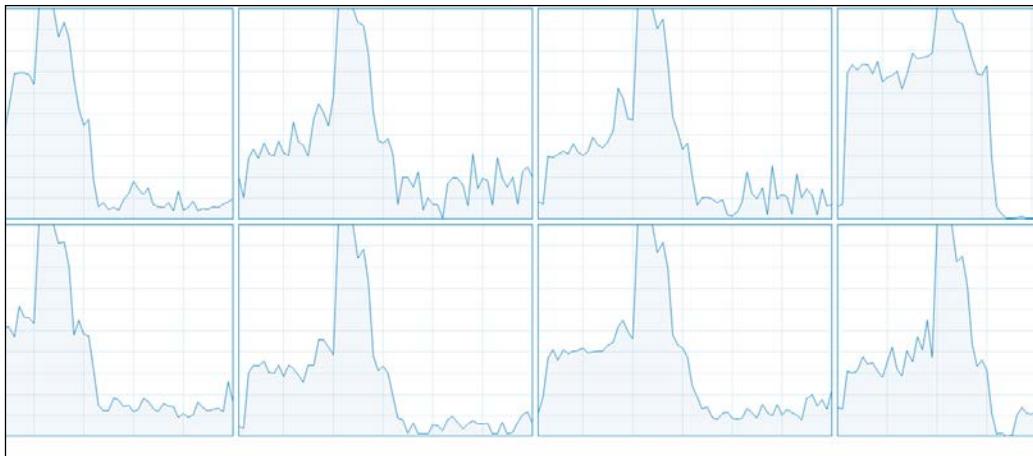
```
julia --machinefile machines driver.jl
```

Here, `machines` is a file that contains the names of the computers you want to engage, like this:

```
node01  
node01  
node02  
node02  
node03
```

Here `node01`, `node02`, and `node03` are the three names of computers in the cluster, and we want to start two workers each on `node01` and `node02`, and one worker on `node03`.

The `driver.jl` file is the script that runs the calculations and has the process identifier 1. This command uses a password-less SSH login to start the worker processes on the specified machines. The following screenshot shows all the eight processors on an eight core machine when engaged in a parallel operation:



The horizontal axis is time, and the vertical is the CPU usage. On each core, a worker process is engaged in a long-running Fibonacci calculation.

Processors can be dynamically added or removed to a master Julia process, both locally on symmetric multiprocessor systems, remotely on a computer cluster as well as in the cloud. If more versatility is needed, you can work with the `ClusterManager` type (see <http://docs.julialang.org/en/latest/manual/parallel-computing/>).

Using low-level communications

Julia's native parallel computing model is based on two primitives: **remote calls** and **remote references**. At this level, we can give a certain worker a function with arguments to execute with `remotecall`, and get the result back with `fetch`. As a trivial example in the following code, we call upon worker 2 to execute a square function on the number 1000:

```
r1 = remotecall(2, x -> x^2, 1000)
```

This returns `RemoteRef(2,1,20)`.

The arguments are: the worker ID, the function, and the function's arguments. Such a remote call returns immediately, thus not blocking the main worker (the REPL in this case). The main process continues executing while the remote worker does the assigned job. The `remotecall` function returns a variable `r1` of type `RemoteRef`, which is a reference to the computed result, that we can get using `fetch`:

```
fetch(r1) which returns 1000000
```

The call to `fetch` will block the main process until worker 2 has finished the calculation. The main processor can also run `wait(r1)`, which also blocks until the result of the remote call becomes available. If you need the remote result immediately in the local operation, use the following command:

```
remotecall_fetch(5, sin, 2pi) which returns -2.4492935982947064e-16
```

This is more efficient than `fetch(remotecall(...))`.

You can also use the `@spawnat` macro that evaluates the expression in the second argument on the worker specified by the first argument:

```
r2 = @spawnat 4 sqrt(2) # lets worker 4 calculate sqrt(2)
fetch(r2) # returns 1.4142135623730951
```

This is made even easier with `@spawn`, which only needs an expression to evaluate, because it decides for itself where it will be executed: `r3 = @spawn sqrt(5)` returns `RemoteRef(5,1,26)` and `fetch(r3)` returns `2.23606797749979`.

To execute a certain function on all the workers, we can use a comprehension:

```
r = [@spawnat w sqrt(5) for w in workers()]
fetch(r[3]) # returns 2.23606797749979
```

To execute the same statement on all the workers, we can also use the `@everywhere` macro:

```
@everywhere println(myid()) 1
From worker 2: 2
From worker 3: 3
From worker 4: 4
From worker 7: 7
From worker 5: 5
From worker 6: 6
From worker 8: 8
From worker 9: 9
```

All the workers correspond to different processes; they therefore do not share variables, for example:

```
x = 5 #> 5
@everywhere println(x) #> 5
# exception on 2 exception on : 4: ERROR: x not defined ...
```

The variable `x` is only known in the main process, all the other workers return the `ERROR: x not defined` error message.

`@everywhere` can also be used to make the data like the variable `w` that is available to all processors, for example, `@everywhere w = 8`.

The following example makes a source file `defs.jl` available to all the workers:

```
@everywhere include("defs.jl")
```

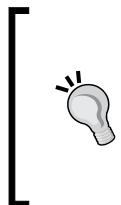
Or more explicitly a function `fib(n)` as follows:

```
@everywhere function fib(n)
    if (n < 2) then
        return n
    else return fib(n-1) + fib(n-2)
    end
end
```

In order to be able to perform its task, a remote worker needs access to the function it executes. You can make sure that all workers know about the functions they need by loading the source code `functions.jl` with `require`, making it available to all workers:

```
require("functions")
```

In a cluster, the contents of this file (and any files loaded recursively) will be sent over the network.



A best practice is to separate your code into two files: one file (`functions.jl`) that contains the functions and parameters that need to be run in parallel, and the other file (`driver.jl`) that manages the processing and collecting the results. Use the `require("functions")` command in `driver.jl` to import the functions and parameters to all processors.

An alternative is to specify the files to load on the command line. If you need the source files `file1.jl` and `file2.jl` on all the `n` processors at start-up time, use the syntax `julia -p n -L file1.jl -L file2.jl driver.jl`, where `driver.jl` is the script that organizes the computations.

Data movement between workers (such as when calling `fetch`) needs to be reduced as much as possible in order to get performance and scalability.

If every worker needs to know a variable `d`, this can be broadcast to all processes with the following code:

```
for pid in workers()
    remotecall(pid, x -> (global d; d = x; nothing), d)
end
```

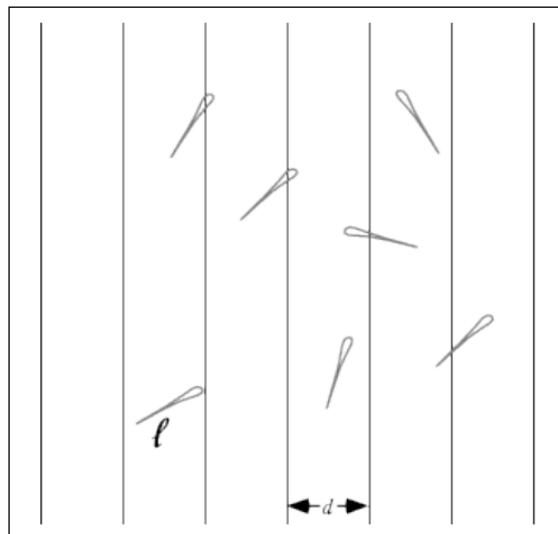
Each worker then has its local copy of data. Scheduling of the workers is done with tasks (refer to the *Tasks* section of *Chapter 4, Control Flow*), so that no locking is required, for example, when a communication operation such as `fetch` or `wait` is executed, the current task is suspended, and the scheduler picks another task to run. When the `wait` event completes (for example, the data shows up), the current task is restarted.

In many cases, however, you do not have to specify or create processes to do parallel programming in Julia, as we will see in the next section.

Parallel loops and maps

A `for` loop with a large number of iterations is a good candidate for parallel execution, and Julia has a special construct to do this: the `@parallel` macro, which can be used for the `for` loops and comprehensions.

Let's calculate an approximation for π using the famous Buffon's needle problem. If we drop a needle onto a floor with equal parallel strips of wood, what is the probability that the needle will cross a line between two strips? Let's take a look at the following screenshot:



Without getting into the mathematical intricacies of this problem (if you are interested, see http://en.wikipedia.org/wiki/Buffon's_needle), a function `buffon(n)` can be deduced from the model assumptions that return an approximation for π when throwing the needle n times (assuming the length of the needle l and the width d between the strips both equal to 1):

```
// code in Chapter 8\parallel_loops_maps.jl
function buffon(n)
    hit = 0
    for i = 1:n
        mp = rand()
        phi = (rand() * pi) - pi / 2 # angle at which needle falls
        xright = mp + cos(phi)/2 # x location of needle
        xleft = mp - cos(phi)/2
        # does needle cross either x == 0 or x == 1?
        p = (xright >= 1 || xleft <= 0) ? 1 : 0
        hit += p
    end
    miss = n - hit
    piapprox = n / hit * 2
end
```

With ever increasing n , the calculation time increases, because the number of the `for` iterations that have to be executed in one thread on one processor increases, but we also get a better estimate for π :

```
@time buffon(100000)
elapsed time: 0.005487779 seconds (96 bytes allocated)
3.1467321186947355
@time buffon(100000000)
elapsed time: 5.362294859 seconds (96 bytes allocated)
3.1418351308191026
```

However, what if we could spread the calculations over the available processors? For this, we have to rearrange our code a bit. In the sequential version, the variable `hit` is increased on every iteration inside the `for` loop with the amount `p` (which is 0 or 1). In the parallel version, we rewrite the code, so that this `p` is exactly the result of the `for` loop (one calculation) done on one of the processors engaged.

Julia also provides a `@parallel` macro that acts on a `for` loop, splitting the range, and distributing it to each process. It optionally takes a "reducer" as its first argument. If a reducer is specified, the results from each remote procedure will be aggregated using the reducer. In the following example, we use the `(+)` function as a reducer, which means that the last values of the parallel blocks on each worker will be summed to calculate the final value of `hit`:

```
function buffon_par(n)
    hit = @parallel (+) for i = 1:n
        mp = rand()
        phi = (rand() * pi) - pi / 2
        xright = mp + cos(phi)/2
        xleft = mp - cos(phi)/2
        (xright >= 1 || xleft <= 0) ? 1 : 0
    end
    miss = n - hit
    piapprox = n / hit * 2
end
```

On my machine with eight processors, this gives the following results:

```
@time buffon_par(100000)
elapsed time: 0.005903334 seconds (296920 bytes allocated)
3.136762860727729
@time buffon_par(100000000)
elapsed time: 0.849702686 seconds (300888 bytes allocated)
3.141665751394711
```

We see much better performance for the higher number of iterations (a factor of 6.3 in this case). By changing a normal `for` loop into a parallel reducing version, we were able to get substantial improvements in the calculation time, at the cost of higher memory consumption. In general, always test whether the parallel version really is an improvement over the sequential version in your specific case!

The first argument of `@parallel` is the reducing operator (here, `(+)`), the second is the `for` loop, which must start on the same line. The calculations in the loop must be independent from one another, because the order in which they run is arbitrary, given that they are scheduled over the different workers. The actual reduction (summing up in this case) is done on the calling process.

Any variables used inside the parallel loop will be copied (broadcasted) to each process. Because of this, the code like the following will fail to initialize the array `arr`, because each process has a copy of it:

```
arr = zeros(100000)
@parallel for i=1:100000
    arr[i] = i
end
```

After the loop, `arr` still contains all the zeros, because it is the copy on the master worker.

If the computational task is to apply a function to all elements in some collection, you can use a **parallel map** operation through the `pmap` function. The `pmap` function takes the following form: `pmap(f, coll)`, applies a function `f` on each element of the collection `coll` in parallel, but preserves the order of the collection in the result. Suppose we have to calculate the rank of a number of large matrices. We can do this sequentially as follows:

```
function rank_marray()
    marr = [rand(1000,1000) for i=1:10]
    for arr in marr
        println(rank(arr))
    end
end

@time rank_marray() # prints out ten times 1000
elapsed time: 4.351479797 seconds (166177728 bytes allocated,
1.43% gc time)
```

Here, parallelizing also gives benefits (a factor of 1.6):

```
function prank_marray()
    marr = [rand(1000,1000) for i=1:10]
    println(pmap(rank, marr))
end

@time prank_marray()
elapsed time: 2.785466798 seconds (163955848 bytes allocated, 1.96% gc
time)
```

The `@parallel` macro and `pmap` are both powerful tools to tackle **map-reduce** problems.

Distributed arrays

When computations have to be done on a very large array (or arrays), the array can be distributed, so that each process works in parallel on a different part of the array. In this way, we can make use of the memory resources of multiple machines, and allow the manipulation of arrays that would be too large to fit on one machine.

The specific data type used here is called a **distributed array** or **DArray**; most operations behave exactly as on the normal `Array` type, so the parallelism is invisible. With `DArray`, each process has local access to just a part of the data, and no two processes share the same data. For example, the following code creates a distributed array of random numbers with dimensions 100×100 and is divided over four workers. The data division given by the third argument says to divide the number of columns evenly over the four workers:

```
// code in Chapter 8\distrib_arrays.jl:  
arr = drand((100,100), workers()[1:4], [1,4])  
100x100 DArray{Float64,2,Array{Float64,2}}: ...
```

The following properties of the `DArray` `arr` makes this clear:

```
arr.pmap # on which workers ? 4-element Array{Int64,1}: 2 3 4 5  
arr.indexes # which worker has which data indices1x4  
Array{ (UnitRange{Int64},UnitRange{Int64}),2 }:  
(1:100,1:25)  (1:100,26:50)  (1:100,51:75)  (1:100,76:100)  
arr.cuts # where the data is partitioned  
2-element Array{Array{Int64,1},1}:  
[1,101]  
[1, 26, 51, 76, 101]  
arr.chunks # references on the workers:  
1x4 Array{RemoteRef,2}:  
RemoteRef(2,1,11164)  RemoteRef(3,1,11165)  ...  RemoteRef(5,1,11167)
```

`DArrays` can also be created with the `@parallel` macro as follows:

```
da = @parallel [2i for i = 1:10]  
# 10-element DArray{Int64,1,Array{Int64,1}}: ...
```

The following code snippet is often used to construct a distributed array divided over the available workers:

```
DArray((10,10)) do I
    println(I)
    return zeros(length(I[1]),length(I[2]))
end
```

(I is a tuple of index ranges, which is constructed automatically).

This returns the following output of a 10x10 array filled with zeros divided over the available workers:

```
From worker 2:  (1:5,1:3)
From worker 8:  (1:5,9:10)
From worker 4:  (1:5,4:5)
From worker 3:  (6:10,1:3)
From worker 5:  (6:10,4:5)
From worker 7:  (6:10,6:8)
From worker 6:  (1:5,6:8)
From worker 9:  (6:10,9:10)
10x10 DArray{Float64,2,Array{Float64,2}}:  0.0  0.0  0.0  0.0 ...
```

For more information on distributed arrays, refer to <http://docs.julialang.org/en/latest/manual/parallel-computing/#distributed-arrays>.

Julia's model for building a large parallel application works by means of a global distributed address space. This means that you can hold a reference to an object that lives on another machine participating in a computation. These references are easily manipulated and passed around between machines, making it simple to keep track of what's being computed where. Also, machines can be added in mid computation when needed.

Summary

In this chapter, we explored a lot of material. We learned how the I/O system in Julia is constructed, how to work with files and DataFrames, and how to connect with databases using ODBC. The basics of network programming in Julia was also discussed, and then we got an overview of the parallel computing functionality, from the primitive operations to map-reduce functions and distributed arrays. In the next chapter, we will take a look at how Julia interacts with the command line and with other languages, and discuss some performance tips.

9

Running External Programs

Sometimes, your code needs to interact with programs in the outside world, be it the operating system in which it runs, or other languages such as C or FORTRAN. This chapter shows how straightforward it is to run external programs from Julia and covers the following topics:

- Running shell commands – interpolation and pipelining
- Calling C and FORTRAN
- Calling Python
- Performance tips – a summary

Running shell commands

To interact with the operating system from within the Julia REPL, there are a few helper functions available as follows:

- `pwd()`: This function prints the current directory, for example, "d:\\test"
- `cd("d:\\test\\week1")`: This function helps to navigate to subdirectories
- In the interactive shell, you can also use the *shell mode* using the ; modifier:
 - ; `ls`: This prints, for example, `file1.txt shell.jl test.txt tosort.txt`
 - ; `mkdir folder`: This makes a directory named `folder`
 - ; `cd folder`: This helps to navigate to `folder`

However, what if you want to run a shell command by the operating system (the OS)? Julia offers an efficient shell integration through the `run` function, which takes an object of type `Cmd` that is defined by enclosing a command string in backticks (` `):

```
# Code in Chapter 9\shell.jl:  
cmd = `echo Julia is smart`  
typeof(cmd) #> Cmd  
run(cmd) # returns Julia is smart  
run(`date`) #> Sun Oct 12 09:44:50 GMT 2014  
cmd = `cat file1.txt`  
run(cmd) # prints the contents of file1.txt
```



Be careful to enclose the command text in backticks (` `), not single quotes (' ').



If the execution of `cmd` by the OS goes wrong, `run` throws a **failed process** error. You might want to first test the command before running it; `success(cmd)` will return true if it will execute successfully, otherwise it returns false.

Julia forks commands as **child processes** from the Julia process. Instead of immediately running the command in the shell, backticks create a `Cmd` object to represent the command, which can then be run, connected to other commands via pipes, and read or write to it.

Interpolation

String interpolation with the `$` operator is allowed in a command object like this:

```
file = "file1.txt"  
cmd = `cat $file` # equivalent to `cat file1.txt`  
run(cmd) #> prints the contents of file1.txt
```

This is very similar to the string interpolation with `$` in strings (refer to the *Strings* section in *Chapter 2, Variables, Types, and Operations*).

Pipelining

Julia defines a pipeline operator with symbol `|>` to redirect the output of a command as the input to the following command:

```
run(`cat $file` |> "test.txt")
```

This writes the contents of the file referred to by `$file` into `test.txt`, which is shown as follows:

```
run("test.txt" |> `cat`)
```

This pipeline operator can even be chained as follows:

```
run(`echo $("\nhi\nJulia")` |> `cat` |> `grep -n J`) #>  
3:Julia
```

If the file `tosort.txt` contains `B A C` on consecutive lines, then the following command will sort the lines:

```
run(`cat "tosort.txt" |> `sort`) # returns A B C
```

Another example is to search for the word "is" in all the text files in the current folder; use the following command:

```
run(`grep is $(readdir())`)
```

To capture the result of a command in Julia, use `readall` or `readline`:

```
a = readall(`cat "tosort.txt" |> `sort`)
```

Now `a` has the value "`A\r\nB\r\nC\r\n`".

Multiple commands can be run in parallel with the `&` operator:

```
run(`cat "file1.txt" & `cat "tosort.txt" `)
```

This will print the lines of the two files intermingled, because the printing happens concurrently.

Using this functionality requires careful testing, and probably, the code will differ according to the operating system on which your Julia program runs. You can obtain the OS from the variable `OS_NAME`, or use the macros `@windows`, `@unix`, `@linux`, and `@osx`, which were specifically designed to handle platform variations. For example, let's say we want to execute the function `fun1()` if we are on Windows, else the function `fun2()`. We can write this as follows:

```
@windows ? fun1() : fun2()
```

Calling C and FORTRAN

While Julia can rightfully claim to obviate the need to write some C or FORTRAN code, it is possible that you will need to interact with the existing C or FORTRAN shared libraries. Functions in such a library can be called directly by Julia, with no glue code, or boilerplate code or compilation needed. Because Julia's LLVM compiler generates native code, calling a C function from Julia has exactly the same overhead as calling the same function from C code itself. However, first, we need to know a few more things:

- For calling out to C, we need to work with pointer types; a native pointer `Ptr{T}` is nothing more than the memory address for a variable of type `T`
- At this lower level, the term `bitstype` is also used; `bitstype` is a concrete type whose data consists of bits, such as `Int8`, `UInt8`, `Int32`, `Float64`, `Bool`, and `Char`
- To pass a string to C, it is converted to a contiguous byte array representation with the function `bytestring()`; given `Ptr` to a C string, it returns a Julia string.

Here is how to call a C function in a shared library (calling FORTRAN is done similarly): suppose we want to know the value of an environment variable in our system, say the language, we can obtain this by calling the C function `getenv` from the shared library `libc`:

```
# code in Chapter 9\callc.jl:
lang = ccall( (:getenv, "libc"), Ptr{UInt8}, (Ptr{UInt8},),
"LANGUAGE")
```

This returns `Ptr{UInt8} @0x00007fff8d178dad`. To see its string contents, execute `bytestring(lang)`, which returns `en_US`.

In general, `ccall` takes the following arguments:

- A `(:function, "library")` tuple with the name of the C function (here, `getenv`) is used as a symbol, and the library name (here, `libc`) as a string
- The return type (here, `Ptr{UInt8}`), which can be any `bitstype`, or `Ptr`
- A tuple of types of the input arguments (here, `(Ptr{UInt8})`), note the tuple)
- The actual arguments if there are any (here, `"LANGUAGE"`)

It is generally advisable to test for the existence of a library before doing the call. This can be tested like this: `find_library(["libc"])`, which returns `"libc"`, when the library is found or `""` when it cannot find the library.

When calling a FORTRAN function, all inputs must be passed by reference. Arguments to C functions are, in general, automatically converted, and the returned values in C types are also converted to Julia types. Arrays of Booleans are handled differently in C and Julia and cannot be passed directly, so they must be manually converted. The same applies for some system dependent types (refer to the following references for more details).

The `ccall` function will also automatically ensure that all of its arguments will be preserved from garbage collection until the call returns. C types are mapped to Julia types, for example, `short` is mapped to `Int16`, and `double` to `Float64`.

A complete table of these mappings as well as a lot more intricate detail can be found in the Julia docs at <http://docs.julialang.org/en/latest/manual/calling-c-and-fortran-code/>. The other way around by calling Julia functions from C code (or embedding Julia in C) is also possible, refer to <http://docs.julialang.org/en/latest/manual/embedding/>. Julia and C can also share array data without copying. Another way that C code can call Julia code is in the form of callback functions (refer to <http://julialang.org/blog/2013/05/callback/>).

If you have the existing C code, you must compile it as a shared library to call it from Julia. With GCC, you can do this using the `-fPIC` command-line arguments. Support for C++ is more limited and is provided by the `Cpp` and `Clang` packages.

Calling Python

The `PyCall` package provides for calling Python from Julia code. As always, add this package to your Julia environment with `Pkg.add("PyCall")`. Then, you can start using it in the REPL or in a script as follows:

```
using PyCall
pyeval("10*10") #> 100
@pyimport math
math.sin(math.pi / 2) #> 1.0
```

As we can see with the `@pyimport` macro, we can easily import any Python library; functions inside such a library are called with the familiar dot notation.

For more details, refer to <https://github.com/stevengj/PyCall.jl>.

Performance tips

Throughout this book, we paid attention to performance. Here, we summarize some of the highlighted performance topics and give some additional tips. These tips need not always be used, and you should always benchmark or profile the code and the effect of a tip, but applying some of them can often yield a remarkable performance improvement. Using type annotations everywhere is certainly *not* the way to go, Julia's type inferring engine does that work for you:

- **Refrain from using global variables.** If unavoidable, make them constant, or at least annotate the types. It is better to use local variables instead; they are often only kept on the stack (or even in registers), especially if they are immutable.
- Structure your code around functions that do their work on local variables via the function arguments, and this returns their results rather than mutating the global objects.
- Type stability is very important:
 - Avoid changing the types of variables over time
 - The return type of a function should only depend on the type of the arguments

Even if you do not know the types that will be used in a function, but you do know it will always be of the same type T and U , then functions should be defined keeping that in mind, as in this code snippet:

```
function myFunc{T,U}(a::T, b::U, c::Int)
    # code
end
```

- If large arrays are needed, indicate their final size with `sizehint` from the start (refer to the *Ranges and Arrays* section of *Chapter 2, Variables, Types, and Operations*).
- If `arr` is a very large array that you no longer need, you can free the memory it occupies by setting `arr = nothing`. The occupied memory will be released the next time the garbage collector runs. You can force this to happen by invoking `gc()`.
- In certain cases (such as real-time applications), disabling garbage collection (temporarily) with `gc_disable()` can be useful.
- Use named functions instead of anonymous functions.
- In general, use small functions.

- Don't test for the types of arguments inside a function, use an argument type annotation instead.
- If necessary, code different versions of a function (several methods) according to the types, so that multiple dispatch applies. Normally, this won't be necessary, because the JIT compiler is optimized to deal with the types as they come.
- Use types for keyword arguments; avoid using the splat operator (...) for dynamic lists of keyword arguments.
- Using mutating APIs (functions with ! at the end) is helpful, for example, to avoid copying large arrays.
- Prefer array operations to comprehensions, for example, `x.^2` is considerably faster than `[val^2 for val in x]`.
- Don't use `try/catch` in the inner loop of a calculation.
- Use immutable types (cfr. package `ImmutableArrays`).
- Avoid using type `Any`, especially in collection types.
- Avoid using abstract types in a collection.
- Type annotate fields in composite types.
- Avoid using a large number of variables, large temporary arrays, and collections, because this provokes much garbage collection. Also, don't make copies of variables if you don't have to.
- Avoid using string interpolation (\$) when writing to a file, just write the values.
- Devectorize your code, that is, use explicit `for` loops on array elements instead of simply working with the arrays and matrices. (This is exactly the opposite advice as commonly given to R, MATLAB, or Python users.)
- If appropriate, use a parallel reducing form with `@parallel` instead of a normal `for` loop (refer to *Chapter 8, I/O, Networking, and Parallel Computing*).
- Reduce data movement between workers in a parallel execution as much as possible (refer to *Chapter 8, I/O, Networking, and Parallel Computing*).
- Fix deprecation warnings.
- Use the macro `@inbounds` so that no array bounds checking occur in expressions (if you are absolutely certain that no `BoundsError` occurs!).
- Avoid using `eval` at runtime.

In general, split your code in functions. Data types will be determined at function calls, and when a function returns. Types that are not supplied will be inferred, but the `Any` type does not translate to the efficient code. If types are stable (that is, variables stick to the same type) and can be inferred, then your code will run fast.

Tools to use

Execute a function with certain parameter values, and then use `@time` (refer to the *Generic functions and multiple dispatch* section in *Chapter 3, Functions*) to measure the elapsed time and memory allocation. If too much memory is allocated, investigate the code for type problems.

Experiment different tips and techniques in the script `array_product_benchmark.jl`. Use `code_typed` (refer to the *Reflection capabilities* section in *Chapter 7, Metaprogramming in Julia*) to see if type `Any` is inferred.

There is a **linter** tool (the `Lint` package) that can give you all kinds of warnings and suggestions to improve your code. Use it as follows:

```
Pkg.add("Lint")
using Lint
lintfile("performance.jl")
```

This produces the output as follows:

```
performance.jl [           ] 33 ERROR Use of undeclared symbol a
performance.jl [with_keyword ] 6 INFO Argument declared but not used:
name
performance.jl [           ] 21 INFO A type is not given to the field name,
which can be slow
```

Some useful type checking and type stability investigation can be done with the package `TypeCheck`, for example, checking the return types of a function or checking types in a loop.

A **profiler** tool is available in the standard library to measure the performance of your running code and identify possible bottleneck lines. This works through calling your code with the `@profile` macro (refer to <http://docs.julialang.org/en/latest/libraries/base/#stdlib-profiling>). The `ProfileView` package provides a nice graphical browser to investigate the profile results (follow the tutorial at <https://github.com/timholy/ProfileView.jl>).

For more tips, examples, and argumentation about performance, look up <http://docs.julialang.org/en/latest/manual/performance-tips/>.

A debugger can be found at <https://github.com/toivoh/Debug.jl>; it should be included in Julia v0.4.

Summary

In this chapter, we saw how easy it is to run commands at the operating system level. Interfacing with C is not that much more difficult, although it is somewhat specialized. Finally, we reviewed the best practices at our disposal to make Julia perform at its best. In the last chapter, we will get to know some of the more important packages when using Julia in real projects.

10

The Standard Library and Packages

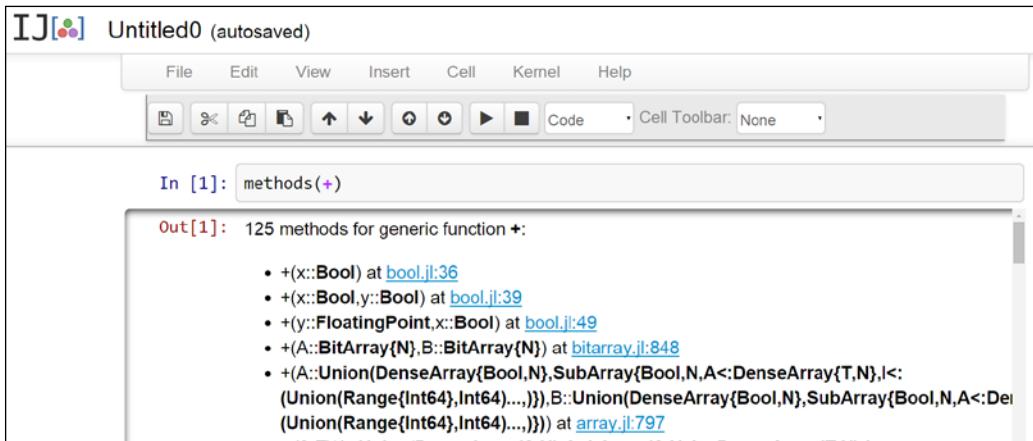
In this final chapter of our mini tour on Julia, we look anew at the standard library and explore the ever-growing ecosystem of packages for Julia. We will discuss the following topics:

- Digging deeper into the standard library
- Julia's package manager
- Publishing a package
- Graphics in Julia
- Using Gadfly on data

Digging deeper into the standard library

The standard library is written in Julia and comprises of a very broad range of functionalities: from regular expressions, working with dates and times (in v 0.4), a package manager, internationalization and Unicode, linear algebra, complex numbers, specialized mathematical functions, statistics, I/O and networking, **Fast Fourier Transformations (FFT)**, parallel computing, to macros, and reflection. Julia provides a firm and broad foundation for numerical computing and data science (for example, much of what NumPy has to offer is provided). Despite being targeted at numerical computing and data science, Julia aims to be a general purpose programming language.

The source code of the standard library can be found in the `share\julia\base` subfolder of Julia's root installation folder. Coding in Julia leads almost naturally to this source code, for example, when viewing all the methods of a particular function with `methods()`, or when using the `@which` macro to find out more about a certain method (refer to the *Generic functions and multiple dispatch* section in *Chapter 3, Functions*). IJulia even provides hyperlinks to the source code, as shown in the following screenshot:



The screenshot shows an IJulia notebook window titled "Untitled0 (autosaved)". The menu bar includes File, Edit, View, Insert, Cell, Kernel, and Help. The toolbar below has icons for file operations and a "Code" button. In the top-left cell, "In [1]: methods(+)". In the bottom-left cell, "Out[1]: 125 methods for generic function +:". Below this, a list of method signatures with their source file and line numbers:

- +(x::Bool) at [bool.jl:36](#)
- +(x::Bool,y::Bool) at [bool.jl:39](#)
- +(y::FloatingPoint,x::Bool) at [bool.jl:49](#)
- +(A::BitArray{N},B::BitArray{N}) at [bitarray.jl:848](#)
- +(A::Union(DenseArray{Bool,N},SubArray{Bool,N,A<:DenseArray{T,N},I<:(Union(Range{Int64},Int64,...,))),B::Union(DenseArray{Bool,N},SubArray{Bool,N,A<:DenseArray{T,N},I<:(Union(Range{Int64},Int64,...,))})) at [array.jl:797](#)

We covered some of the most important types and functions in the previous chapters, and you can refer to the manual for a more exhaustive overview at <http://docs.julialang.org/en/latest/stdlib/base/>.

It is certainly important to know that Julia contains a wealth of functional constructs to work with collections, such as the `reduce`, `fold`, `min`, `max`, `sum`, `any`, `all`, `map`, and `filter` functions. Some examples are as follows:

- `filter(f, coll)` applies the function `f` to all the elements of the collection `coll`:

```
# code in Chapter 10\stdlib.jl:  
filter(x -> iseven(x), 1:10)
```

This returns 5-element `Array{Int64,1}` that consists of 2, 4, 6, 8, and 10.

- `mapreduce(f, op, coll)` applies the function `f` to all the elements of `coll` and then reduces this to one resulting value by applying the operation `op`:

```
mapreduce(x -> sqrt(x), +, 1:10) #> 22.4682781862041  
# which is equivalent to:  
sum(map(x -> sqrt(x), 1:10))
```

- The pipeline operator (`|>`) also lets you write very functionally styled code. Using the form `x |> f`, it applies the function `f` to the argument `x`, and the results of this function can be chained to the following function. With this notation, we can rewrite the previous example as:

```
1:10 |> (x -> sqrt(x)) |> sum
```

Or, it can be written even shorter as follows:

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

When working in the REPL, it can be handy to store a variable in the operating system's clipboard if you want to clean the REPL's variables memory with `workspace()`. Consider the ensuing example:

```
a = 42
clipboard(a)
workspace()
a # returns ERROR: a not defined
a = clipboard() # returns "42"
```

This also works while copying information from another application, for example, a string from a website or from a text editor. On Linux, you will have to install `xclip` with the following command:

```
sudo apt-get install xclip
```

Julia's package manager

The *Packages* section in *Chapter 1, Installing the Julia Platform*, introduced us to the Julia's package system (some 370 packages and counting) and its manager program `Pkg`. Most Julia libraries are written exclusively in Julia; this makes them not only more portable, but also an excellent source for learning and experimenting with Julia in your own modified versions. The packages that are useful for the data scientists are `Stats`, `Distributions`, `GLM`, and `Optim`. You can search for applicable packages in the <http://pkg.julialang.org/indexorg.html> repository. For a list of the packages we encountered in this book, consult the *List of Packages* section in *Appendix, List of Macros and Packages*, after this chapter.

Installing and updating packages

It is advisable to regularly (and certainly, before installing a new package) execute the `Pkg.update()` function to ensure that your local package repository is up to date and synchronized, as shown in the following screenshot:

```
julia> Pkg.update()
INFO: Updating METADATA...
INFO: Updating cache of JSON...
INFO: Updating cache of Winston...
INFO: Updating cache of DataArrays...
INFO: Updating cache of Distributions...
INFO: Updating ANN...
INFO: Computing changes...
```

As we saw in *Chapter 1, Installing the Julia Platform*, packages are installed via `Pkg.add("PackageName")` and brought into scope using `PackageName`. This presumes the package is published on the METADATA repository; if this is not the case, you can clone it from a git repository as follows: `Pkg.clone("git@github.com:EricChiang/ANN.jl.git")`.

An alternative way is to add one or more package names to the REQUIRE file in your Julia home folder, and then execute `Pkg.resolve()` to install them and their dependencies.

If you need to force a certain package to a certain version (perhaps an older version), use `Pkg.pin()`, for example, use `Pkg.pin("HDF5", v"0.4.3")` to force the use of Version 0.4.3 of package HDF5, even when you already have v 0.4.4 installed.

Publishing a package

All package management in Julia is done via GitHub. Here are the steps for publishing your own package:

1. Fork the package METADATA.jl on GitHub, get the address of your fork, and execute the following:

```
$ git clone git@github.com:your-user-name/METADATA.jl.git
$ cd METADATA.jl
```

2. Make a new branch with the following commands:

```
$ git branch mypack  
$ git checkout mypack
```

3. Add the stuff for your package in a folder, say MyPack. Your code should go in a /src folder, which should also contain mypack.jl, that will be run when the command using MyPack is issued. The tests should go in the /tests folder. You should have a runtests.jl file in the folder that runs the tests for your package.

A text file named REQUIRE is where any dependencies on other Julia packages go; it is also where you specify compatible versions of Julia.

For example, it can contain the following:

```
julia 0.3-  
BinDeps  
@windows WinRPM
```

This certifies that this package is compatible with Julia v0.3 or higher; it needs the package BinDeps, and on Windows it needs the package WinRPM.

The license you want goes in LICENSE.md, and some documentation goes in README.md.

Then, you will have to run the following commands:

```
$ git add MyPack/*  
$ git commit -m "My fabulous package"
```

4. Then, push it to GitHub:

```
$ git push --set-upstream origin mypack
```

5. Go to the GitHub website of your fork of METADATA.jl and a green button **Compare & pull request** should appear, and you're just a few clicks away from finishing the pull request.

For more details, refer to <http://docs.julialang.org/en/release-0.3/manual/packages/#publishing-your-package>.

Graphics in Julia

Several packages exist to plot data and visualize data relations, which are as follows:

- **Winston:** (refer to the *Packages* section in *Chapter 1, Installing the Julia Platform*) This package offers 2D MATLAB-like plotting through an easy `plot(x, y)` command. Add a graphic to an existing plot with `oplot()`, and save it in the PNG, EPS, PDF, or SVG format with `savefig()`. From within a script, use `display(p1)`, where `p1` is the plot object to make the plot appear. For a complete code example, refer to `Chapter 10\winston.jl` (use it in the REPL). For more information, see the excellent docs at <http://winston.readthedocs.org/en/latest/> and <https://github.com/nolta/Winston.jl> for the package itself.
- **PyPlot:** (refer to the *Installing and working with IJulia* section in *Chapter 1, Installing the Julia Platform*) This package needs Python and `matplotlib` installed and works with no overhead through the `PyCall` package.

Here is a summary of the main commands:

- `plot(y), plot(x, y)` plots `y` versus `x` using the default line style and color
 - `semilogx(x, y), semilogy(x, y)` for log scale plots
 - `title("A title"), xlabel("x-axis"), and ylabel("foo")` to set labels
 - `legend(["curve 1", "curve 2"], "northwest")` to write a legend at the upper-left side of the graph
 - `grid(), axis("equal")` adds grid lines, and uses equal `x` and `y` scaling
 - `title(L"the curve $e^{\sqrt{x}}$")` sets the title with LaTeX equation
 - `savefig("fig.png"), savefig("fig.eps")` saves as the PNG or EPS image
- **Gadfly:** This provides ggplot2-like plotting package using data-driven documents (`d3`) and is very useful for statistical graphs. It renders publication quality graphics to PNG, PostScript, PDF, and SVG, for the last mode interactivity such as panning, zooming, and toggling. Here are some plotting commands (refer to `Chapter 10\gadfly.jl`, and use it in the REPL):

```
draw(SVG("gadfly.svg", 6inch, 3inch), plot([x -> x^2], 0, 25))
pl = plot([x -> cos(x)/x], 5, 25)
draw(PNG("gadfly.png", 300, 100), pl)
```

We'll examine a concrete example in the next section. For more information, refer to <http://gadflyjl.org/>.

Using Gadfly on data

Let's apply Gadfly to visualize the histogram we made in the *Using DataFrames* section of *Chapter 8, I/O, Networking, and Parallel Computing*, when examining the quality of wine samples:

```
# see code in Chapter 8\DataFrame.jl:  
using Gadfly  
p = plot(df_quality, x="qual", y="no",  
         Geom.bar(), Guide.title("Class distributions (\\"quality\\")"))  
draw(PNG(14cm,10cm), p)
```

This produces the following output:



Here is an example to explore medical data: `medical.csv` is a file that contains the following columns: `IX`, `Sex`, `Age`, `sBP`, `dBP`, `Drink`, and `BMI` (`IX` is a number for each data line, `sBP` and `dBP` are systolic and diastolic blood pressure, `Drink` indicates whether the person drinks alcohol, and `BMI` is the body mass index). The following code reads in the data in a DataFrame `df` file that contains 50 lines and seven columns:

```
# code in Chapter 10\medical.jl
using Gadfly, DataFrames
df = readtable("medical.csv")
print("size is ", size(df)) #> size is (50,7)
df[1:3, 1:size(df,2)]
# data sample:
IX  Sex  Age   sBP    dBp  Drink  BMI
0   1    39   106.0  70.0  0     26.97
1   2    46   121.0  81.0  0     28.73
2   1    48   127.5  80.0  1     25.34
```

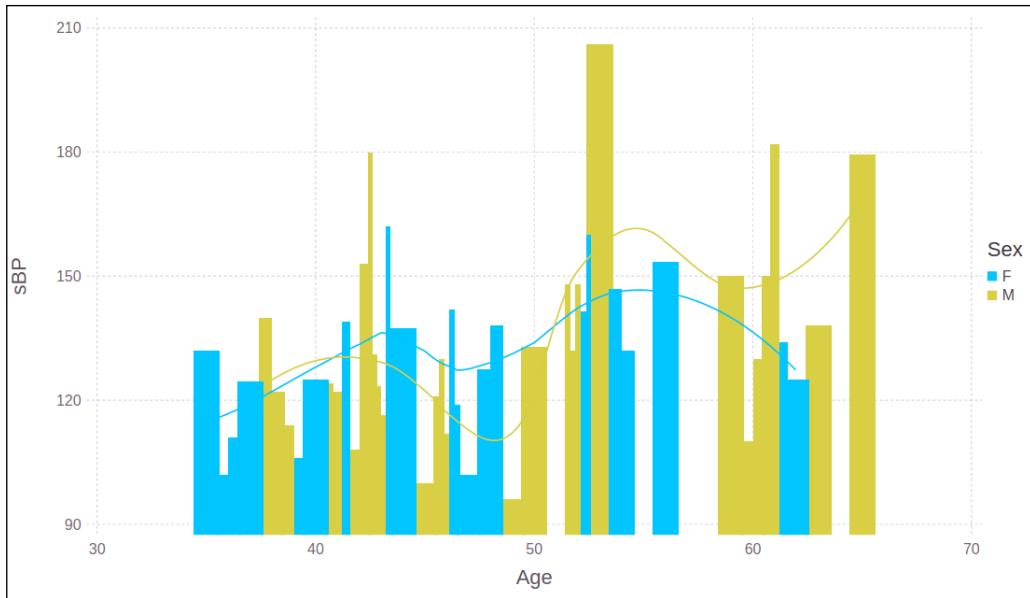
Let's transform our data a bit. The data for `Sex` contains 1 for female, 2 for male. Let's change that to F and M respectively. Similarly, change 0 to N and 1 to Y for the `Drink` data. `DataFrames` has a handy `ifelse` function ready for just this purpose:

```
# transforming the data:
df[:Sex] = ifelse(df[:Sex]==1, "F", "M")
df[:Drink] = ifelse(df[:Drink]==1, "Y", "N")
df[1:3, 1:size(df,2)]
# transformed data sample:
IX  Sex  Age   sBP    dBp  Drink  BMI
0   F    39   106.0  70.0  N     26.97
1   M    46   121.0  81.0  N     28.73
2   F    48   127.5  80.0  Y     25.34
```

Use `describe(df)` to get some statistical numbers on the data. For example, the standard deviation on the `Age` value is given by `std(df["Age"])` that gives 8.1941.

Let's plot systolic blood pressure versus age, using a different color for male and female, and apply data smoothing to draw a continuous line through the histogram rendered in a browser:

```
set_default_plot_size(20cm, 12cm)
plot(df, x="Age", y="sBP", color="Sex", Geom.smooth,
      Geom.bar(position=:dodge))
```



If you want to save the image in a file, give the plot a name and pass that name to the `draw` function:

```
pl = plot(df, x="Age", y="sBP", color="Sex", Geom.smooth, Geom.
bar(position=:dodge))
draw(PDF("medical.pdf", 6inch, 3inch), pl)
```

Lots of other plots can be drawn in Gadfly, such as scatter plots, 2D histograms, and box plots.

Summary

In this chapter, we looked at the built-in functionality Julia has to offer in its standard library. We also peeked at some of the more useful packages to apply in the data sciences.

We hope that this whirlwind overview of Julia has shown you why Julia is a rising star in the world of scientific computing and (big) data applications and that, you will take it up in your projects.

List of Macros and Packages

Macros

Chapter	Name	Section
2	@printf	The <i>Formatting numbers and strings</i> section under the <i>Strings</i> heading
	@sprintf	The <i>Formatting numbers and strings</i> section under the <i>Strings</i> heading
3	@which	<i>Generic functions and multiple dispatch</i>
	@time	<i>Generic functions and multiple dispatch</i>
	@elapsed	<i>Generic functions and multiple dispatch</i>
4	@task	Tasks
7	@assert	The <i>Testing</i> section under the <i>Built-in macros</i> heading
	@test	The <i>Testing</i> section under the <i>Built-in macros</i> heading
	@test_approx_eq	The <i>Testing</i> section under the <i>Built-in macros</i> heading
	@test_approx_eq_eps	The <i>Testing</i> section under the <i>Built-in macros</i> heading
	@show	The <i>Debugging</i> section under the <i>Built-in macros</i> heading
	@timed	The <i>Benchmarking</i> section under the <i>Built-in macros</i> heading
	@allocated	The <i>Benchmarking</i> section under the <i>Built-in macros</i> heading
	@async	The <i>Starting a task</i> section under the <i>Built-in macros</i> heading (also refer to <i>Chapter 8, I/O, Networking, and Parallel Computing</i>)

List of Macros and Packages

Chapter	Name	Section
8	@data	<i>Using DataFrames</i>
	@spawnat	<i>Parallel Programming, Using low-level communications</i>
	@async	<i>Working with TCP Sockets and servers</i>
	@sync	<i>Working with TCP Sockets and servers</i>
	@spawn	<i>Parallel Programming, Using low-level communications</i>
	@spawnat	<i>Parallel Programming, Using low-level communications</i>
	@everywhere	<i>Parallel Programming, Using low-level communications</i>
	@parallel	<i>Parallel Programming, Parallel loops and maps</i>
9	@windows	<i>Running External Programs</i>
	@unix	<i>Running External Programs</i>
	@linux	<i>Running External Programs</i>
	@osx	<i>Running External Programs</i>
	@inbounds	<i>Performance tips</i>
	@profile	<i>Performance tips</i>

List of packages

Chapter	Name	Section
The Rationale for Julia	MATLAB	<i>A comparison with other languages for the data scientist</i>
	Rif	<i>A comparison with other languages for the data scientist</i>
	PyCall	<i>A comparison with other languages for the data scientist</i>
1	Winston	<i>Packages</i>
	IJulia	<i>Installing and working with IJulia</i>
	PyPlot	<i>Installing and working with IJulia</i>
	ZMQ	<i>Installing Sublime-IJulia</i>
	Jewel	<i>Installing Juno</i>
2	Dates	<i>Dates and Times (<= v 0.3)</i>
	TimeZones	<i>Dates and Times (>= v 0.4)</i>
5	ImmutableArrays	<i>Matrices</i>
	Compat	<i>Dictionaries</i>

Chapter	Name	Section
8	DataFrames	<i>Using DataFrames</i>
	DataArrays	<i>Using DataFrames</i>
	RDatasets	<i>Using DataFrames</i>
	JSON	<i>Using DataFrames</i>
	LightXML	<i>Using DataFrames</i>
	YAML	<i>Using DataFrames</i>
	HDF5	<i>Using DataFrames</i>
	IniFile	<i>Using DataFrames</i>
	ODBC	ODBC
9	Cpp	<i>Calling C and FORTRAN</i>
	Clang	<i>Calling C and FORTRAN</i>
	Lint	<i>Performance tips</i>
	TypeCheck	<i>Performance tips</i>
	ProfileView	<i>Performance tips</i>
10	Gadfly	<i>Graphics in Julia</i>

Module 2

Julia High Performance

Design and develop high performing programs with Julia

1

Julia is Fast

In many ways, the history of programming languages has often been driven by, and certainly intertwined, with the needs of numerical and scientific computing. The first high-level programming language, Fortran, was created with scientific computing in mind, and continues to be important in the field even to this day. In recent years, the rise of data science as a specialty has brought additional focus to scientific computing, particularly for statistical uses. In this area, somewhat counterintuitively, both specialized languages such as R and general-purpose languages such as Python are in widespread use. The rise of Hadoop and Spark has spread the use of Java and Scala respectively among this community. In the midst of all this, Matlab has had a strong niche within engineering and communities, while Mathematica remains unparalleled for symbolic operations.

A new language for scientific computing therefore has a very high barrier to overcome. It's been only a few short years since the Julia language was introduced into the world. In this time, its innovative features, which make it a dynamic language, based on multiple dispatch as its defining paradigm, has created growing niche within the numerical computing world. However, its the claim of high performance that excited its early adopters the most.

This, then, is a book that celebrates writing high-performance programs. With Julia, this is not only possible, but also reasonably straightforward, within a low-overhead, dynamic language.

As a reader of this book, you have likely already written your first few Julia programs. We will assume that you have successfully installed Julia, and have a working programming environment available. We expect you are familiar with very basic Julia syntax, but we will discuss and review many of those concepts throughout the book as we introduce them.

- Julia – fast and dynamic
- Designed for speed
- How fast can Julia be?

Julia – fast and dynamic

It is a widely believed myth in programming language communities that high-performance languages and dynamic languages are completely disjoint sets. The perceived wisdom is that, if you want programmer productivity, you should use a dynamic language, such as Ruby, Python or R. On the other hand, if you want fast code execution, you should use a statically typed language such as C or Java.

There are always exceptions to this rule. However, for most mainstream programmers, this is a strongly held belief.

This usually manifests itself in what is known as the "two language" problem. This is something that is especially prominent in scientific computing. This is the situation where the performance-critical inner kernel is written in C, but is then wrapped and used from a dynamic, higher-level language. Code written in traditional, scientific computing environments such as R, Matlab or NumPy follows this paradigm.

Code written in this fashion is not without its drawbacks however. Even though it looks like this gets you the best of both worlds – fast computation, while allowing the programmer to use a high-level language – this is a path full of hidden dangers. For one, someone will have to write the low-level kernel. So, you need two different skillsets. If you are lucky to find the low level code in C for your project, you are fine. However, if you are doing anything new or original, or even slightly different from the norm, you will find yourself writing both C and a high-level language. This severely limits the number of contributors that your projects or research will get: to be really productive, they have to be familiar with two languages.

Secondly, when running code routinely written in two languages, there can be severe and unforeseen performance pitfalls. When you can drop down to C code quickly, everything is fine. However, if, for whatever reason, your code cannot call into a C routine, you'll find your program taking hundreds or even thousands of times more longer than you expected.

Julia is the first modern language to make a reasonable effort to solve the "two language" problem. It is a high-level, dynamic, language with powerful features that make for a very productive programmer. At the same time, code written in Julia usually runs very fast, almost as fast as code written in statically typed languages.

The rest of this chapter describes some of the underlying design decisions that make Julia such a fast language. We also see some evidence of the performance claims for Julia.

The rest of the book shows you how to write your Julia programs in a way that optimizes its time and memory usage to the maximum. We will discuss how to measure and reason performance in Julia, and how to avoid potential performance pitfalls.

For all the content in this book, we will illustrate our point individually with small and simple programs. We hope that this will enable you grasp the crux of the issue, without getting distracted by unnecessary elements of a larger program. We expect that this methodology will therefore provide you with an instinctive intuition about Julia's performance profile.

Julia has a refreshingly simple performance model – and thus writing fast Julia code is a matter of understanding a few key elements of computer architecture, and how the Julia compiler interacts with it. We hope that, by the end of this book, your instincts are well developed to design and write your own Julia code with the fastest possible performance.

Versions of Julia

Julia is a fast moving project, with an open development process. All the code and examples in this book are targeted at version 0.4 of the language, which is the currently released version at the time of publication. Check Packt's website for changes and errata for future versions of Julia.

Designed for speed

When the creators of Julia launched the language into the world, they said the following in a blog post entitled *Why We Created Julia*, which was published in early 2012:

"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 interactive and we want it compiled.

(Did we mention it should be as fast as C?)"

High performance, indeed nearly C-level performance, has therefore been one of the founding principles of the language. It's built from the ground up to enable a fast execution of code.

In addition to being a core design principle, it has also been a necessity from the early stages of its development. A very large part of Julia's standard library, including very basic low-level operations, is written in Julia itself. For example, the + operation to add two integers is defined in Julia itself. (Refer to: <https://github.com/JuliaLang/julia/blob/1986c5024db36b4c921130351597f5b4f9f81691/base/int.jl#L8>). Similarly, the basic for loop uses the standard iteration mechanism available to all user-defined types. This means that the implementation had to be very fast from the very beginning to create a usable language. The creators of Julia did not have the luxury of escaping to C for even the core elements of the library.

We will note throughout the book many design decisions that have been made with an eye to high performance. But there are three main elements that create the basis for Julia's speed.

JIT and LLVM

Julia is a **Just In Time** (JIT) compiled language, rather than an interpreted one. This allows Julia to be dynamic, without having the overhead of interpretation. This compilation infrastructure is build on top of **Low Level Virtual Machine (LLVM)** (<http://llvm.org>).



The LLVM compiler without infrastructure project originated at University of Illinois. It now has contributions from a very large number of corporate as well as independent developers. As a result of all this work, it is now a very high-quality, yet modular, system for many different compilation and code generation activities.

Julia uses LLVM for its JIT compilation needs. The Julia runtime generates LLVM **Intermediate Representation (IR)** and hands it over to LLVM's JIT compiler, which in turn generates machine code that is executed on the CPU. As a result, sophisticated compilation techniques that are built into LLVM are ready and available to Julia, from the simple (such as *Loop Unrolling* or *Loop Deletion*) to state-of-the-art (such as *SIMD Vectorization*) ones. These compiler optimizations form a very large body of work, and in this sense, the existence of LLVM is very much a pre-requisite to the existence of Julia. It would have been an almost impossible task for a small team of developers to build this infrastructure from scratch.

Just-In-Time compilation

Just-in-Time compilation is a technique in which the code in a high-level language is converted to machine code for execution on the CPU at runtime. This is in contrast to interpreted languages, whose runtime executes the source language directly. This usually has a significantly higher overhead. On the other hand, **Ahead of Time (AOT)** compilation refers to the technique of converting source language into machine code as a separate step prior to running the code. In this case, the converted machine code can usually be saved to disk as an executable file.

Types

We will have much more to say about types in Julia throughout this book. At this stage, suffice it to say that Julia's concept of types is a key ingredient in its performance.

The Julia compiler tries to infer the type of all data used in a program, and compiles different versions of functions specialized to particular types of its arguments. To take a simple example, consider the `sqrt` function. This function can be called with integer or floating-point arguments. Julia will compile two versions of the code, one for integer arguments, and one for floating point arguments. This means that, at runtime, fast, straight-line code without any type checks will be executed on the CPU.

The ability of the compiler to reason about types is due to the combination of a sophisticated dataflow-based algorithm, and careful language design that allows this information to be inferred from most programs before execution begins. Put in another way, the language is designed to make it easy to statically analyze.

If there is a single reason for Julia is being such a high-performance language, this is it. This is why Julia is able to run at C-like speeds while still being a dynamic language. *Type inference* and *code specialization* are as close to a secret sauce as Julia gets. It is notable that, outside this type inference mechanism, the Julia compiler is quite simple. It does not include many advanced Just in Time optimizations that Java and JavaScript compilers are known to use. When the compiler has enough information about the types within the code, it can generate optimized, straight-line, code without many of these advanced techniques.

It is useful to note here that unlike some other optionally typed dynamic languages, simply adding type annotations to your code does not usually make Julia go any faster. Type inference means that the compiler is, in most cases, able to figure out the types of variables when necessary. Hence you can usually write high-level code without fighting with the compiler about types, and still achieve superior performance.

How fast can Julia be?

The best evidence of Julia's performance claims is when you write your own code. However, we can provide an indication of how fast Julia can be by comparing a similar algorithm over multiple languages.

As an example, let's consider a very simple routine to calculate the power sum for a series, as follows:

$$\sum_{n=1}^{1000} \frac{1}{n^2}$$

The following code runs this computation in Julia 500 times:

```
function pisum()
    sum = 0.0
    for j = 1:500
        sum = 0.0
        for k = 1:10000
            sum += 1.0/(k*k)
        end
    end
    sum
end
```

You will notice that this code contains no type annotations. It should look quite familiar to any modern dynamic language. The same algorithm implemented in C would look something similar to this:

```
double pisum() {
    double sum = 0.0;
    for (int j=0; j<500; ++j) {
        sum = 0.0;
        for (int k=1; k<=10000; ++k) {
            sum += 1.0/(k*k);
        }
    }
    return sum;
}
```

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

- Log in or register to our website using your e-mail address and password
- Let the mouse pointer hover on the **SUPPORT** tab at the top
- Click on **Code Downloads & Errata**
- Enter the name of the book in the **Search** box
- Select the book for which you're looking to download the code files
- Choose from the drop-down menu where you purchased this book from
- Click on **Code Download**



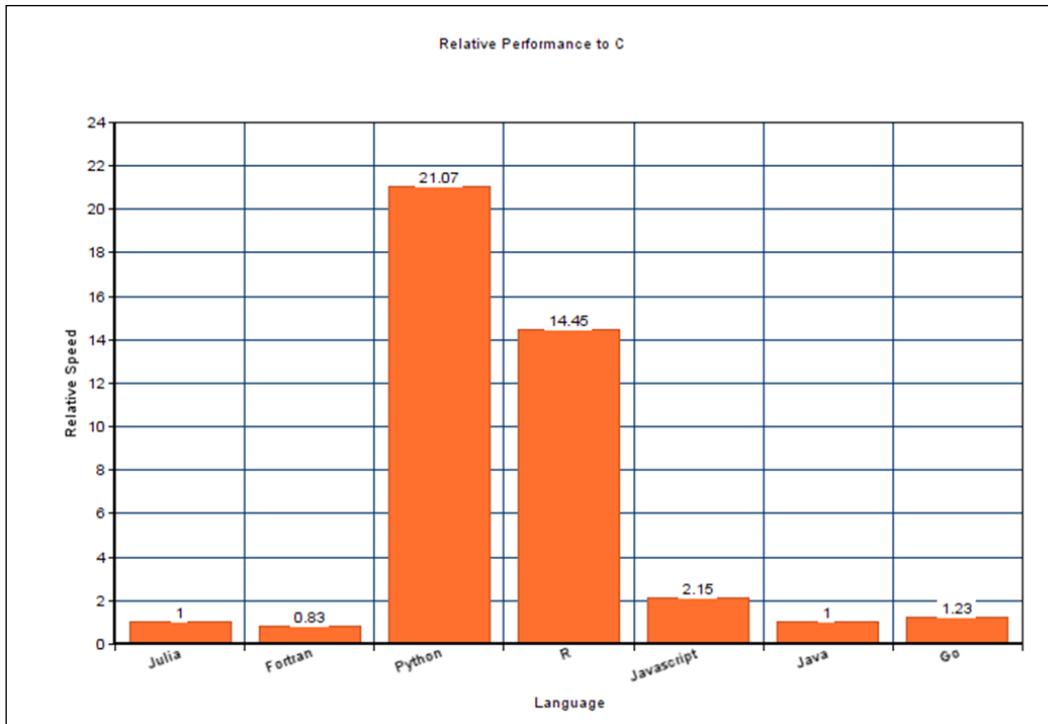
You can also download the code files by clicking on the **Code Files** button on the book's webpage at the Packt Publishing website. This page can be accessed by entering the book's name in the **Search** box. Please note that you need to be logged in to your Packt account.

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

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

By timing this code, and its re-implementation in many other languages (all of which are available at <https://github.com/JuliaLang/julia/tree/master/test/perf/micro>), we can note that Julia's performance claims are certainly borne out in this limited test. Julia can perform at a level similar to C and other statically typed and compiled languages.

This is of course a micro benchmark, and should therefore not be extrapolated too much. However, I hope you will agree that it is possible to achieve excellent performance in Julia. The rest of the book will attempt to show how you can achieve performance close to this standard in your code.



Summary

In this chapter, you noted that Julia is a language that is built from the ground up for high performance. Its design and implementation have always been focused on providing the highest possible performance on the modern CPU.

The rest of the book will show you how to use the power of Julia to the maximum, to write the fastest possible code in this language. In the next chapter, we will discuss how to measure the speed of Julia code, and identify performance bottlenecks. You will learn some of the tools that are built into Julia for this purpose.

2

Analyzing Julia Performance

Before we can try and optimize any Julia code we have written, we first need to understand its performance characteristics. Is the code fast enough for our needs? If not, how much slower is it from what it needs to be? And finally, can we understand where the bottlenecks are, so that we can prioritize where to focus our optimization effort? This chapter will show us the tools available in Julia to answer these questions and more. In later chapters, we will take a look at how to use this information to optimize our code.

In this chapter we will cover the following topics:

- Timing Julia functions
- Accurate benchmarking
- Profiling Julia functions
- Tracking detailed memory allocation

Timing Julia code

The first step to understanding anything is to measure it. The same goes for writing high-performance Julia code; we need to measure the performance of the code as the first step to achieving this. Fortunately Julia makes this extremely easy for us. There are simple ways to measure the time taken by any Julia code built into the Julia runtime. Moreover, if you want to perform statistically accurate benchmarking, there are high-quality packages available.

Tic and Toc

The simplest way to measure time in Julia is using the `tic()` and `toc()` functions. Place these functions respectively before and after any piece of Julia code, and we will note the time taken by this code on the console. Run the following code:

```
julia> tic(); sqrt(rand(1000)); toc();
elapsed time: 0.000137693 seconds
```

In the preceding code, we measured the time taken to generate 1,000 random numbers, and to compute its square root. Technically, all the `toc()` function does is print the `elapsed time` value since the last invocation of `tic()`. The time printed in this case (and other cases) is the actual `elapsed time` value, not the time spent by the CPU on the process. In other words, this is the wall-clock time. In particular, this time can be affected by any other CPU's intensive processes running on the machine at the same time.

Using these functions might be convenient when running a script, but it is not convenient during interactive development on Julia's **Read Eval Print Load (REPL)** console. Therefore, the most common way to measure the elapsed time of Julia code is to use the `@time` macro, which we will discuss next.

The `@time` macro

Whenever you care about the performance of your code (which you should do all the time), the `@time` macro will end up being one of your most used commands on the Julia prompt. Built into the runtime, this macro wraps the provided expression to calculate and print the elapsed time while running it. It also measures and outputs the amount of memory allocated while running this code as follows:

```
julia> @time sqrt(rand(1000));
0.000023 seconds (8 allocations: 15.969 KB)
```

Any kind of Julia expression can be wrapped by the `@time` macro. Usually, it is a function call as before, but it could be any other valid expression as follows:

```
julia> s=0
0

julia> @time for i=1:1000
           s=s+sqrt(i)
       end
0.001270 seconds (2.40 k allocations: 54.058 KB)
```



Timing measurements and JIT compiling

Recall that Julia is a JIT compiled language. The Julia compiler and runtime compiles any Julia code into machine code the first time it sees it. This means that, if you measure the execution time of any Julia expression that executes for the first time, you will end up measuring the time (and memory use) required to compile this code. So, whenever you time any piece of Julia code, it is crucial to run it at least once, prior to measuring the execution time. Always measure the second or later invocation.

The @timev macro

An enhanced version of the `@time` macro is also available: the `@timev` macro. This macro operates in a very similar manner to `@time`, but measures some additional memory statistics, and provides elapsed time measurements with nanosecond precision. Take a look at the following code:

```
julia> @timev sqrt(rand(1000));
0.000012 seconds (8 allocations: 15.969 KB)
elapsed time (ns): 11551
bytes allocated: 16352
pool allocs: 6
non-pool GC allocs:2
```

Both the `@time` and `@timev` macros return the value of the expression whose performance they measured. Hence, these can be added without side-effects to almost any location within the Julia code.

The Julia profiler

The Julia runtime includes a built-in profiler that can be used to measure which lines of code contribute the most to the total execution time of a codebase. It can therefore be used to identify bottlenecks in code, which can in turn be used to prioritize optimization efforts.

This built-in system is what is known as a *sampling profiler*. Its work is to inspect the call stack of the running system every few milliseconds (by default, 1 millisecond on UNIX and 10 milliseconds on Windows), and identify each line of code that contributes to this call stack. The idea is that the lines of code that are executed most often are found more often on the call stack. Hence, over many such samples, the count of how often each line of code is encountered will be a measure of how often this code runs.

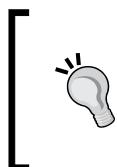
The primary advantage of a sampling profiler is that it can run without modifying the source program, and thus has a very minimal overhead. The program runs at almost full speed when being profiled. The downside of the profiler is that the data is statistical in nature, and may not reflect exactly how the program performed. However, when sampled over a reasonable period of time (say a few hundred milliseconds at least), the results are accurate enough to provide a good understanding of how the program performs, and what its bottlenecks are.

Using the profiler

The profiler code lives within the `profile` module within Julia. So the first step in using the profiler is to import its namespace into the current session. You can do this via the following code.

```
julia> using Base.Profile
```

This makes the `@profile` macro available to measure and store the performance profile of the expression supplied to it.



Do not profile the JIT

As with measuring the time of execution, remember to run your code at least once before attempting to profile it. Otherwise, you will end up profiling the Julia JIT compiler, rather than your code.

To see how the profiler works, let's start with a test function that creates 1,000 sets of 10,000 random numbers, and then computes the standard deviation of each set. Run the following:

```
function testfunc()
    x = rand(10000, 1000)
    y = std(x, 1)
    return y
end
```

After calling the function once to ensure that all the code is compiled, we can run the profiler over this code. as follows:

```
julia> @profile testfunc()
```

This will execute the expression while collecting profile information. The expression will return as usual, and the collected profile information will be stored in memory.

```
julia> Profile.print()
34 REPL.jl; anonymous; line: 93
```

```
34 REPL.jl; eval_user_input; line: 63
34 profile.jl; anonymous; line: 16
21 random.jl; rand!; line: 347
21 dSFMT.jl; dsfmt_fill_array_close_open!; line: 76
12 statistics.jl; var; line: 169
1 reducedim.jl; reduced_dims; line: 19
6 statistics.jl; mean; line: 31
6 reducedim.jl; sum!; line: 258
6 reducedim.jl; _mapreducedim!; line: 197
4 reduce.jl; mapreduce_pairwise_impl; line: 111
2 reduce.jl; mapreduce_pairwise_impl; line: 111
...
2 reduce.jl; mapreduce_pairwise_impl; line: 112
...
2 reduce.jl; mapreduce_pairwise_impl; line: 112
2 reduce.jl; mapreduce_pairwise_impl; line: 111
...
5 statistics.jl; varm!; line: 152
5 statistics.jl; centralize_sumabs2!; line: 117
4 reduce.jl; mapreduce_pairwise_impl; line: 111
4 reduce.jl; mapreduce_pairwise_impl; line: 112
2 reduce.jl; mapreduce_pairwise_impl; line: 111
2 reduce.jl; mapreduce_pairwise_impl; line: 111
2 reduce.jl; mapreduce_pairwise_impl; line: 108
2 simdloop.jl; mapreduce_seq_impl; line: 67
2 reduce.jl; mapreduce_pairwise_impl; line: 112
...
```

As you can note, the output from the profiler is a hierarchical list of code locations, representing the call stack for the program. The number against each line counts the number of times this line was sampled by the profiler. Therefore, the higher the number, the greater the contribution of that line to the total runtime of the program. It indicates the time spent on the line, and all its *callees*.

What does this output tell us? Well, among other things, it shows that the creation of the random arrays took most of the execution time, about two-thirds. For the remainder of the calculation of the standard deviation, the time was evenly split between the computation of the mean and variance.

There are a few profiler options that are sometimes useful, although the defaults are a good choice for most use cases. Primary among them is the *sampling interval*. This can be provided as keyword arguments to the `Profile.init()` method. The default delay is 1 millisecond on Linux, and should be increased for very long-running programs through the following line of code:

```
julia> Profile.init(delay=.01)
```

The delay may be reduced as well, but the overhead of profiling can increase significantly if it is lowered too much.

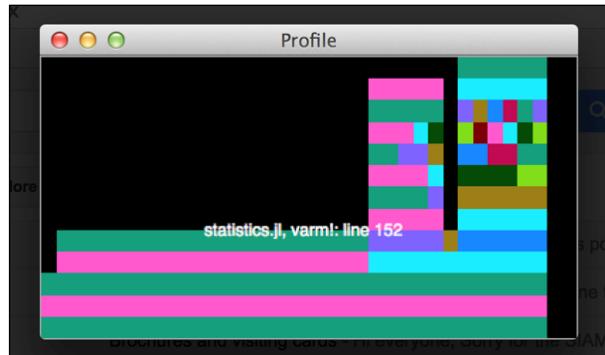
Finally, you may have realized that the profiler stores its samples in memory to be viewed later. In order to profile a different program during an existing Julia session, it may be necessary to clear the stored profile from memory. The `Profile.clear()` function does this, and must therefore be run between any two invocations of `@profile` within the same Julia process.

ProfileView

The textual display of the profiler output shown before is useful and elucidating in many cases, but can get confusing if read for long, or deeply nested call graphs. In this case, or in general if you would prefer a graphical output, the `ProfileView` package provides such an output. However, this is not built in to the base of Julia, and must be installed as an external package

```
Pkg.add("ProfileView")
```

This will install the `ProfileView` package and its dependencies (which include the Tk graphical environment). Once installed, its usage is very simple. Simply call the `ProfileView.view()` function instead of `Profile.print()` after the profile samples have been collected using `@profile`. A user interface window will pop up, with the profile displayed as a *flame graph*, looking similar to the following screenshot. Move your cursor over the blocks to note a hover containing the details of the call location:



This view provides the same information as the tree view seen earlier, but may be easier to navigate and understand, particularly for larger programs. In this chart, elapsed time goes from left to right, while the call stack goes from bottom to top. The width of the bar therefore shows the time spent by the program in a particular call location and its callees. The bars stacked on top of one another show a call from one to the other.

Analyzing memory allocation

The amount of memory used by a program is sometimes as important to track as the amount of time taken to run it. This is not only because memory is a limited resource that can be in short supply, but also because excessive allocation can easily lead to excessive execution time. The time taken to allocate and de-allocate memory and run the garbage collection can become quite significant when a program uses large amounts of memory.

The `@time` macro seen in the previous sections provides information about memory allocation for the expression or function being timed. In some cases however it may be difficult to predict where exactly in the code the memory allocation occurs. For these situations, Julia's track allocation functionality is just what is needed.

Using the memory allocation tracker

To get Julia to track memory allocation, start the `julia` process with the `-track-allocation=user` option as follows:

```
julia> track -allocation=user
```

This will start a normal Julia session in which you can run your code as usual. However, in the background, Julia will track all the memory used, which will be written to `.mem` files when Julia exits. There will be a new `.mem` file for each `.jl` file that is loaded and executed. These files will contain the Julia code from their corresponding source files, with each line annotated with the total amount of memory that was allocated as a result of executing this line.

As we discussed before, when running Julia code, the compiler will compile user code at runtime. Once again, we do not want to measure the memory allocation due to the compiler. To achieve this, first run the code under measurement once, after starting the Julia process. Then run the `Profile.clear_malloc_data()` function to restart the allocation measurement counters. Finally, run the code under measurement once again, and then exit the process. This way, we will get the most accurate memory measurements.

Statistically accurate benchmarking

The tools described in this chapter, particularly the `@time` macro, are useful to identify and investigate bottlenecks in our program. However, they are not very accurate for a fine-grained analysis of fast programs. If you want to, for example, compare two functions that take a few milliseconds to run, the amount of error and variability in the measurement will easily swamp the running time of this function.

Using Benchmarks.jl

The solution then is to use the `Benchmarks.jl` package for statistically accurate benchmarking. This package is not yet published in the official repository, but is stable and high-quality nevertheless. It can be installed with `Pkg.clone("https://github.com/johnmyleswhite/Benchmarks.jl.git")` and the subsequent usage is simple. Instead of using `@time`, as before, simply use `@benchmark`. Unlike `@time` however, this macro can only be used in front of function calls, rather than any expression. It will evaluate the parameters of the function separately, and then call the function multiple times to build up a sample of execution times.

The output will show the mean time taken to run the code, but with statistically accurate upper and lower bounds. These statistics are computed using an ordinary least squares fit of the measured execution time to estimate the expected distribution. Take a look at the following:

```
julia> using Benchmarks
julia> @benchmark sqrt(rand(1000))
=====
Time per evaluation: 9.48 μs [9.26 μs, 9.69 μs]
Proportion of time in GC: 5.43% [4.22%, 6.65%]
Memory allocated: 15.81 kb
Number of allocations: 4 allocations
Number of samples: 6601
Number of evaluations: 1080001
R² of OLS model: 0.913
Time spent benchmarking: 10.28 s
```

Summary

In this chapter, we discussed how to use the available tools to measure the performance of Julia code. You learned to measure the time and memory resources used by code, and understood the hotspots for any program.

In subsequent chapters, you will learn how to fix the issues that we identified using these tools, and make our Julia programs perform at their fastest.

3

Types in Julia

Julia is a dynamically typed language in which, unlike languages such as Java or C, the programmer does not need to specify the fixed type of every variable in the program. Yet, somewhat counterintuitively, Julia achieves its impressive performance characteristics by inferring and using type information for all the data in the program. In this chapter, we will start with a brief look at the type system in the language and then explain how to use this type system to write high-performance code.

- The Julia type system
- Type-stability
- Types at storage locations

The Julia type system

Types in Julia are essentially tag-on values that restrict the range of potential values that can possibly be stored at this location. Being a dynamic language, these tags are relevant only to runtime values. Types are not enforced at compile time (except in rare cases); rather, they are checked at runtime. However, type information is used at compile time to generate specialized methods and different kinds of function argument.

Using types

In most dynamic languages, types are usually implicit in how values are created. Julia can, and usually is, written in this way—with no explicit type annotations. However, additionally in Julia, you can specify that variables or function parameters should be restricted to specific types using the `::` symbol. Here's an example:

```
foo(x::Integer) = "an integer"    #Declare type of function argument
foo(x::ASCIIString) = "a string"
```

```
function bar(a, b)
    x::Int64 = 0                      #Declare type of local variable
    y = a+b                           #Type of variable will be inferred
    return y
end

julia> foo(1)                         #Dispatch on type of argument
"an integer"

julia> foo("1")                       #Dispatch on type of argument
"a string"

julia> foo(1.5)                        #Dispatch fails
ERROR: `foo` has no method matching foo(::Float64)
```



A note on terminology

In Julia, an abstract operation represented by a name is called a function, while the individual implementations for specific types are called methods. Thus, in the preceding code, we can talk of the `foo` function and the `foo` methods for `Integer` and `ASCIIString`.

Multiple dispatch

If there were one unifying strand through the design of the Julia language, it would be *multiple dispatch*. Simply put, dispatch is the process of selecting a function to be executed at runtime. Multiple dispatch, then, is the method of determining the function to be called based on the types of all the parameters of the function. Thus, one of the most important uses of types in Julia programs is to arrange the appropriate method dispatch by specifying the types of function arguments.

Note that this is different from the concept of method overloading. Dispatch is a runtime process, while method overloading is a compile-time concept. In most traditional object-oriented languages, dispatch at runtime occurs only on the runtime type of the *receiver* of the method (for example, the object before the dot)—hence the term "single dispatch."

Julia programs, therefore, usually contain many small function definitions for different types of arguments. It is good practice, however, to constrain argument types to the widest level possible. Use tight constraints only when you know that the method will fail on other types. Otherwise, write your method to accept unconstrained types and depend on the runtime to dispatch nested calls to the correct methods.

As an example, consider the following function to compute the sum of the square of two numbers:

```
sumsqr(x, y) = x^2 + y^2
```

In this code, we do not specify any type constraints for the `x` and `y` arguments of our `sumsqr` function. The base library will contain different `+` and `^` methods for integers and floats, and the runtime will dispatch to the correct method based on the types of the arguments. Take a look at the output:

```
julia> sumsqr(1, 2)
5

julia> sumsqr(1.5, 2.5)
8.5

julia> sumsqr(1 + 2im, 2 + 3im)
-8 + 16im
```

Abstract types

Types in Julia can be concrete or abstract. Abstract types cannot have any instantiated values. In other words, they can only be the nodes of the type hierarchy, not its leaves. They represent sets of related types. For example, Julia contains integer types for 32-bit and 64-bit integers—`Int32` and `Int64`, respectively. Both these types therefore inherit from the `Integer` abstract type.

Abstract types are defined using the `abstract` keyword. The inheritance relationship between types is denoted using the `<:` symbol followed by the name of the parent (or super) type. As an example, shown here are the abstract types defined as the basis of Julia's number types:

```
abstract Number
abstract Real      <: Number
abstract FloatingPoint <: Real
abstract Integer    <: Real
abstract Signed     <: Integer
abstract Unsigned   <: Integer
```

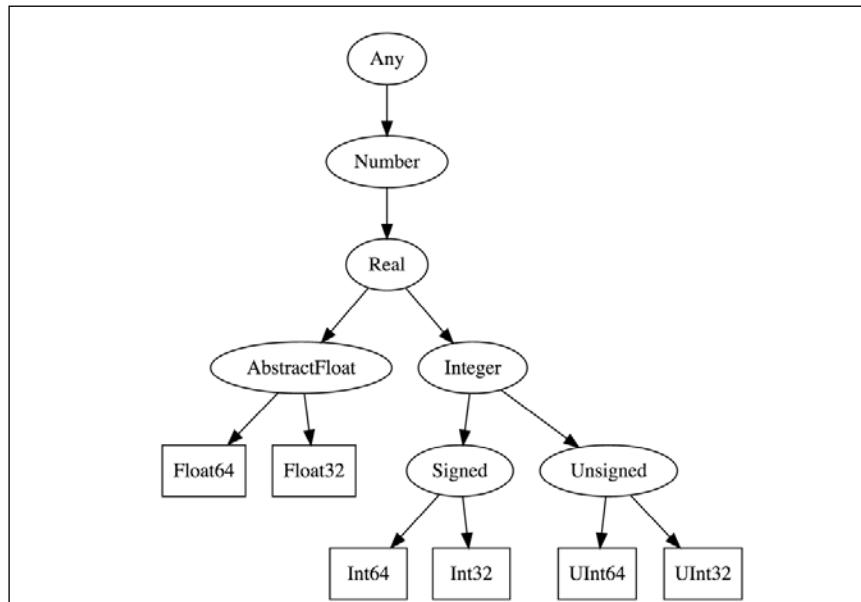
You will notice that the `Number` type is declared without any explicit super type. Hence, as discussed in the next section, it is the direct subtype of `Any`.

Concrete types, on the other hand, are the types that can be instantiated to values. Thus, every value in Julia is of one concrete type. One of the most important points to note about concrete types is that they cannot have any subtypes. Only abstract types can be subtyped. In the language of type theory, all concrete types are declared final in Julia.

Julia's type hierarchy

All types in Julia live within a type hierarchy. This hierarchy is rooted at the top by the `Any` type. All Julia types without exception live within this hierarchy. In particular, unlike languages such as Java, there is no distinction between so-called primitive types and reference types. While there may be differences in how the numbers are represented internally compared to user-defined types, as far as the type system is concerned they form a unified hierarchy.

When a type declaration is omitted for a variable or parameter (as in many of the examples in the previous chapter), it can contain values of any type. This is denoted by the special `Any` type. The `Any` type can therefore be seen as being at the top of Julia's type hierarchy. All other Julia types are subtypes of this type. Visualizing the type hierarchy of some of the numeric types described in the previous chapter is instructive, as follows:



At the other end of the spectrum resides the `None` type. This type lives at the bottom of the type hierarchy. All types are super types of `None`, and there can be no actual instances of this type.

Another special type is the `Void` type. This type has a single instance defined named `nothing`. This is typically used to denote the absence of a value. For example, methods that don't return any other value (for instance, a return type of `void` in some languages), return `nothing`.

Composite and immutable types

Composite types in Julia are collections of named fields. They are equivalent to a `struct` in C and can be thought of as roughly equivalent to a class without behavior in object-oriented languages. They are defined with the `type` keyword and contain the names and types of the fields within them. Take a look at the following code:

```
type Pixel
    x::Int64
    y::Int64
    color::Int64
end

julia> p = Pixel(5,5, 100)
Pixel(5,5,100)

julia> p.x = 10;

julia> p.x
10
```

By default, the fields of a composite type can be changed at any time. In cases where this is undesirable, an immutable type can be declared using the `immutable` keyword. In this case, field values can be set only while constructing an instance of the type. Once created, field values cannot change. Take a look at the following code:

```
immutable IPixel
    x::Int64
    y::Int64
    color::Int64
end
```

```
julia> p = IPixel(5,5, 100)
IPixel(5,5,100)

julia> p.x=10
ERROR: type IPixel is immutable
```

Type parameters

Type parameters are one of the most useful and powerful features of Julia's type system. This is the ability to use parameters when defining types (or functions), thereby defining a whole set of types, one for each value of the parameter. This is analogous to generic or template programming in other languages.

Type parameters are declared within curly braces. For the preceding `Pixel` type, if we wanted to store `color` as an integer, a hexadecimal string, or as an RGB type, we could write it as follows. In this case, `Pixel` itself becomes an abstract type, and `Pixel{Int64}` or `Pixel{ASCIIString}` are the concrete types:

```
type Pixel{T}
    x::Int64
    y::Int64
    color)::T
end
```

Parameters of a type are usually other types. This will be familiar if you have used template classes in C++ or Java generics. In Julia, however, type parameters are not restricted to be other types. They can be values though they are restricted to a set of constant, immutable types. Hence, you can use, among others, integers or symbols as type parameters.

The built-in `Array{T,N}` type is a good example of this usage. This type is parameterized by two parameters, one of which is a type and the other a value. The first parameter, `T`, is the type of the elements of the array. The second, `N`, is an integer specifying the number of dimensions of the array.

The addition of type parameters provides more information to the compiler about the composition of memory. For example, it allows the programmer to assert (or the compiler to infer) the types of elements stored within a container. This, as we'll discuss in the next section, allows the compiler to generate code in turn that is optimized to the types and storage in question.

Type inference

Types in Julia are optional and unobtrusive. The type system usually does not impede for the programmer. It is not necessary or recommended to annotate all variables with type information.

This is not to say that type information is redundant. Quite the opposite is true, in fact. A large part of Julia's speed comes from the ability of the compiler to compile and cache specialized versions of each function for all the possible types to which it can be applied. This means that most functions can be compiled down to their best possible optimized representations.

To achieve this balance, the runtime tries to figure out as much type information as it can through type inference. The algorithm is based on forward dataflow analysis. It should be noted that this is not an implementation of the famous Hindley-Milner algorithm using unification, which is used in the ML family of languages. In these languages, it is mandatory for the compiler to be able to determine the types of every value in the system. For Julia, however, the type inference can be performed on a best-effort basis, with any failure handled with a runtime fallback.

As a simple example of visible type inference, consider the following line of code that creates an array from a range of integers. This code does not have any type annotations. Yet the runtime is able to create an array with properly typed elements of `Int64`:

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

In this section, we provided a quick overview of some important type features in Julia. For more information, visit the online documentation at <http://docs.julialang.org/en/release-0.4/manual/types/>.

For the rest of this chapter, we will assume familiarity with these concepts and look at how this impacts the performance of Julia code

Type-stability

In order for the Julia compiler to compile a specialized version of functions for each different type of its argument, it needs to infer, as best as possible, the parameter and return types of all functions. Without this, Julia's speed would be hugely compromised. In order to do this effectively, the code must be written in a way that it is *type-stable*.

Definitions

Type-stability is the idea that the type of the return value of a function is dependent only on the types of its arguments and not the values. When this is true, the compiler can infer the return type of a function by knowing the types of its inputs. This ensures that type inference can continue across chains of function invocations without actually running the code, even though the language is fully dynamic.

As an example, let's look at the following code, which returns the input for positive numbers but 0 for negative numbers:

```
function trunc(x)
    if x < 0
        return 0
    else
        return x
    end
end
```

This code works for both integers and floating-point output, as follows:

```
julia> trunc(-1)
0

julia> trunc(-2.5)
0

julia> trunc(2.5)
2.5
```

However, you may notice an issue with calling this function with the float input. Take a look at the following:

```
julia> typeof(trunc(2.5))
Float64
```

```
julia> typeof(trunc(-2.5))  
Int64
```

The return type of the `trunc` function, in this case, depends on the value of the input and not just its type. The type of the argument for both the preceding invocations is `Float64`. However, if the value of the input is less than zero, the type of the return is `Int64`. On the other hand, if the input is zero or greater, then the type of the output is `Float64`. This makes the function type-unstable.

Fixing type-instability

Now that we can recognize type-unstable code, the question arises: how can we fix code such as this? There are two obvious solutions. One would be to separate the write versions of the `trunc` function for different input types. So, we could have a version of `trunc` for integers and another for floating point. However, this would cause instances of repeated, copy-pasted code. Also, there would not be just two such instances; there would be copies for `Float32`, `Float64`, `Int32`, `Int64`, and so on. Further, we would have to write a new version of this function for all the new numeric types that are defined. It should be obvious that writing generic functions that operate on a wide variety of related types is really the best way to get concise and elegant Julia code.

The second obvious solution is to branch on the input type within the generic function. So, we could write code similar to this:

```
if typeof(x) == Float64  
    return 0.0  
elseif typeof(x) == Float32  
    return Float32(0.0)  
elseif typeof(x) == Int64  
    return 0  
.....  
end
```

I hope you can see that this can quickly get tedious. However, this type of code provides us with a hint to the correct solution. In Julia, whenever you find yourself explicitly checking the type of any variable, it is time to let dispatch do the job.

The Julia base library contains a `zero(x)` function that takes as its argument any numeric value and returns an appropriately typed zero value for this type. Using this function, we can write a generic `trunc` function that is type-stable yet works for any input type, as follows:

```
function trunc_fixed(x)  
    if x < 0
```

```
        return zero(x)
    else
        return x
    end
end
```

Output of the code:

```
julia> trunc_fixed(-2.4)
0.0

julia> trunc_fixed(-2)
0

julia> typeof(trunc_fixed(-2.4))
Float64

julia> typeof(trunc_fixed(-2))
Int64
```

In making the `trunc` function type-stable, we used a standard library function to move the type variable part of the code into another function. The principle applies when you do not have a base function to fall back upon. Isolate the part of your function that varies depending on the type of the input and allow Julia's dispatch to call the correct piece of code, depending on the type.

Performance pitfalls

We said that type-stability is very important for high-performance Julia code. The speed of Julia programs arises from its ability to compile and cache specialized code for each function argument type. When a function is type-unstable, the Julia compiler cannot compile a fast, specialized version of its caller. Let's take a look at this in action with the preceding code:

```
julia> @benchmark trunc(2.5)
=====
Benchmark Results
=====
Time per evaluation: 13.38 ns [13.04 ns, 13.73 ns]
Proportion of time in GC: 2.39% [1.76%, 3.01%]
Memory allocated: 16.00 bytes
Number of allocations: 1 allocations
Number of samples: 13501
```

```
Number of evaluations: 774542001  
R2 of OLS model: 0.802  
Time spent benchmarking: 10.50 s
```

```
julia> @benchmark trunc_fixed(2.5)  
===== Benchmark Results =====  
Time per evaluation: 5.90 ns [5.86 ns, 5.94 ns]  
Proportion of time in GC: 0.00% [0.00%, 0.00%]  
Memory allocated: 0.00 bytes  
Number of allocations: 0 allocations  
Number of samples: 10601  
Number of evaluations: 48829501  
R2 of OLS model: 0.985  
Time spent benchmarking: 0.51 s
```

Note that the type-stable version is twice as fast as the type-unstable version. Crucially, the type-stable version does not allocate any memory, while the type-unstable version does allocate quite a lot of memory. This combination of slow execution and large memory access is something that you will want to get rid of from your code at all times. Thankfully, it is not that hard to identify type-unstable tools. With the tools available within the language, you will be able to build up an intuition about this very quickly.

Identifying type-stability

In the preceding `trunc` function, the type instability was found by reading and understanding the code. In many cases where the code is longer or more complicated, it may not be easy or even possible to understand the type behavior of a function merely by inspection. It would be useful to have some tools at our disposal.

Fortunately, Julia provides the `@code_warntype` macro that enables us to view the types inferred by the compiler, thereby identifying any type instability in our code. The output of `@code_warntype` is the lowered, type-inferred AST structure. In other words, the compiler parses and processes the source code into a standardized form and then runs the type inference on the result to figure out the possible types of all the variables and function calls within the code.

Let's run this on our type-unstable method and take a look at what it says, as follows:

```
julia> @code_warntype trunc(2.5)
Variables:
    x::Float64
    ##fy#7786::Float64

Body:
begin # none, line 2:
    ##fy#7786 =
    (Base.box)(Float64,(Base.sitofp)(Float64,0))::Float64
    unless
        (Base.box)(Base.Bool,(Base.or_int)((Base.lt_float)(x::Float64,##fy#7786::Float64)::Bool,(Base.box)(Base.Bool,(Base.and_int)((Base.box)(Base.Bool,(Base.and_int)((Base.eq_float)(x::Float64,##fy#7786::Float64)::Bool,(Base.lt_float)(##fy#7786::Float64,9.223372036854776e18)::Bool)::Any)::Bool,(Base.slt_int)((Base.box)(Int64,(Base.fptosi)(Int64,##fy#7786::Float64))::Int64,0)::Bool)::Any)::Bool))::Bool goto 0 # none,
line 3:
    return 0
    goto 1
0: # none, line 5:
    return x::Float64
1:
end::UNION{FLOAT64, INT64}
```

While this output might look slightly scary at first, the relevant portions are easy to highlight. If you run this on Julia REPL, you will see that, in the last line of the output, "Union{Float64, Int64}", is highlighted in red (this is represented by capital letters in the preceding output). This line shows that the compiler inferred that the return type of this function, when passed `Float64` as an argument, can either be `Float64` or `Int64`. Therefore, this function is type-unstable, and this is made obvious by the red highlighting in REPL.

In general, the output from `@code_warntype`, as the name suggests, will warn us of any type inference problem in the code, highlighting it in red. These will usually be variables for which the compiler cannot infer any bound, those typed as `ANY`, or where there are multiple options for possible types denoted as `Union`. While there are some cases where these warnings might be false positives, they should always be investigated if they are unexpected.

If we run this macro on the `trunc_fixed` function, which we made type-stable, we will note that the compiler can infer `Float64` as the return type of the function. Upon running this in REPL, there is no red font in the output, giving us confidence that the function is type-stable. Take a look at the following:

```
julia> @code_warntype trunc_fixed(-2.4)
Variables:
    x::Float64
    ##fy#8904::Float64

Body:
begin # none, line 2:
    ##fy#8904 = (Base.box)(Float64, (Base.sitofp)
    (Float64, 0)::Any)::Float64
    unless
        (Base.box)(Base.Bool, (Base.or_int)((Base.lt_float)(x::Float64, ##fy#89
        04::Float64)::Bool, (Base.box)(Base.Bool, (Base.and_int)((Base.box)(Bas
        e.Bool, (Base.and_int)((Base.eq_float)(x::Float64, ##fy#8904::Float64):
        :Bool, (Base.lt_float)(##fy#8904::Float64, 9.223372036854776e18)::Bool)
        ::Any)::Bool, (Base.slt_int)((Base.box)(Int64, (Base.fptosi)(Int64, ##fy
        #8904::Float64)::Any)::Int64, 0)::Bool)::Any)::Bool goto
    0 # none, line 3:
        return (Base.box)(Float64, (Base.sitofp)(Float64, 0)::Any)::Float64
    goto 1
    0: # none, line 5:
        return x::Float64
    1:
end::Float64
```

Further evidence of the benefits of type-stability can be observed by looking at the LLVM bitcode produced by the Julia compiler. This can be seen using the `@code_llvm` macro, which outputs the result of compiling Julia code into LLVM bitcode. While the details of the output are not relevant, it should be obvious that the type-stable function compiles a much smaller amount of code. It comprises fewer instructions and thus is significantly faster. Take a look at the following code:

```
julia> @code_llvm trunc(2.5)

define %jl_value_t* @julia_trunc_23088(double) {
top:
    %1 = fcmp uge double %0, 0.000000e+00
```

```
br i1 %1, label %L, label %if

if:                                ; preds = %top
    ret %jl_value_t* inttoptr (i64 4356202576 to %jl_value_t*)

L:                                    ; preds = %top
    %2 = call %jl_value_t* @jl_gc_alloc_1w()
    %3 = getelementptr inbounds %jl_value_t* %2, i64 -1, i32 0
    store %jl_value_t* inttoptr (i64 4357097552 to %jl_value_t*),
        %jl_value_t** %3, align 8
    %4 = bitcast %jl_value_t* %2 to double*
    store double %0, double* %4, align 16
    ret %jl_value_t* %2
}

julia> @code_llvm trunc_fixed(2.5)

define double @julia_trunc_fixed_23089(double) {
top:
    %1 = fcmp uge double %0, 0.000000e+00
    br i1 %1, label %L, label %if

if:                                ; preds = %top
    ret double 0.000000e+00

L:                                    ; preds = %top
    ret double %0
}
```

If you are more comfortable with assembly instructions than with LLVM bitcode, the same inference can be gleaned from looking at the final assembly instructions that the Julia code compiles to. This can be output using the `@code_native` macro and is the final code that gets run on the computer's processor. This output is the result of the full gamut of compiler optimizations implemented by the Julia compiler as well as LLVM's JIT. Looking at the output for our usual functions, we can see once again that the type-stable function does significantly less work, as follows:

```
julia> @code_native trunc(2.5)
    .section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 5
    pushq    %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    vmovsd  %xmm0, -8(%rbp)
    vxorpd  %xmm1, %xmm1, %xmm1
    vucomisd %xmm0, %xmm1
    ja     L67
Source line: 5
    movabsq  $jl_gc_alloc_1w, %rax
    callq   *%rax
    movabsq  $4357097552, %rcx          ## imm = 0x103B40850
    movq    %rcx, -8(%rax)
    vmovsd  -8(%rbp), %xmm0
    vmovsd  %xmm0, (%rax)
    jmpq   L77
L67:   movabsq  $4356202576, %rax      ## imm = 0x103A66050
Source line: 3
L77:   addq    $16, %rsp
    popq    %rbp
    ret

julia> @code_native trunc_fixed(2.5)
    .section    __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 5
    pushq    %rbp
    movq    %rsp, %rbp
    vxorpd  %xmm1, %xmm1, %xmm1
    vucomisd %xmm0, %xmm1
    jbe    L22
    vxorpd  %xmm0, %xmm0, %xmm0
Source line: 5
L22:   popq    %rbp
    ret
```

Loop variables

Another facet of type-stability that is important in Julia is that variables within a loop should not change their type from one iteration of the loop to another. Let's first look at a case where this is not true, as follows:

```
function sumsqrtn(n)
    r = 0
    for i = 1:n
        r = r + sqrt(i)
    end
    return r
end
```

In this function, the `r` variable starts out as `Int64`, when the loop is entered in the first iteration. However the `sqrt` function returns `Float64`, which when added to `Int64`, returns `Float64`. At this point, at Line 4 of the function, `r` becomes `Float64`. This violates the rule of not changing the type of a variable within a loop and makes this code type-unstable.

Inspecting the `@code_warntype` output for this function makes this obvious. Viewing this in REPL, we're faced with a swathe of red, which again is highlighted in capital letters here:

```
julia> @code_warntype sumsqrtn(5)
Variables:
  n::Int64
  r::ANY
  #s52::Int64
  i::Int64

Body:
begin  # none, line 2:
    r = 0 # none, line 3:
    GenSym(0) = $(Expr(:new, UnitRange{Int64}, 1,
:((top(getfield))(Base.Intrinsics,:select_value)::I)((Base.sle_int)(1,n::Int64)::Bool,n::Int64,(Base.box)(Int64,(Base.sub_int)(1,1))::Int64)::Int64)))
    #s52 = (top(getfield))(GenSym(0),:start)::Int64
    unless (Base.box)(Base.Bool,(Base.not_int)(#s52::Int64 ===
(Base.box)(Base.Int,(Base.add_int)((top(getfield))(GenSym(0),:stop)::Int64,1))::Int64::Bool))::Bool goto 1
    2:
```

```

GenSym(2) = #s52::Int64
GenSym(3) =
(Base.box)(Base.Int,(Base.add_int)(#s52::Int64,1))::Int64
i = GenSym(2)
#s52 = GenSym(3) # none, line 4:
r = r::Union{Float64,Int64} +
(Base.Math.box)(Base.Math.Float64,(Base.Math.sqrt_llvm)((Base.box)(Fl
oat64,(Base.sitofp)(Float64,i)::Int64))::Float64::Float64
3:
unless
(Base.box)(Base.Bool,(Base.not_int)((Base.box)(Base.Bool,(Base.not_in
t)(#s52::Int64 ===
(Base.box)(Base.Int,(Base.add_int)((top(getfield))(GenSym(0),:stop)::
Int64,1))::Int64::Bool))::Bool)goto 2
1:
0: # none, line 6:
return r::UNION{FLOAT64,INT64}
end::UNION{FLOAT64,INT64}

```

This output shows that the compiler cannot infer a tight bound for the value of `r` (it is typed as ANY), and the function itself can return either `Float64` or `Int64` (for example, it is typed as `Union{Float64, Int64}`)

Fixing the instability is easy in this case. We just need to initialize the `r` variable to be the `Float64` value as we know that that is the type it will eventually take. Take a look at the following function now:

```

function sumsqrt_fixed(n)
    r = 0.0
    for i = 1:n
        r = r + sqrt(i)
    end
    return r
end

```

The `@code_warntype` output for this function is now clean, as follows:

```

julia> @code_warntype sumsqrt_fixed(5)
Variables:
n::Int64
r::Float64
#s52::Int64

```

```
i::Int64

Body:
begin # none, line 2:
    r = 0.0 # none, line 3:
    GenSym(0) = $(Expr(:new, UnitRange{Int64}, 1,
:((top(getfield))(Base.Intrinsics,:select_value)::I)((Base.sle_int)(
1,n::Int64)::Bool,n::Int64,(Base.box)(Int64,(Base.sub_int)(1,1))::Int
64)::Int64)))
    #s52 = (top(getfield))(GenSym(0),:start)::Int64
    unless (Base.box)(Base.Bool,(Base.not_int)(#s52::Int64 ===
(Base.box)(Base.Int,(Base.add_int)((top(getfield))(GenSym(0),:stop)::
Int64,1))::Int64::Bool))::Bool goto 1
    2:
        GenSym(2) = #s52::Int64
        GenSym(3) =
(Base.box)(Base.Int,(Base.add_int)(#s52::Int64,1))::Int64
        i = GenSym(2)
        #s52 = GenSym(3) # none, line 4:
        r =
(Base.box)(Base.Float64,(Base.add_float)(r::Float64,(Base.Math.box)(B
ase.Math.Float64,(Base.Math.sqrt_llvm)((Base.box)(Float64,(Base.sitof
p)(Float64,i::Int64))::Float64))::Float64))::Float64
    3:
        unless
(Base.box)(Base.Bool,(Base.not_int)((Base.box)(Base.Bool,(Base.not_in
t)(#s52::Int64 ===
(Base.box)(Base.Int,(Base.add_int)((top(getfield))(GenSym(0),:stop)::
Int64,1))::Int64::Bool))::Bool goto 2
    1:
        0: # none, line 6:
        return r::Float64
    end::Float64
```

To show why all of this is important, let's time both of these functions, as follows:

```
julia> @benchmark sumsqrt(1000_000)
=====
Time per evaluation: 36.26 ms [34.02 ms, 38.49 ms]
Proportion of time in GC: 18.81% [15.57%, 22.05%]
Memory allocated: 30.52 mb
```

```
Number of allocations: 2000000 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 3.80 s

julia> @benchmark sumsqrtn_fixed(1000_000)
=====
Time per evaluation: 9.52 ms [9.05 ms, 9.99 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.98 s
```

Here, we can see that the type-stable version is four times as fast. More importantly, the type-unstable version of the function allocates a large amount of memory, which is unnecessary. Using type-unstable code, therefore, is extremely prejudicial to high-performance code.

Kernel methods

Type inference in Julia primarily works by inspecting the types of function parameters and identifying the type of the return value. This suggests that some type instability issues may be mitigated by breaking up a function into smaller functions. This can provide additional hints to the compiler, making more accurate type inferencing possible.

For an example of this, consider a contrived function that takes as input the "Int64" or "Float64" string and returns an array of 10 elements, the types of which correspond to the type name passed as the input argument. Functions such as this may arise when creating arrays based on user input or by reading a file in which the type of the output is determined at runtime. Take a look at the following:

```
function string_zeros(s::AbstractString)
    x = Array(s=="Int64"?Int64:Float64, 1_000_000)
    for i in 1:length(x)
        x[i] = 0
    end
    return x
end
```

We will benchmark this code to find an average execution time of over 38 milliseconds per function call with a large memory allocation, as shown by the following code:

```
julia> @benchmark string_zeros("Int64")
=====
Time per evaluation: 38.05 ms [36.80 ms, 39.30 ms]
Proportion of time in GC: 6.45% [6.07%, 6.83%]
Memory allocated: 22.88 mb
Number of allocations: 999492 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 4.19 s
```

This seems to be unnecessarily high. The loop in the function is the obvious place where most of the time is spent within this function. We note that, in this loop, the type of the variable being accessed (`x`) cannot be known before the function is called, even when the type of the function arguments is known. This prevents the compiler from generating an optimized loop operating on one specific type.

What we need to do is ensure that the loop operates in such a way that the type of the `x` variable is known to the compiler. As we said earlier, type inference operates on function boundaries, which suggests a solution to our conundrum. We can split out the loop into its own function, separating the determination of the type of `x` and the operations on `x` across a function call, as follows:

```
function string_zeros_stable(s::AbstractString)
    x = Array(s=="Int64"?Int64:Float64, 1_000_000)
    return fill_zeros(x)
end

function fill_zeros(x)
    for i in 1:length(x)
        x[i] = 0
    end
    return x
end
```

Now, by benchmarking this solution, we will find that the execution time of our function reduces by a factor of 10, with a corresponding fall in the allocated memory. Therefore, in situations where the types of variables are uncertain, we need to be careful in ensuring that the compiler can be provided with as much information as necessary.

Types in storage locations

We discussed in the earlier sections that, when writing idiomatic Julia code, we should try and write functions with the minimum amount of type constraints possible in order to write generic code. We do not need to specify the types of function arguments or local variables for performance reasons. The compiler will be able to infer the required types. Thus, while the types are important, they are usually optional when writing Julia code. In general, bindings do not need to be typed; they are inferred.

However, when defining storage locations for data, it is important to specify a concrete type. So, for things that hold data, such as arrays, dictionaries, or fields in composite types, it is best to explicitly define the type that it will hold.

Arrays

As an example, let's create two arrays containing the same data—the numbers one to ten, which are of the `Int64` type. The first array we will create is defined to hold values of the `Int64` type. The second is defined to hold values of the abstract `Number` type, which is a supertype of `Int64`. Take a look at the following code:

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

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

```
5
6
7
8
9
10
```

We will then pass these arrays into the following function that calculates the sum of squares of the elements of these arrays, as follows:

```
function arr_sumsqr{T <: Number}(x::Array{T})
    r = zero(T)
    for i = 1:length(x)
        r = r + x[i] ^ 2
    end
    return r
end
```

By timing the invocations, we will see that, when using the `Int64` array, this computation is over ten times faster than when using the `Number` array, even when the data within the arrays is identical, as follows:

```
julia> @benchmark arr_sumsqr(a)
=====
Time per evaluation: 34.52 ns [34.06 ns, 34.99 ns]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 9301
Number of evaluations: 14145701
R2 of OLS model: 0.955
Time spent benchmarking: 0.54 s

julia> @benchmark arr_sumsqr(b)
=====
Time per evaluation: 463.24 ns [455.46 ns, 471.02 ns]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 6601
```

```
Number of evaluations: 1080001
R2 of OLS model: 0.951
Time spent benchmarking: 0.57 s
```

The reason for this massive difference lies in how the values are stored within the array. When the array is defined to contain a specific concrete type, the Julia runtime can store the values inline within the allocation of the array since it knows the exact size of each element. When the array can contain an abstract type, the actual value can be of any size. Thus, when the Julia runtime creates the array, it only stores the pointers to the actual values within the array. The values are stored elsewhere on the heap. This not only causes an extra memory load when reading the values, the indirection can mess up pipelining and cache affinity when executing this code on the CPU.

Composite types

There is another situation where concrete types must be specified for good performance: in the fields of composite types.

As an example, consider a composite type holding the location of a point in 2D space. In this scenario, we could define the object as follows:

```
immutable Point
    x
    y
end
```

However, this definition would perform quite badly. The primary issue is that the `x` and `y` fields in this type can be used to store values of any type. In particular, they could be other complex types that are accessed as pointers. In this case, the compiler will not know whether access to the fields of the `Point` type requires a pointer indirection, and thus it cannot optimize the reading of these values.

It will be much better to define this type with the field values constrained to concrete types. This will have two benefits. Firstly, the field values will be stored inline when the object is allocated rather than being directed via pointer. Secondly, all code that uses fields of this type will be able to be type-inferred correctly, as follows:

```
immutable ConcretePoint
    x::Float64
    y::Float64
end
```

Parametric composite types

While the preceding definition of `ConcretePoint` performs well, it loses some significant flexibility. If we wanted to store the field values as `Float32` or `Float16`, we would be unable to use the same type. To lose so much flexibility for performance seems very unfortunate.

It would be tempting to fix this using an abstract type as the fields. In this case, all the concrete floating point numbers are subtypes of the `AbstractFloat` type. Here, we could then define a `PointsWithAbstract` type that contains fields annotated as `AbstractFloat`, as follows:

```
immutable PointWithAbstract
    x::AbstractFloat
    y::AbstractFloat
end
```

However, this code has the same drawbacks as the original `Point` type mentioned earlier. It will be slow, and the compiler will be unable to optimize access to the type. The solution is to use a parametric type, as follows:

```
immutable ParametricPoint{T <: AbstractFloat}
    x::T
    y::T
end
```

When we write the type in this manner, our code remains generic. We can write our methods with the confidence that the `ParametricPoint` type can hold values for any type of a floating point number. Yet, at runtime, when an instance of this type is created, it is instantiated with a particular type of float. In other words, once an instance is created, `T` becomes known. At this point, all the benefits of specifying the concrete type discussed before are applicable. Both storage and type inferences are efficient now.

Summary

In this chapter, we discussed how types play a crucial role in writing idiomatic and performant code in Julia. Much of what we discussed here is exactly what makes Julia unique—a dynamic language where types, dispatch, and inference play a fundamental role.

We discussed how to write type-stable code and when and how to define type annotations for performance. In the next chapter, we will discuss the performance characteristics of another important part of the language: functions.

4

Functions and Macros – Structuring Julia Code for High Performance

In Julia, the function is the primary unit of a code structure. Idiomatic Julia code consists of many small functions that are defined with different types of arguments. In general, the overhead of a function call in Julia is very small, and, with type specialization, the compiled version of the function is very efficient. In this chapter, we will look at some of the techniques that Julia uses to make function calls very fast. We will also look at some limitations that are worth keeping in mind for the fastest code. Finally, we will look at some situations where moving code out of functions and into other structures, such as macros and staged functions, allows code to be faster and more efficient:

- Using globals
- Inlining
- Closures and anonymous functions
- Using macros for performance
- Using generated functions
- Using named parameters

Using globals

One of the first performance tips that you come across when learning Julia is the advice not to use global variables. This is usually not a very onerous requirement, as global state is often considered bad programming practice. Further, this limitation is most likely going to be removed in future versions of Julia. However, given how easy it is to fall into this trap and the large amount of performance degradation that can occur, it is important to keep this in mind when writing Julia code.

The trouble with globals

In the previous chapter, we saw how Julia achieves its high performance runtime by compiling specialized versions of functions for particular types of arguments—a process that relies on type inference using data flow techniques. However, global variables can be written to at any time, and by any code. The compiler cannot keep track of all writes to global variables; this would be akin to solving the halting problem. Therefore, the data-flow technique fails to perform any inference for these types of global variables. As a result, the compiler cannot create specialized functions when using these variables.

To understand the performance hit of using global variables, let's use a simple function that calculates the sum of the integer powers of a set of floating point values. We use a global variable to store the integer power:

```
p = 2

function pow_array(x::Vector{Float64})
    s = 0.0
    for y in x
        s = s + y^p
    end
    return s
end
```

Benchmarking this function, we see that it takes approximately 10 milliseconds for each evaluation of this function for an input array of length 100000. This is way too high for something that should only take a few machine instructions to execute:

```
julia> t=rand(100000);  
  
julia> @benchmark pow_array(t)  
===== Benchmark Results =====  
Time per evaluation: 9.39 ms [8.48 ms, 10.30 ms]  
Proportion of time in GC: 4.58% [0.00%, 10.14%]  
Memory allocated: 4.58 mb  
Number of allocations: 300000 allocations  
Number of samples: 100  
Number of evaluations: 100  
Time spent benchmarking: 0.97 s
```

A look at the `@code_warntype` output for this function shows us that the compiler has been unable to infer the type of the result when calculating with the global variable, marking it as ANY. This then flows through the entire function, right up to the return value (as usual, any untyped variables, displayed in red in the REPL, are shown in capital letters, as follows):

```
julia> @code_warntype pow_array(t)  
  
Variables:  
x::Array{Float64,1}  
s::ANY  
#s641::Int64  
y::Float64  
  
Body:  
begin # none, line 2:  
    s = 0.0 # none, line 3:  
    #s641 = 1  
    GenSym(2) = (Base.arraylen)(x::Array{Float64,1})::Int64  
    unless  
(Base.box)(Base.Bool,(Base.not_int)((Base.slt_int)(GenSym(2),#s641  
::Int64)::Bool))::Bool goto 1  
    2:  
    GenSym(4) = (Base.arrayref)(x::Array{Float64,1},#s641::Int64)::Flo  
at64  
    GenSym(5) = (Base.box)(Base.Int,(Base.add_int)  
(#s641::Int64,1)::Any)::Int64  
    y = GenSym(4)  
    #s641 = GenSym(5) # none, line 4:  
    s = s + y::Float64 ^ Main.p::ANY::ANY  
    3:
```

```
GenSym(3) = (Base.arraylen)(x::Array{Float64,1})::Int64
unless
(Base.box)(Base.Bool,(Base.not_int)((Base.box)(Base.Bool,(Base.not
_int)((Base.slt_int)(GenSym(3),#s641::Int64)::Bool))::Bool))::Bool
goto 2
1:
0: # none, line 6:
return s
end:: ANY
```

Fixing performance issues with globals

A simple way to get back performance is to declare the global variable `a` `const`:

```
const p2 = 2
function pow_array2(x::Vector{Float64})
    s = 0.0
    for y in x
        s = s + y^p2
    end
    return s
end
```

Just this change will get us a little under two orders of magnitude performance gain on the following function:

```
julia> @benchmark pow_array2(t)
=====
Benchmark Results
=====
Time per evaluation: 123.90 μs [120.54 μs, 127.27 μs]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 3901
Number of evaluations: 82201
R² of OLS model: 0.926
Time spent benchmarking: 10.44 s
```



Global const

The `const` declaration in Julia means something different from the similar keyword in C. In Julia, a global variable declared as `const` can change its value (a warning is printed). However, what it cannot do is change its type. Also, note that you cannot explicitly declare the type of a global variable. That is, an incantation, such as `x::Int64 = 2`, will raise an error when made in the global scope.

Once again, `@code_warntype` will show us that this function is now correctly type inferred all the way through. Compare this output against the one from the previous function in the preceding section. You will notice that the return value of this function is being inferred as `Float64`:

```
julia> @code_warntype pow_array2(t)

Variables:
  x::Array{Float64,1}
  s::Float64
  #s614::Int64
  y::Float64

Body:
begin  # none, line 2:
  s = 0.0 # none, line 3:
  #s614 = 1
  GenSym(2) = (Base.arraylen)(x)::Array{Float64,1}::Int64
  unless (Base.box)(Base.Bool,(Base.not_int)((Base.slt_int)(GenSym(2),
  #s614::Int64)::Bool))::Bool goto 1
  2:
  GenSym(4) = (Base.arrayref)(x::Array{Float64,1},#s614::Int64)::Flo
at64
  GenSym(5) = (Base.box)(Base.Int,(Base.add_int)
  (#s614::Int64,1))::Int64
  y = GenSym(4)
  #s614 = GenSym(5) # none, line 4:
  s = (Base.box)(Base.Float64,(Base.add_float)(s::Float64,(Base.
  Math.box)(Base.Math.Float64,(Base.Math.powi_llvm)(y::Float64,(Base.box)
  (Int32,(Base.checked_trunc_sint)(Int32,Main.p2))::Int32))::Float64))::Flo
at64
```

```
3:
    GenSym(3) = (Base.arraylen)(x::Array{Float64,1})::Int64
    unless (Base.box)(Base.Bool, (Base.not_int)((Base.box)(Base.
Bool, (Base.not_int)((Base.slt_int)(GenSym(3), #s614::Int64)::Bool))::Bool))
) ::Bool goto 2
1:
0: # none, line 6:
    return s::Float64
end::Float64
```

Another way to solve the issue of the global variable is to pass the global as a function argument. A function argument can be type inferred; hence, the function specialization will be effected in this case.

Inlining

As we've mentioned before, Julia code consists of many small functions. Unlike most other language implementations, some of the core primitives in the base library are also implemented in Julia. This means that the function call overhead has the potential to be a bottleneck in a Julia program. This is mitigated using some aggressive inlining performed by the Julia compiler.

Inlining is an optimization performed by a compiler, where the contents of a function or method is inserted directly into the body of the caller of that function. Thus, instead of making a function call, execution continues directly by executing the operations of the callee within the caller's body.

In addition, many compiler optimization techniques work within the body of a single function. Inlining, therefore, allows many more optimizations to be effective within the program.

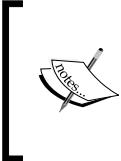
Compiler optimizations



Julia uses the LLVM compiler to generate machine code, which is finally run on the CPU. Most of the usual compiler optimization techniques that run on Julia code are performed by LLVM. The one major exception is inlining, which is performed by the Julia compiler itself before LLVM is invoked.

Default inlining

The Julia compiler automatically inlines functions that it considers inline-worthy. The compiler implements a set of heuristics to determine what to inline. Essentially, this boils down to small functions with deterministic types.



While inlining usually results in an increase in code speed, it also simultaneously increases the size of the code. Hence, a balance needs to be maintained. The heuristics are, therefore, tuned to maximize the performance of typical Julia code without causing excessive bloating of the compiled code.



As an example, let's take a look at a simple set of functions, some of which we've seen in previous chapters:

```
trunc(x) = x < 0 ? zero(x) : x

function sqrt_sin(x)
    y = trunc(x)
    return sin(sqrt(y)+1)
end
```

We can then look at the processed AST after the compiler has run its type inference and inlining passes. Note how in the following output, the code for the `trunc` function has been inserted into the `sqrt_sin` function as the first few lines:

```
julia> @code_typed sqrt_sin(1)
1-element Array{Any,1}:
 :(($Expr(:lambda, Any[:x],
Any[Any[Any[:x, Int64, 0], Any[:y, Int64, 18], Any[:_var0, Int64, 2]], Any[],
Any[Float64, Float64, Float64], Any[], :(begin # none, line 2:
unless (Base.slt_int)(x::Int64, 0)::Bool goto 1
_var0 = 0
goto 2
1:
_var0 = x::Int64
2:
y = _var0::Int64 # none, line 3:
GenSym(0) = (Base.box)(Base.Float64, (Base.add_float)((Base.Math.
box)(Base.Math.
.Float64, (Base.Math.sqrt_llvm)((Base.box)(Float64, (Base.sitofp)(Float64,y
::Int64))::Float64))::Float64, (Base.box)(Float64, (Base.sito
fp)(Float64, 1))::Float64)::Float64
```

```
GenSym(2) =  
(top(ccall)((top(tuple))("sin",Base.Math.libm)::Tuple{ASCIIString,  
ASCIIString},Base.Math.Float64,(top(svec))(Base.Math.Float64)::Si  
mpleVector,GenSym(0),0)::Float64  
return  
(Base.Math.nan_dom_err)(GenSym(2),GenSym(0))::Float64  
end)::Float64)))
```

Controlling inlining

Sometimes, the heuristics to inline that are built into the Julia compiler will fail to inline functions that we want inlined. These would typically be performance-sensitive functions that are called many times in inner loops, for example, array indexers. For this purpose, Julia provides the `@inline` macro. This macro needs to be placed in front of a function definition. When that function is called, its body will be placed inline at the location where it is called.



There is no call-site annotation to force inlining. We cannot inline a particular invocation of an, otherwise, normal function. The function itself should be marked with `@inline`, and then every invocation of that function will be inlined.

Let's demonstrate this with an example. In the following code, we define an `f(x)` function that performs some numerical operations on its arguments, as well as a `g(x)` function that calls `f` after transforming its argument:

```
function f(x)  
    a=x*5  
    b=a+3  
    c=a-4  
    d=b/c  
end
```

This function `f` is too long to be inlined by default, which we verify by inspecting the `@code_typed` output of its `g` calling function. Note that the function definition of `g` continues to contain a call to the `f` function:

```
julia> @code_typed g(3)  
1-element Array{Any,1}:  
:(($(Expr(:lambda, Any[:x],  
Any[Any[Any[:x, Int64, 0]], Any[], Any[], Any[], :begin # none, line 1:
```

```
    return
(Main.f) ((Base.box)(Int64,(Base.mul_int)(2,x::Int64))::Int64)::Fl
oat64
end::Float64))))
```

We then define the same computation in a function that we declare with the `@inline` macro:

```
@inline function f_inline(x)
    a=x*5
    b=a+3
    c=a-4
    d=b/c
end

g_inline(x) = f_inline(2*x)
```

When we inspect the compiled AST for this function, it is apparent that the called function has been inlined into the caller:

```
julia> @code_typed g_inline(3)
1-element Array{Any,1}:
 :($Expr(:lambda, Any[:x], Any[Any[Any[:x, Int64, 0], Any[symbol("##a#6865"
), Int64, 18], Any[symbol("##b#6866"), Int64, 18], Any[symbol("##c#6867"), Int64, 18], Any[symbol("##d
#6868"), Float64, 18]], Any[], Any[Float64], Any[], :(begin # none, line
1:
    ##a#6865 =
(Base.box)(Int64, (Base.mul_int)((Base.box)(Int64, (Base.mul_int)(2,x::
Int64))::Int64, 5))::Int64
    ##b#6866 =
(Base.box)(Base.Int, (Base.add_int)(##a#6865::Int64, 3))::Int64
    ##c#6867 =
(Base.box)(Int64, (Base.sub_int)(##a#6865::Int64, 4))::Int64
    GenSym(0) =
(Base.box)(Base.Float64, (Base.div_float)((Base.box)(Float64, (Base.sit
ofp)(Float64, ##b#6866::Int64))::Float64, (Base.box)(Float64, (Base.sito
fp)(Float64, ##c#6867::Int64))::Float64))::Float64
    ##d#6868 = GenSym(0)
    return GenSym(0)
end)::Float64))))
```

It is even more instructive to see the LLVM bitcode that is generated from this function. We can see this using the `@code_llvm` macro. Note that the first line of the function is now `%1 = mul i64 %0, 10`. This shows the argument of the function being multiplied by 10. Look back at the source of the function – the argument is multiplied by 2 in the `g` function and, subsequently, by 5 in the `f` function. The LLVM optimizer has recognized this and consolidated these two operations into a single multiplication. This optimization has occurred by merging code across two different functions and, thus, couldn't have happened without inlining:

```
julia> @code_llvm g_inline(3)

define double @julia_g_inline_21456(i64) {
top:
    %1 = mul i64 %0, 10
    %2 = add i64 %1, 3
    %3 = add i64 %1, -4
    %4 = sitofp i64 %2 to double
    %5 = sitofp i64 %3 to double
    %6 = fdiv double %4, %5
    ret double %6
}
```

Disabling inlining

We've seen how useful inlining can be for the performance of our programs. However, in some situations, it may be useful to turn off all inlining. These can be during complex debugging sessions or while running code coverage analysis. For example, in any situation where one needs to maintain direct correspondence between source lines of code and executing machine code, inlining can be problematic.

Therefore, Julia provides a `-inline=no` command line option to be used in these circumstances. Using this option will disable all inlining, including the ones marked with `@inline`. We warned you that using this option makes all Julia code significantly slower. However, in rare situations this is exactly what is needed.

Closures and anonymous functions

We saw how important functions are in idiomatic Julia code. While not a pure functional language, Julia shares many features with such languages. In particular, functions in Julia are first class entities, and they can be passed around to other functions to create higher-order functions. A canonical example of such a higher-order function is the `map` function, which evaluates the given function over each element of the provided collection.

As you would expect from a language with these functional features, it is also possible to create closures and anonymous functions in Julia. Anonymous functions, as the name suggests, are functions without a name, and they are usually created at the point where they are passed in to another function as an argument. In Julia, they are created with the `->` operator separating the arguments from the function body. These, and named functions created within the scope of another function, and referring to variables from this outer scope, are called closures. This name arises from the idea of these functions "closing over" the outer scope.

Anonymous functions and closures are much slower than named functions in versions of Julia prior to 0.5. This is due to the fact that the Julia compiler currently cannot type infer the result of anonymous functions. It should be obvious that the lack of type inference will significantly slow these functions down. As always, it is instructive to look at an example and measure its performance. First, we define a `sqr` function, which returns the square of its input argument:

```
sqr(x) = x ^ 2
```

We then measure the performance of `map`, evaluating this function over a random array of 100,000 `Float64` elements. We also measure the performance of `map` when it is passed the same computation as an anonymous function, rather than the named `sqr` function:

```
julia> @benchmark map(sqr, rand(100_000))
=====
Time per evaluation: 3.81 ms [2.98 ms, 4.64 ms]
Proportion of time in GC: 8.88% [0.00%, 20.33%]
Memory allocated: 3.81 mb
Number of allocations: 200003 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.41 s
```

```
julia> @benchmark map(x->x^2, rand(100_000))
=====
Time per evaluation: 7.97 ms [6.97 ms, 8.96 ms]
Proportion of time in GC: 5.38% [0.00%, 12.70%]
Memory allocated: 3.81 mb
Number of allocations: 200003 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.83 s
```

It is apparent that using a named function is about twice as fast as using an anonymous function. It should be noted that while this is true of the current version of Julia at the time of writing (0.4,) this limitation will be removed in future versions of Julia. If you are using Julia v0.5 or later, then you do not need to consider any of the content in this section or the next section. In these versions, anonymous functions are as fast as named functions. However, for the moment, it is advisable to limit uses of closures and anonymous functions as much as possible in performance-sensitive code.

FastAnonymous

However, in many situations, it is necessary or even convenient to use anonymous functions. We have a language with many functional features, and it would be a shame to forgo closures. So, if the slow performance of these constructs are a bottleneck in your code, the innovative Julia community has a workaround in the form of the `FastAnonymous` package.

Using this package is easy and causes very low programmer overhead. After installing and importing, writing an `@anon` macro before an anonymous function declaration will transform it into a form that can be type inferred, and this is, thus, much faster. Running the example from the previous section with this approach yields a significantly faster runtime:

```
julia> using FastAnonymous

julia> @benchmark map(@anon(x->x^2), rand(100_000))
```

```
===== Benchmark Results =====
Time per evaluation: 488.63 μs [298.53 μs, 678.73 μs]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 781.31 kb
Number of allocations: 2 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.29 s
```

Once again, we should note that use of this package will become unnecessary in version 0.5 and further versions of Julia when the performance difference between anonymous and named functions are removed.

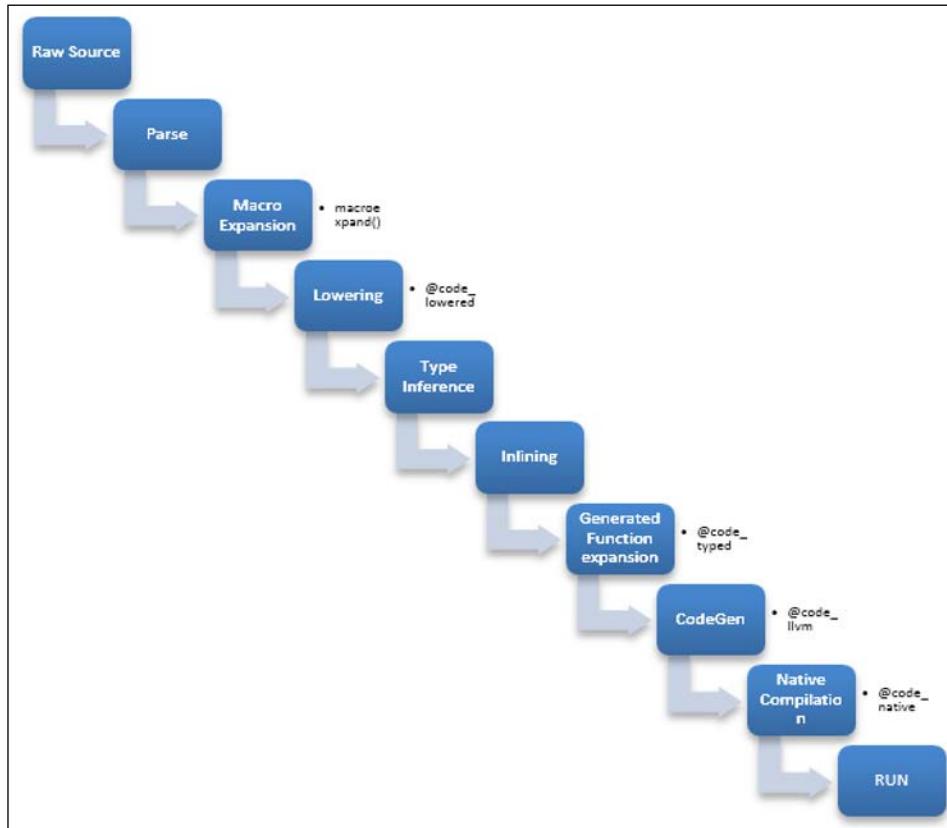
Using macros for performance

So far in this chapter, we have focused on making our functions run faster. However, as fast we make them, all the computation occurs when a function is called. The best way to make any code faster is, however, to do less work. So, a strategy is to move any possible work to compile time, which leaves less work to do at runtime.

The Julia compilation process

However, for a dynamic language such as Julia, the terms compile time and runtime are not always clearly defined. In some sense, everything happens at runtime because our code is not compiled to a binary ahead of time. However, there are clearly divided processes that occur from when the code is read from disk to when it is finally executed on the CPU.

As the compiler goes through each stage, it can write code to execute at various points along this pipeline rather than everything waiting until the end—the runtime. While we might loosely use the terminology of compile time for some of our metaprogramming techniques, having the ability to run code at multiple stages along this pipeline provides some powerful capabilities:



Using macros

Julia macros are code that can be used to write Julia code. A macro is executed very early in the compiler process, as soon as the code is loaded and parsed.

Macros are usually used as a means to reduce repetitive code, whereby large volumes of code with a common pattern can be generated from a smaller set of primitives. However, they can also be used to improve performance in some situations. This usually involves moving common or constant computation to the compile time wherever possible. To see how this can work, let's look at the problem of evaluating a polynomial.

Evaluating a polynomial

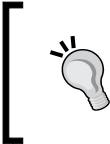
Consider the following polynomial expression:

$$p(x) = \sum_{i=0}^n a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n$$

Given a set of coefficients $[a_0, a_1, a_2, \dots, a_n]$, we need to find the value of the $p(x)$ function for a particular value of x .

A simple and naive but general implementation to evaluate any polynomial may be, as follows:

```
function poly_naive(x, a...)
    p=zeros(x)
    for i = 1:length(a)
        p = p + a[i] * x^(i-1)
    end
    return p
end
```



Type Stability, once again

You will recognize this from the discussions in the previous chapter that the initialization of $p=zeros(x)$ rather than $p=0$ ensures the type stability of this code.

Using this function, let's imagine that we need to compute a particular polynomial:

$$f(x) = 1 + 2x + 3x^2 + 4x^3 + 5x^4$$

```
julia> f_naive(x) = poly_naive(x, 1, 2, 3, 4, 5)
```

```
julia> f_naive(3.5)
```

```
966.5625
```

Let's verify the calculation by hand to test its accuracy and then benchmark the computation to see how fast it can run:

```
julia> 1 + 2*3.5 + 3*3.5^2 + 4*3.5^3 + 5*3.5^4
```

```
966.5625
```

```
julia> @benchmark f_naive(3.5)
=====
Time per evaluation: 162.51 ns [160.31 ns, 164.71 ns]
Proportion of time in GC: 0.18% [0.00%, 0.39%]
Memory allocated: 32.00 bytes
Number of allocations: 2 allocations
Number of samples: 9701
Number of evaluations: 20709801
R² of OLS model: 0.953
Time spent benchmarking: 3.39 s
```

This computation takes a little over 160 nanoseconds. While this is not a particularly long interval, it is quite long for modern CPUs. A 2.4 GHz processor should be able to perform around 10,000 floating point operations in that time, which seems like a lot of work to compute a polynomial with five terms. The primary reason why this is slower than we would expect is that floating-point exponentiation is a particularly expensive operation.



Peak Flops

The peakflops() Julia function will return the maximum number of **floating point operations per second (flops)** possible on the current processor.

Horner's method

So, the first thing to do is to find a better algorithm, one which can replace the exponentiation into multiplications. This can be done by the Horner method, which is named after the nineteenth century British mathematician, William George Horner. This is accomplished by defining a sequence, as follows:

$$\begin{aligned} b_n &= a_n \\ b_{n-1} &= a_{n-1} + b_n x \\ b_{n-2} &= a_{n-2} + b_{n-1} x \\ &\vdots \\ b_0 &= a_0 + b_{n-1} x \end{aligned}$$

Then, b_0 is the value of the $p(x)$ polynomial.

This algorithm can be implemented in Julia, as follows:

```
function poly_horner(x, a...)
    b=zero(x)
    for i = length(a):-1:1
        b = a[i] + b * x
    end
    return b
end
```

We can then test and benchmark this for the same polynomial:

```
f_horner(x) = poly_horner(x, 1,2,3,4,5)
```

```
julia> @benchmark f_horner(3.5)
=====
Time per evaluation: 41.51 ns [40.96 ns, 42.06 ns]
Proportion of time in GC: 1.16% [0.75%, 1.57%]
Memory allocated: 32.00 bytes
Number of allocations: 2 allocations
Number of samples: 12301
Number of evaluations: 246795401
R² of OLS model: 0.943
Time spent benchmarking: 10.36 s
```

We see that using a better algorithm gets us a $4x$ improvement in the evaluation speed of this polynomial. Can we do better?

The Horner macro

Improving the speed of this computation starts with realizing that the coefficients of the polynomial are constants. They do not change and are known when writing the program. In other words, they are known at compile time. So, maybe we can expand and write out the expression for the Horner's rule for our polynomial. This will take the following form, for the polynomial that we used previously:

```
muladd(x,muladd(x,muladd(x,muladd(x,5,4),3),2),1)
```

This is likely to be the fastest way to compute our polynomial. However, writing this out for every polynomial that we might want to use will be extremely annoying. We loose the benefit of having a general library function that can compute any polynomial.

This is exactly the kind of situation where macros can help. We can write a macro that will produce the previous expression when given a set of polynomial coefficients. This can be done when the compiler loads the code. At runtime, when this function is called, it will execute this optimized expression. Julia's base library contains this macro, which we can see repeated, as follows:

```
macro horner(x, p...)
    ex = esc(p[end])
    for i = length(p)-1:-1:1
        ex = :(muladd(t, $ex, $(esc(p[i]))))
    end
    Expr(:block, :(t = $(esc(x))), ex)
end

f_horner_macro(x) = @horner(x, 1, 2, 3, 4, 5)

julia> @benchmark f_horner_macro(3.5)
=====
Benchmark Results
=====
Time per evaluation: 3.66 ns [3.62 ns, 3.69 ns]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 11601
Number of evaluations: 126646601
R² of OLS model: 0.970
Time spent benchmarking: 0.53 s
```

So, this method using a macro gives us an amazing $10x$ improvement over calling the Horner's method as a function. Also, this function does not allocate any memory at runtime.

We've seen how this strategy of generating customized code for particular problems using a macro can sometimes lead to massive performance increases. While the `@horner` macro is a simple and canonical example of this strategy, it can be used to great effect in our own code.

Generated functions

Macros run very early in the compilers process when there is no information about how the program might execute. The inputs to a macro are, therefore, simply symbols and expressions – the textual tokens that make up a program. Given that a lot of Julia's powers come from its type system, it may be useful to have something such as macros – code that generates code – at a point where the compiler has inferred the types of the variables and function arguments in the program. Generated functions (also sometimes called *staged* functions) fulfill this need.

Using generated functions

Declaring a generated function is simple. Instead of the usual `function` keyword, generated functions are declared with the appropriately named `@generated` function keyword. This declares a function that can be called normally from any point in the rest of the program.

Generated functions come in two parts, which are related to how they are executed. They are invoked once for each unique type of its arguments. At this point, the arguments themselves take the values of their types. The return value of this execution must be an expression that is used as the body of the function when called with values of these types. This cycle is executed each time the function is called with new types. The function is called with types as values once, and then the returned expression is used for all invocations with argument values of this type.



More on generated functions

In this section, we quickly described how to write generated functions. We will not go into too much detail. For more information along with examples, please refer to the online Julia documentation

Using generated functions for performance

As with macros, strategies to use generated functions for performance revolve around moving constant parts of the computation earlier into the compilation stage. However, unlike macros, here the computations are fixed only for a certain type of argument. For different types of argument, the computations are different. Staged functions handle this difference elegantly.

As an example, let's consider a rather trivial problem: calculating the number of cells of a multidimensional array. The answer is of course a product of the number of elements in each dimension. As Julia has true multidimensional arrays, the number of dimensions, and the number of multiplications are not known upfront. One possible implementation is to loop over the number of dimensions, multiplying as we go:

```
function prod_dim{T, N}(x::Array{T, N})
    s = 1
    for i = 1:N
        s = s * size(x, i)
    end
    return s
end
```



Type parameters

Please review the Julia documentation on type parameters or refer to *Type parameters* section in *Chapter 3, Types in Julia*, if the preceding code looks unfamiliar.



This function will now work for arrays with any number of dimensions. Let's test this to see whether it works:

```
julia> prod_dim(rand(10,5,5))
250
```

Optimizing this computation with a generated function starts with the observation that the number of iterations of the loop is equal to the number of dimensions of the array, which is encoded as a type parameter for arrays. In other words, for a particular type of input (and array of a particular dimension), the loop size is fixed. So, what we can try to do in a generated function is move the loop to the compile time:

```
@generated function prod_dim_gen_impl{T, N}(x::Array{T, N})
    ex = :(1)
    for i = 1:N
        ex = :(size(x, $i) * $ex)
    end
    return ex
end
```

In this generated function, the loop runs at compile time when the type of *x* is known. We create an *ex* expression, which then becomes the body of the function when actually called with an instance of an array. We can see that this function works; it returns the same result as our earlier version with the loop:

```
julia> prod_dim_gen_impl(rand(10, 5, 5))
250
```

However, it would be instructive to see the code that is generated and actually run for this function. For this purpose, we can paste the body of the generated function into a normal function, as follows:

```
function prod_dim_gen_impl{T, N}(x::Array{T, N})
    ex = :(1)
    for i = 1:N
        ex = :(size(x, $i) * $ex)
    end
    return ex
end
```

We can then call this function with the type of the arguments as input, and the returned expression will show us how this generated function works:

```
julia> x = rand(2, 2, 2);
julia> prod_dim_gen_impl(x)
:(size(x,3) * (size(x,2) * (size(x,1) * 1)))

julia> x = rand(2, 2, 2, 2);
julia> prod_dim_gen_impl(x)
:(size(x,4) * (size(x,3) * (size(x,2) * (size(x,1) * 1))))
```

It should be apparent what has happened here. For an array of three dimensions, we are multiplying three numbers; while for an array of four dimensions, we are multiplying two numbers. The loop of `1:N` ran at compile time and then disappeared. The resulting code will be much faster without the loop, particularly if this function is called excessively in some other inner loop.

The technique of removing loops and replacing them with the calculations inline is usually called *loop-unrolling*, and it is often performed manually in performance-sensitive code. However, in Julia, generated functions are an easy and elegant way to achieve this without too much effort.

Also, note that this function looks much simpler without the loop. The number of tokens in this function is significantly reduced. This might make the function inlineworthy and cause the compiler to inline this function, making this code even faster.

Using named parameters

Julia supports a convenient named parameter syntax that is useful when creating complicated API with many optional parameters. However, the compiler cannot infer the types of named parameters effectively. Therefore, it should now be apparent that using named parameters can cause degraded performance.

As an example, we shall write the same function, once with named arguments, and once with regular, positional arguments. It will be apparent that the version with named arguments does not perform very well. (As an aside, note that the `Benchmarks` package that we've been using does not support named arguments. Therefore, we are benchmarking this code in a very simple way):

```
julia> named_param(x; y=1, z=1) = x^y + x^z
named_param (generic function with 1 method)

julia> pos_param(x,y,z) = x^y + x^z
pos_param (generic function with 1 method)

julia> @time for i in 1:100000;named_param(4; y=2, z=3);end
 0.032424 seconds (100.23 k allocations: 9.167 MB)

julia> @time for i in 1:100000;pos_param(4, 2, 3);end
 0.000832 seconds
```

It is apparent that using named parameters incurs a significant overhead in Julia. However, when designing high-level functions, it is still advantageous to use named parameters in order to create easy to use API's. Just don't use them in performance-sensitive inner loops.

Summary

In this chapter, we saw different ways to structure our code to make it perform better. The function is the primary element in Julia code; however, sometimes it is not the best option. Macros and generated functions can play an important role where appropriate.

In the next chapter, we will look deeper into the problem of numbers. We will see how Julia designs its core number types, and how to make basic numeric operations fly.

5

Fast Numbers

As it is a numerical programming language, fast computations with numbers are central to everything we do in Julia. In the previous chapters, we discussed how the Julia compiler and runtime perform across a wide range of code. In this chapter, we will take a detailed look at how these core constructs are designed and implemented in Julia.

In this chapter, we will cover the following topics:

- Numbers in Julia
- Trading performance for accuracy
- Subnormal numbers

Numbers in Julia

The basic number types in Julia are designed to closely follow the hardware on which it runs. The default numeric types are as close to the metal as possible—a design decision that contributes to Julia's C-like speed.

Integers

Integers in Julia are stored as binary values. Their default size, as in C, depends on the size of the CPU/OS on which Julia runs. On a 32-bit OS, the integers are 32 bits by default, and on a 64-bit machine, they are 64 bits by default. These two integer sizes are represented as different types within Julia: `Int32` and `Int64`, respectively. The `Int` type alias represents the actual integer type used by the system. The `WORD_SIZE` constant contains the bit width of the current Julia environment, which is as follows:

```
julia> WORD_SIZE  
64
```

The `bits` function displays the underlying binary representation of the numbers. On a 64-bit machine, we get:

The default integer types are signed. That is, the first (and the most significant) bit is set to 1 to denote negative numbers, which are then stored as two's complement, as follows:

Types such as these, and the following floating point types whose representation is simply a set of bits, have optimized handling within the Julia runtime. They are called *bits* types, and this feature can be queried for any type using the `isbits` function, as follows:

```
julia> isbits(Int64)  
true
```

```
julia> isbits(ASCIIString)  
false
```

One point to note is that, as a Julia value, basic numeric types can be boxed. That is, when stored in memory they are prefixed with a tag that represents their type. However, the Julia compiler is usually very good at removing any unnecessary boxing/unboxing operations. They can usually be compiled out. For example, we can define a function that adds two numbers and inspect the machine code that is generated and executed when this function is called via the following code:

```
myadd(x, y) = x + y
```

Looking at the output of the following compiled code, (even if, like me, you are not an expert at reading assembly), it should be apparent that, other than the function overhead to set the stack and return the result, the generated code simply consists of the CPU instruction to add two machine integers, `addq`. There is no boxing/unboxing operation remaining in the native code when the function is called. Take a look at the following:

```
julia> @code_native myadd(1,2)
        .section __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 1
    pushq %rbp
    movq %rsp, %rbp
Source line: 1
    addq %rsi, %rdi
    movq %rdi, %rax
    popq %rbp
    ret
```

There is an even bigger advantage to storing numbers using the machine representation. Arrays of these numbers can be stored using contiguous storage. A type tag is stored once at the start. Beyond this, data in numerical arrays is stored in a packed form. This not only means that these arrays can be passed to C libraries as-is (minus the type tag) but also that the compiler can optimize computations on these arrays easily. There is no need for pointer dereferencing when operating on numerical arrays of bit types.

Integer overflow

A further consequence of the decision to use machine integers by default is that there are no overflow checks present within any base mathematical operation in Julia.

With a fixed number of bytes available to represent integers of a certain type, the possible values are bounded. These bounds can be viewed using the `typemax` and `typemin` functions, as follows:

When the result of any operation is beyond the possible values for a type, it overflows. This typically results in the number being wrapped around from the maximum to the minimum, as in the following code:

```
julia> 9223372036854775806 + 1  
9223372036854775807  
  
julia> 9223372036854775806 + 1 + 1  
-9223372036854775808  
  
julia> typemin(Int64)  
-9223372036854775808
```

Another way to think about an overflow is that, to represent larger numbers, additional bits are required in the most significant positions. These bits are then chopped off, and the remaining bits are returned as the result. Thinking about it this way explains many counterintuitive results when it comes to overflows. Take a look at the following code:

```
julia> 2^64  
0  
julia> 2^65  
0
```

This behavior is very different from what is observed in popular dynamic languages, such as Ruby and Python. In these languages, every basic mathematical operation includes an overflow check. When the overflow is detected, the value is automatically upgraded to a wider type capable of storing the larger value. However, this causes a significant overhead to all numerical computation. Not only do we have to pay the cost for the extra CPU operation for the overflow check, but the conditional statement also prevents CPU pipelining from being effective. For this reason, Julia (as with Java and C) chooses to operate directly on machine integers and forgo all overflow checks.

This may be confusing and frustrating at first glance if you have a background in programming Python or Ruby, but this is the price you pay for high-performance computing. Once you understand that Julia's numbers are really close to the metal and designed to be directly operated on by the CPU, it should not be any more difficult to construct correctly behaving programs in practice.

BigInt

If you know your program needs to operate on large integers beyond the range of `Int32` or `Int64`, there are various options in Julia. First, if your numbers can still be bounded, there is `Int128`. However, for arbitrarily large integers, Julia has built-in support via the `BigInt` type. Run the following code:

```
julia> big(9223372036854775806) + 1 + 1  
9223372036854775808
```

```
julia> big(2)^64  
18446744073709551616
```

Operations on `Int128` are slower, and for `BigInts` they are much slower than for the default integers. However, we can use these in situations where they are warranted without compromising on the performance of computations that fit within the bounds of the default types.

The floating point

The default floating-point type is always 64-bits wide and is called `Float64`. This is true irrespective of the underlying machine and OS bit width. It is represented in memory using the IEEE 754 binary standard.

The IEEE 754 standard is the universally accepted technical standard for floating point operations in computer hardware and software. Almost all commonly used CPU types implement their floating-point support using this standard. As a result, storing numbers in this format means that the CPU (or rather the FPU—the floating point unit within the CPU) can operate on them natively and quickly.

The binary storage standard for 64-bit floating point numbers consists of 1 sign bit, 11 bits of exponent, and 52 bits of the mantissa (or the significand), as follows:

Unchecked conversions for unsigned integers

The basic integers described previously are all signed values. Unsigned integers can be specified using the `UInt64` and `UInt32` types. As with many other Julia types, the type conversions can be done via type constructors, as follows:

```
julia> UInt64(UInt32(1))  
0x0000000000000001
```

These conversions check for out-of-range values. They throw an error when trying to convert a value that does not fit in the resulting type, as follows:

```
julia> UInt32(UInt64(1))  
0x00000001
```

```
julia> UInt32(typemax(UInt64))  
ERROR: InexactError()  
in call at essentials.jl:56
```

The conditional check will have an overhead when performing this calculation, not only because of following out the CPU's instructions but also due to pipeline failures. In some situations, when working with binary data, it may be acceptable to truncate 64-bit values to 32-bit values without checking. In such situations, there is a shortcut in Julia, which is to use the `%` operator with the type, as in the following code:

```
julia> typemax(UInt64) % UInt32  
0xffffffff
```

Using this construct prevents any errors from being thrown for out-of-bound values, and it is much faster than the checked version of the conversion. This also works for other base unsigned types, such as `UInt16` and `UInt8`.

Trading performance for accuracy

In this book, we largely focus on performance. However, at this stage, it should be said that accurate math is usually an even bigger concern. All basic floating-point arithmetic in Julia follows strict IEEE 754 semantics. Rounding is handled carefully in all base library code to guarantee the theoretical best error limits. In some situations, however, it is possible to trade off performance for accuracy and vice versa.

The `@fastmath` macro

The `@fastmath` macro is a tool to loosen the constraints of IEEE floating point operations in order to achieve greater performance. It can rearrange the order of evaluation to something with is mathematically equivalent but that would not be the same for discrete floating point numbers due to rounding/error effects. It can also replace some intrinsic operations with their faster variants that do not check for NaN or Infinity. This results in faster operation but might cause a compromise in accuracy. This option is similar to the `-ffast-math` compiler option in clang or GCC.

As an example, consider the following code that calculates the finite difference between the elements of an array and then sums them. We can create two versions of the function that are identical except for the fact that one has the `@fastmath` annotation and one doesn't. Simply use the following code:

```
function sum_diff(x)
    n = length(x); d = 1/(n-1)
    s = zero(eltype(x))
    s = s + (x[2] - x[1]) / d
    for i = 2:length(x)-1
        s = s + (x[i+1] - x[i+1]) / (2*d)
    end
    s = s + (x[n] - x[n-1])/d
end

function sum_diff_fast(x)
    n=length(x); d = 1/(n-1)
    s = zero(eltype(x))
    @fastmath s = s + (x[2] - x[1]) / d
    @fastmath for i = 2:n-1
        s = s + (x[i+1] - x[i+1]) / (2*d)
    end
    @fastmath s = s + (x[n] - x[n-1])/d
end
```

We can note that the `@fastmath` macro can be used in front of statements or loops. In fact, it can be used in front of any block of code, including functions. Anything relevant within this block will be rewritten by the macro.

Benchmarking the two implementations shows that `@fastmath` provides an approximate $2.5x$ improvement over the base version. Take a look at the following:

```
julia> t=rand(2000);  
  
julia> sum_diff(t)  
46.636190420898515  
  
julia> sum_diff_fast(t)  
46.636190420898515  
  
julia> @benchmark sum_diff(t)  
===== Benchmark Results =====  
    Time per evaluation: 5.74 μs [5.68 μs, 5.81 μs]  
Proportion of time in GC: 0.00% [0.00%, 0.00%]  
    Memory allocated: 0.00 bytes  
    Number of allocations: 0 allocations  
    Number of samples: 3901  
    Number of evaluations: 82201  
    R2 of OLS model: 0.987  
    Time spent benchmarking: 0.53 s  
  
julia> @benchmark sum_diff_fast(t)  
===== Benchmark Results =====  
    Time per evaluation: 2.10 μs [2.09 μs, 2.11 μs]  
Proportion of time in GC: 0.00% [0.00%, 0.00%]  
    Memory allocated: 0.00 bytes  
    Number of allocations: 0 allocations  
    Number of samples: 4901  
    Number of evaluations: 213901  
    R2 of OLS model: 0.997  
    Time spent benchmarking: 0.50 s
```

This result is very much dependent on the nature of the computation. In many situations, the improvements are much lower. Also, in this case, the two functions return the exact same value, which is not true in the general case. The message, then, is to test and measure extensively when using this feature.

As with everything else in Julia, we can introspect and take a look at what changes the macro makes to our code. We can observe that the macro rewrites the intrinsic functions with its own `_fast` versions in the following code:

```
julia> macroexpand(:(@fastmath for i=2:n-1; s = s + (x[i+1] -  
x[i+1]) / (2*d); end))  
:(for i = 2:Base.FastMath.sub_fast(n,1) # none, line 1:  
    s =  
    Base.FastMath.add_fast(s,Base.FastMath.div_fast(Base.FastMath.sub_fas  
t(x[Base.FastMath.add_fast(i,1)],x[Base.FastMath.add_fast(i,1)]),Base  
.FastMath.mul_fast(2,d)))  
end)
```

The K-B-N summation

Adding a collection of floating point values is a very common operation, but it is surprisingly susceptible to the accumulation of errors. A naïve implementation—that is, adding elements from the first to the last—accumulates errors at the rate of $O(\sqrt{n})$, where n is the number of elements being summed. Julia's `sum` base uses a pairwise summation algorithm that does better by accumulating errors at $O(\sqrt{\log(n)})$ but is almost as fast. However, there exists a more complicated summation algorithm attributed to William Kahan whose error is bound by $O(1)$. This is implemented in Julia in the `sum_kbn` function.

In order to test the accuracy of `sum`, we will use a set of numbers that are particularly susceptible to rounding errors. The sum of the set of three numbers (1, -1, and 10^{-100}) should be 10^{-100} . However, as one of these numbers is much smaller than the other two, the result will be incorrectly rounded to 0. Take a look at the following code:

```
julia> sum([1 1e-100 -1])  
0.0  
  
julia> sum_kbn([1 1e-100 -1])  
1.0e-100  
  
julia> @benchmark sum([1 1e-100 -1])  
===== Benchmark Results =====  
Time per evaluation: 6.72 ns [6.68 ns, 6.75 ns]  
Proportion of time in GC: 0.00% [0.00%, 0.00%]  
Memory allocated: 0.00 bytes  
Number of allocations: 0 allocations
```

```
Number of samples: 10701
Number of evaluations: 53712201
R2 of OLS model: 0.991
Time spent benchmarking: 0.52 s
```

```
julia> @benchmark sum_kbn([1 1e-100 -1])
=====
Time per evaluation: 9.53 ns [9.47 ns, 9.60 ns]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 10601
Number of evaluations: 48829501
R2 of OLS model: 0.987
Time spent benchmarking: 0.52 s
```

In summary, the default `sum` function is adequate for most situations. It is fast and quite accurate. However, for pathological cases or when summing millions of elements, the `sum_kbn` function may give up some performance in favor of increased accuracy.

Subnormal numbers

Subnormal numbers (also sometimes called denormal) are very small floating point values near zero. Formally, they are numbers *smaller* than those that can be represented without leading zeros in the significand (for example, normal numbers). Typically, floating point numbers are represented without leading zeros in the significand. Leading zeros in the number are moved to the exponent (that is, 0.0123 is represented as 1.23×10^{-2}). Subnormal numbers are, therefore, numbers in which such a representation would cause the exponent to be lower than the minimum possible value. In such a situation, the significand is forced to have leading zeros. Much more detail on these numbers is available on Wikipedia at https://en.wikipedia.org/wiki/Denormal_number.

Subnormal numbers in Julia can be identified by the `issubnormal` function, as follows:

```
julia> issubnormal(1.0)
false
```

```
julia> issubnormal(1.0e-308)
true
```

Subnormal numbers are useful for a gradual underflow. Without them, for example, subtraction between extremely small values of floating point numbers might underflow to zero, causing subsequent *divide-by-zero* errors. This is shown in the following code:

```
julia> 3e-308 - 3.001e-308  
-1.0e-311  
  
julia> issubnormal(3e-308 - 3.001e-308)  
true
```

Subnormal numbers to zero

Subnormal numbers cause a significant slowdown on modern CPUs, sometimes by up to $100x$. This may be hard to track down because these performance problems can occur when the inputs take certain values even if we hold the algorithm constant. They manifest as unexplained, intermittent slowdowns.

One solution would be to force all subnormal numbers to be treated as zero. This will set a CPU flag that discards all the subnormal numbers and uses zero in its place. While this solves the performance problem, it should be used with care as it may cause accuracy and numerical stability problems. In particular, it is no longer true that $x-y == 0 \Rightarrow x == y$, as can be noted in the following code:

```
julia> set_zero_subnormals(true)  
true  
  
julia> 3e-308 - 3.001e-308  
-0.0  
  
julia> 3e-308 == 3.001e-308  
false  
  
julia> get_zero_subnormals()  
true
```

One of the ways subnormal numbers arise is when a calculation exponentially decays to zero. This gradual flattening of the curve results in many subnormal numbers being created and causes a sudden performance drop. As an example, we will take a look at one such computation here:

```
function timestep( b, a, dt )
    n = length(b)
    b[1] = 1
    two = eltype(b)(2)
    for i=2:n-1
        b[i] = a[i] + (a[i-1] - two*a[i] + a[i+1]) * dt
    end
    b[n] = 0
end

function heatflow( a, nstep )
    b = similar(a)
    o = eltype(a)(0.1)
    for t=1:div(nstep,2)
        timestep(b,a,o)
        timestep(a,b,o)
    end
end
```

We will then benchmark these functions with and without forcing subnormal numbers to zero. We can note a speedup by around two times by forcing subnormal numbers to zero. Take a look at the following:

```
julia> set_zero_subnormals(false)
true

julia> @benchmark heatflow(a, 1000)
=====
Time per evaluation: 4.19 ms [2.29 ms, 6.09 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 3.98 kb
Number of allocations: 1 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.46 s
```

```
julia> set_zero_subnormals(true)
true

julia> @benchmark heatflow(a, 1000)
=====
Time per evaluation: 2.20 ms [2.06 ms, 2.34 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 3.98 kb
Number of allocations: 1 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.25 s
```

Summary

In this chapter, we discussed how Julia uses a machine representation of numbers to achieve C-like performance for its arithmetic computations. We noted how to work within these design constraints and considered the edge cases that are introduced.

Working with single numbers, however, is the easy part. Most numerical computations, as we noted throughout this book, consist of working on large sets of numbers. In the next chapter, we will take a look at how to make arrays perform fast.

6

Fast Arrays

It should not be a surprise to readers of this book that array operations are often the cornerstone of scientific and numeric programming. While arrays are a fundamental data structure in all programming, there are special considerations when they are used in numerical programming. One particular difference is that arrays are not just viewed as entities for data storage. Rather, they represent the fundamental mathematical structures of vectors and matrices.

In this chapter, we will discuss how to use arrays in Julia in the fastest possible way. When you profile your program, you will find that, in many cases, the majority of its execution time is spent in array operations. Therefore, the discussions in this chapter will likely turn out to be crucial in creating high-performance Julia code. The following are the topics we will cover:

- Array internals and storage
- Bounds checks
- In-place operations
- Subarrays
- SIMD parallelization
- Yepp! for fast vector operations
- Writing generic library functions using arrays

Array internals in Julia

We discussed how Julia's performance comes out of using *type* information to compile specific and fast machine code for different data types. Nowhere is this more apparent than in array-related code. This is probably where all of Julia's design choices pay off in creating high-performance code.

Array representation and storage

An array type in Julia is parameterized by the type of its elements and the number of its dimensions. Hence, the type of an array is represented as `Array{T, N}`, where `T` is the type of its elements, and `N` is the number of dimensions. So, for example, `Array{UTF8String, 1}` is a one-dimensional array of strings, while `Array{Float64, 2}` is a two-dimensional array of floating point numbers.

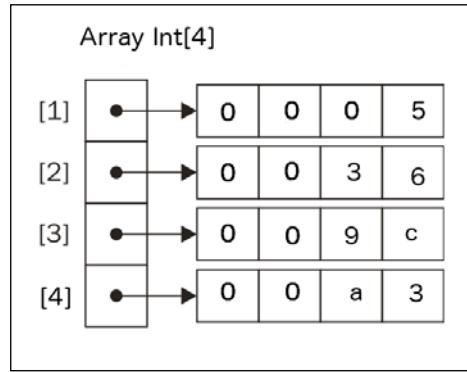
 **Type parameters**
You must have realized that type parameters in Julia do not always have to be other types; they can be constant values as well. This makes Julia's type system enormously powerful. It allows the type system to represent complex relationships and enables many operations to be moved to compile (or dispatch) time rather than at runtime.

Representing the type of an element within the type of arrays as a type parameter allows powerful optimization. It allows arrays of primitive types (and many immutable types) to be stored inline. In other words, the elements of the array are stored within the array's own primary memory allocation.

In the following diagram, we will show this storage mechanism. The numbers in the top row represent array indexes, while the numbers in the boxes are the integer elements stored within the array. The numbers in the bottom row represent the memory addresses where each of these elements is stored:

	[1]	[2]	[3]	[4]	[5]	[6]
Values	15	22	32	56	34	55
	1000	1004	1008	1012	1016	1020

In most other dynamic languages, all arrays are stored using pointers to their values. This is usually because the language runtime does not have enough information about the types of values to be stored in an array and hence cannot allocate the correctly sized storage. As represented in the following figures, when an array is allocated, contiguous storage simply consists of pointers to the actual elements, even when these elements are primitive types that can be stored natively in memory.



This method of storing arrays inline, without pointer indirection as much as possible, has many advantages and, as we discussed earlier, is responsible for much of Julia's performance claims. In other dynamic languages, the type of every element of the array is uncertain and the compiler has to insert type checks on each access. This can quickly add up and become a major performance drain.

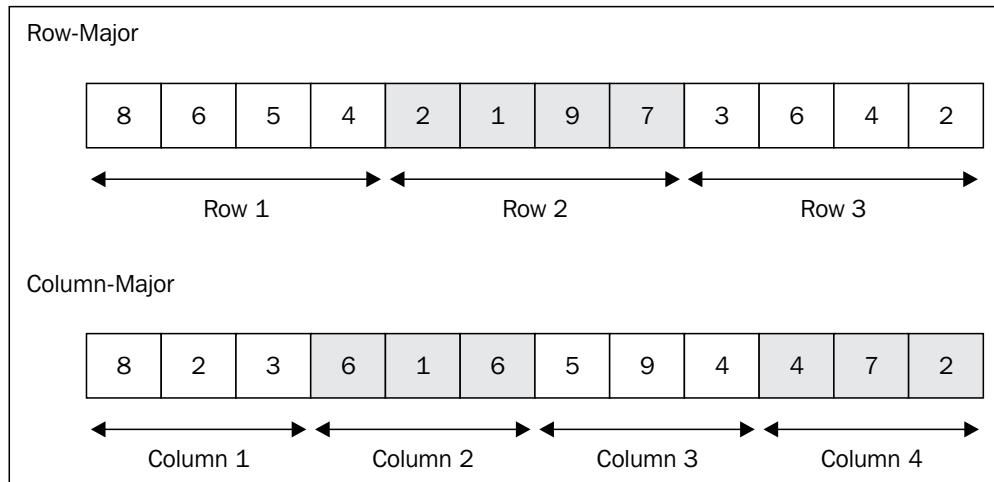
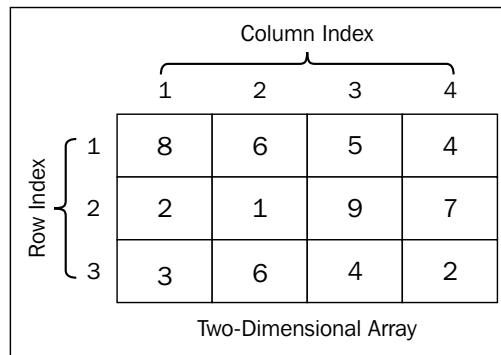
Further, even when every element of the array is of the same type, we pay the price of memory load for every array element if they are stored as pointers. Given the relative costs of a CPU operation versus a memory load on a modern processor, not doing this is a huge benefit.

There are other benefits too. When the compiler and CPU notice operations on a contiguous block of memory, CPU pipelining and caching are much more efficient. Some CPU optimizations, such as **Single Instruction Multiple Data (SIMD)**, are also unavailable when using indirect array loads.

Column-wise storage

When an array has only one dimension, its elements can be stored one after the other in a contiguous block of memory. As we observed in the previous section, operating on this array sequentially from its starting index to its end can be very fast, being amenable to many compiler and CPU optimizations.

Two-dimensional or higher arrays can, however, be stored in two different ways. We can store them row-wise or column-wise. In other words, we can store from the beginning of the array the elements of the first row, followed by the elements of the second row, and so on. Alternatively, we can store the elements of the first column, then the elements of the second column, and so on.



Arrays in C are stored as row-ordered. Julia, on the other hand, chooses the latter strategy, storing arrays as column-ordered, similar to MATLAB and Fortran. This rule applies to higher-dimensional arrays as well. In Julia, the array is stored with the last dimension first.

Naming convention



Conventionally, the term *row* refers to the first dimension of a two-dimensional array, and *column* refers to the second dimension. As an example, for a two-dimensional array of `x::Array{Float64, 2}` floats, the expression `x[2, 4]` refers to the elements in the second row and the fourth column.

This particular strategy of storing arrays has implications for how we navigate them. The most efficient way to read an array is in the same order in which it is laid out in memory. That is, each sequential read should access contiguous areas in memory.

We can demonstrate the performance impact of reading arrays in sequence with the following code, which squares and sums the elements of a two-dimensional floating point array, writing the result at each step back to the same position. This code exercises both the read and write operations for the array:

```
function col_iter(x)
    s=zero(eltype(x))
    for i = 1:size(x, 2)
        for j = 1:size(x, 1)
            s = s + x[j, i] ^ 2
            x[j, i] = s
        end
    end
end

function row_iter(x)
    s=zero(eltype(x))
    for i = 1:size(x, 1)
        for j = 1:size(x, 2)
            s = s + x[i, j] ^ 2
            x[i, j] = s
        end
    end
end
```

The `row_iter` function operates on the array in the first order row, while the `col_iter` function operates on the array in the first order column. We expect, based on the description of the previous array storage, that the `col_iter` function would be considerably faster than the `row_iter` function. Running the benchmarks, this is indeed what we see, as follows:

```
julia> a = rand(1000, 1000);

julia> @benchmark col_iter(a)
=====
Time per evaluation: 2.37 ms [1.64 ms, 3.10 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
    Memory allocated: 0.00 bytes
    Number of allocations: 0 allocations
    Number of samples: 100
    Number of evaluations: 100
    Time spent benchmarking: 0.28 s

julia> @benchmark row_iter(a)
=====
Time per evaluation: 6.53 ms [4.99 ms, 8.08 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
    Memory allocated: 0.00 bytes
    Number of allocations: 0 allocations
    Number of samples: 100
    Number of evaluations: 100
    Time spent benchmarking: 0.71 s
```

The difference between the two is quite significant. Column major access is more than twice as fast. This kind of difference in the inner loop of an algorithm can make a very noticeable difference in the overall runtime. It is, therefore, crucial to consider the order in which multidimensional arrays are processed when writing performance-sensitive code.

Bound checking

Like most dynamic languages, the Julia runtime performs bound checks on arrays by default. This means that the Julia compiler and runtime verify that the arrays are not indexed outside their limits and that all the indexes lie between the actual start and end of an array. Reading values of memory mistakenly beyond the end of an array is often the cause of many bugs and security issues in unmanaged software. Hence, bound checking is an important determinant of safety in your programs.

Removing the cost of bound checking

However, as with any extra operation, bound checking has costs too. There are extra operations for all array reads and writes. While this cost is reasonably small and is usually a good trade-off for safety, in some situations, where it can be guaranteed that the array bounds are never crossed, it may be worthwhile to remove these checks. This is possible in Julia using the `@inbounds` macro, as follows:

```
function prefix_bounds(a, b)
    for i = 2:size(a, 1)
        a[i] = b[i-1] + b[i]
    end
end

function prefix_inbounds(a, b)
    @inbounds for i = 2:size(a, 1)
        a[i] = b[i-1] + b[i]
    end
end
```

The `@inbounds` macro can be applied in front of a function or loop definition. Once this is done, all bound checking is disabled within the code block annotated with this macro. The performance benefit of doing this is small but may be significant overall for hot inner loops. Take a look at the following code:

```
julia> @benchmark prefix_bounds(x, y)
=====
Time per evaluation: 1.78 ms [1.72 ms, 1.83 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.20 s
```

```
julia> @benchmark prefix_inbounds(x, y)
=====
Time per evaluation: 1.50 ms [1.24 ms, 1.76 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
    Memory allocated: 0.00 bytes
    Number of allocations: 0 allocations
    Number of samples: 100
    Number of evaluations: 100
    Time spent benchmarking: 0.17 s
```

The `@inbounds` annotation should only be used when it can be guaranteed that the array access within the annotated block will never be out of bounds. This typically should be only when the limits of the loop depend directly on the length of the array—that is, for code of the `for i in 1:length(array)` form. If the programmer disables bound checking for some code and the array access is actually out of bounds, the results will be undefined. At best, the program will crash quickly.

Configuring bound checks at startup

The Julia runtime can use a command-line flag to set up bound-checking behavior for the entire session. The `-check-bounds` option can take two values: `yes` and `no`. These options will override any macro annotation in the source code.

When the Julia environment is started with `-check-bounds=yes`, all `@inbounds` annotations in code are ignored, and bound checks are mandatorily performed. This option is useful when running tests to ensure that code errors are properly reported and debugged if any.

Alternatively, when the Julia runtime is started with `-check-bounds=no`, no bound checking is done at all. This is equivalent to annotating all array access with the `@inbounds` macro. This option should only be used sparingly in the case of extremely performance-sensitive code, in which the system is very well tested and with minimal user inputs.

Allocations and in-place operations

Consider the following trivial function, `xpow`, which takes an integer as input and returns the first few powers of the number. Another function, `xpow_loop`, uses the first function to compute the sum of squares of a large sequence of numbers, as follows:

```
function xpow(x)
    return [x x^2 x^3 x^4]
end

function xpow_loop(n)
    s = 0
    for i = 1:n
        s = s + xpow(i) [2]
    end
    return s
end
```

Benchmarking this function for a large input shows that this function is quite slow, as follows:

```
julia> @benchmark xpow_loop(1000000)
=====
Benchmark Results
=====
Time per evaluation: 103.17 ms [101.39 ms, 104.95 ms]
Proportion of time in GC: 13.15% [12.76%, 13.53%]
Memory allocated: 152.58 mb
Number of allocations: 4999441 allocations
Number of samples: 97
Number of evaluations: 97
Time spent benchmarking: 10.16 s
```

The clue is in the number of allocations displayed in the preceding output. Within the `xpow` function, a four-element array is allocated for each invocation of this function. This allocation and the subsequent garbage collection take a significant amount of time. The `Proportion of time in GC` statistic displayed in the preceding code snippet also hints at this problem.

Preallocating function output

Note that, in the `xpow_loop` function, we only require one array at a time to compute our result. The array returned from one `xpow` call is differenced in the next iteration of the loop. This suggests that all these allocations for new array are a waste, and it may be easier to preallocate a single array to hold the result for each iteration, as follows:

```
function xpow! (result::Array{Int, 1}, x)
    @assert length(result) == 4
    result[1] = x
    result[2] = x^2
    result[3] = x^3
    result[4] = x^4
end

function xpow_loop_noalloc(n)
    r = [0, 0, 0, 0]
    s = 0
    for i = 1:n
        xpow!(r, i)
        s = s + r[2]
    end
    s
end
```

Note that the `xpow!` function now has an exclamation mark in its name. This Julia convention denotes that this function takes an output variable that mutates as an argument. We allocate a single variable outside the loop in the `xpow_loop_noalloc` function and then use it in all loop iterations to store the result of the `xpow!` function. Take a look at the following code:

```
@benchmark xpow_loop_noalloc(1000000)
=====
Time per evaluation: 11.02 ms [10.47 ms, 11.57 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 96.00 bytes
Number of allocations: 1 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 1.13 s
```

The result of this change is quite impressive. The runtime of the function, doing the same computation, decreases by an order of magnitude. Even more impressively, instead of millions of allocations, the program got by with only a single allocation.

The message, then, is simple: pay attention to what allocations happen within your inner loops. Julia provides you with simple tools to track this, so this is easy to fix. In fact, we don't need a full-fledged benchmarking infrastructure to figure this out. The simple `@time` macro also displays the allocations clearly, as shown by the following code:

```
julia> @time xpow_loop(1000000)
0.115578 seconds (5.00 M allocations: 152.583 MB, 21.99% gc time)

julia> @time xpow_loop_noalloc(1000000)
0.011720 seconds (5 allocations: 256 bytes)
```

Mutating versions

Given what we discussed in the previous section about the benefits of preallocating output, it should come as no surprise that many base library functions in Julia have mutating counterparts that modify their arguments rather than allocating a new output structure.

For example, the `sort` base library function, which sorts an array, allocates a new array of the same size as its input to hold its output: the sorted array. On the other hand, `sort!` makes an in-place sorting operation, in which the input array is itself sorted, as follows:

```
Julia> @benchmark sort!(a)
=====
Benchmark Results
=====
Time per evaluation: 15.92 ms [15.16 ms, 16.69 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 1.63 s
```

```
julia> @benchmark sort(a)
=====
Time per evaluation: 18.51 ms [17.22 ms, 19.80 ms]
Proportion of time in GC: 4.78% [0.34%, 9.22%]
Memory allocated: 7.63 mb
Number of allocations: 4 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 1.90 s
```

In this case, while the performance difference is significant, note that the allocating version of the function spends a significant proportion of its time in garbage collection and allocates a large amount of memory.

Array views

Julia, similarly to most scientific languages, has a very convenient syntax for array slicing. Consider the following example that sums each column of a two-dimensional matrix. First, we will define a function that sums the elements of a vector to produce a scalar. We will then use this function inside a loop to sum the columns of a matrix, passing each column one by one to our vector adder, as follows:

```
function sum_vector(x::Array{Float64, 1})
    s = 0.0
    for i = 1:length(x)
        s = s + x[i]
    end
    return s
end

function sum_cols_matrix(x::Array{Float64, 2})
    num_cols = size(x, 2)
    s = zeros(num_cols)
    for i = 1:num_cols
        s[i] = sum_vector(x[:, i])
    end
    return s
end
```

The `x[:, j]` syntax denotes all the row elements of the j^{th} column. In other words, it slices a matrix into its individual columns. Benchmarking this function, we will notice that the allocations and GC times are quite high. Take a look:

```
julia> @benchmark sum_cols_matrix(rand(1000, 1000))
=====
Time per evaluation: 4.45 ms [3.45 ms, 5.46 ms]
Proportion of time in GC: 17.55% [3.19%, 31.91%]
Memory allocated: 7.76 mb
Number of allocations: 3979 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.48 s
```

The reason for the high allocation is the fact that in Julia, array slices create a copy of the slice. In other words, for every `x[:, j]` slice operation in the preceding code snippet, a new vector is allocated to hold the column, and the element values are copied into it from the original matrix. This obviously causes a large overhead in this kind of algorithms.

What we would like in this case is to create a vector representing one column of the matrix that shares its storage with the original array. This saves a significant amount of allocation and copying.

Julia 0.4 includes a `sub()` function, which does exactly this. It returns a new array that is actually a view into the original array. Creating a `SubArray` is very fast, much faster than creating a sliced copy. Accessing a `SubArray` can be slower than accessing a regular dense array, but Julia's standard library has some extremely well-tuned code for this purpose. This code achieves performance nearly on a par with using regular arrays.

Using `sub()`, we can rewrite our `sum_cols_matrix` function to reduce the allocations due to slicing. However, first, we need to loosen the parameter type of `sum_vector`, as we will now pass `SubArray` to this function. The `SubArray` type is a subtype of `AbstractArray`, but it is obviously a different type than the `Array` concrete type, which denotes dense, contiguous stored arrays. Take a look at the following code:

```
function sum_vector(x::AbstractArray)
    s = 0.0
    for i = 1:length(x)
        s = s + x[i]
    end
```

```
    return s
end

function sum_cols_matrix_views(x::Array{Float64, 2})
    num_cols = size(x, 2); num_rows = size(x, 1)
    s = zeros(num_cols)
    for i = 1:num_cols
        s[i] = sum_vector(sub(x, 1:num_rows, i))
    end
    return s
end
```

We can note that this function, which uses the views of arrays to operate on portions of them, is significantly faster than using slices and copies. Most importantly, in the following benchmark, the number of allocations and the time spent in GC are much lower, as follows:

```
julia> @benchmark sum_cols_matrix_views(rand(1000, 1000))
=====
Time per evaluation: 1.38 ms [1.06 ms, 1.71 ms]
Proportion of time in GC: 0.81% [0.00%, 5.64%]
    Memory allocated: 101.64 kb
    Number of allocations: 3001 allocations
        Number of samples: 100
    Number of evaluations: 100
Time spent benchmarking: 0.18 s
```

SIMD parallelization

SIMD is the method of parallelizing computation whereby a single operation is performed on many data elements simultaneously. Modern CPU architectures contain instruction sets that can do this, operating on many variables at once.

Say you want to add two vectors, placing the result in a third vector. Let's imagine that there is no standard library function to achieve this, and you were writing a naïve implementation of this operation. Execute the following code:

```
function sum_vectors!(x, y, z)
    n = length(x)
    for i = 1:n
        x[i] = y[i] + z[i]
    end
end
```

Say the input arrays to this function has 1,000 elements. Then, the function essentially performs 1,000 sequential additions. A typical SIMD-enabled processor, however, can add maybe eight numbers in one CPU cycle. Adding each of the elements sequentially can, therefore, be a waste of CPU capabilities.

On the other hand, rewriting code to operate on parts of the array in parallel can get complex quickly. Doing this for a wide range of algorithms can be an impossible task. Julia, as you would expect, makes this significantly easier using the `@simd` macro. Placing this macro against a loop gives the compiler the freedom to use SIMD instructions for the operations within this loop if possible, as shown in the following code:

```
function sum_vectors_simd!(x, y, z)
    n = length(x)
    @inbounds @simd for i = 1:n
        x[i] = y[i] + z[i]
    end
end
```

With this one change to the function, we can now achieve significant performance gains on this operation, as follows:

```
julia> @benchmark sum_vectors!(zeros(Float32, 1000000), rand(Float32, 1000000), rand(Float32, 1000000))
```

```
===== Benchmark Results =====
```

```
Time per evaluation: 1.88 ms [1.73 ms, 2.03 ms]
```

```
Proportion of time in GC: 0.00% [0.00%, 0.00%]
```

```
Memory allocated: 0.00 bytes
```

```
Number of allocations: 0 allocations
```

```
Number of samples: 100
```

```
Number of evaluations: 100
```

```
Time spent benchmarking: 0.24 s
```

```
julia> @benchmark sum_vectors_simd!(zeros(Float32, 1000000), rand(Float32, 1000000), rand(Float32, 1000000))
```

```
===== Benchmark Results =====
```

```
Time per evaluation: 1.02 ms [980.93 µs, 1.06 ms]
```

```
Proportion of time in GC: 0.00% [0.00%, 0.00%]
```

```
Memory allocated: 0.00 bytes
```

```
Number of allocations: 0 allocations
```

```
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.24 s
```

There are a few limitations to using the `@simd` macro. This does not make every loop faster. In particular, note that using SIMD implies that the order of operations within and across the loop might change. The compiler needs to be certain that the reordering will be safe before it attempts to parallelize a loop. Therefore, before adding `@simd` annotation to your code, you need to ensure that the loop has the following properties:

- All iterations of the loop are independent of each other. That is, no iteration of the loop uses a value from a previous iteration or waits for its completion. The significant exception to this rule is that certain reductions are permitted.
- The arrays being operated upon within the loop do not overlap in memory.
- The loop body is straight-line code without branches or function calls.
- The number of iterations of the loop is obvious. In practical terms, this means that the loop should typically be expressed on the length of the arrays within it.
- The subscript (or index variable) within the loop changes by one for each iteration. In other words, the subscript is unit stride.
- Bounds checking is disabled for SIMD loops. (Bound checking can cause branches due to exceptional conditions.)

To check whether the compiler successfully vectorized your code, use the `@code_llvm` macro to inspect the generated LLVM bitcode. While the output might be long and inscrutable, the keywords to look for in the output are sections prefixed with `vector` and vectorized operations that look similar to `<n * float>`.

The following is an extract from the output of `@code_llvm` for the function we ran before, showing a successful vectorization of the operations. Thus, we know that the performance gains we observed are indeed coming from an automatic vectorization of our sequential code:

```
julia> @code_llvm sum_vectors_simd! (zeros(Float32, 1000000),
rand(Float32, 1000000), rand(Float32, 1000000))
.....
vector.ph: ; preds = %if3
%n.vec = sub i64 %20, %n.mod.vf
%28 = sub i64 %n.mod.vf, %20
```

```
br label %vector.body

vector.body:                                ; preds =
%vector.body, %vector.ph

%lsr.iv42 = phi i64 [ %lsr.iv.next43, %vector.body ], [ 0,
%vector.ph ]

%29 = mul i64 %lsr.iv42, -4
%uglygep71 = getelementptr i8* %25, i64 %29
%uglygep7172 = bitcast i8* %uglygep71 to <8 x float>*
%wide.load = load <8 x float>* %uglygep7172, align 4
%30 = mul i64 %lsr.iv42, -4
%sunkaddr = ptrtoint i8* %25 to i64
%sunkaddr73 = add i64 %sunkaddr, %30
%sunkaddr74 = add i64 %sunkaddr73, 32
%sunkaddr75 = inttoptr i64 %sunkaddr74 to <8 x float>*
%wide.load14 = load <8 x float>* %sunkaddr75, align 4
```

Yepp!

Many algorithms for scientific computing compute transcendental functions (log, sin, and cos) on arrays of floating point values. These are heavily used operations with strict correctness requirements and thus have been the target of many optimization efforts over the years. Faster versions of these functions can have a huge impact on the performance of many applications in the scientific computing domain.

In this area, the **Yepp!** software suite can be considered state-of-the-art. Primarily written at Georgia Institute of Technology by Marat Dukhan, Yepp! provides optimized implementations of modern processors of these functions, which are much faster compared to the implementations in system libraries.

Julia has a very easy-to-use binding to Yepp! within a package. It can be installed using the in-built package management mechanism `Pkg.add("Yepp")`. Once installed, the functions are available with the `Yepp` module. There is no simpler way to get a $4x$ performance boost. With performance gains of this magnitude, there is little reason to use anything else for code where a large number of transcendental functions needs to be computed. Run the following code:

```
julia> @benchmark log(a)
=====
Benchmark Results =====
```

```
Time per evaluation: 17.41 ms [16.27 ms, 18.55 ms]
Proportion of time in GC: 5.08% [0.32%, 9.83%]
Memory allocated: 7.63 mb
Number of allocations: 2 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 1.81 s
```

```
julia> @benchmark Yeppp.log(a)
=====
Time per evaluation: 4.45 ms [3.54 ms, 5.35 ms]
Proportion of time in GC: 15.63% [1.55%, 29.71%]
Memory allocated: 7.63 mb
Number of allocations: 2 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.49 s
```

Yeppp also provides in-place versions of its functions that can be faster in many situations, saving allocations and subsequent garbage collection. The in-place version of log, for example, provides a 2x performance gain over the allocating version we ran before. Take a look at the following code:

```
julia> @benchmark Yeppp.log!(a)
=====
Time per evaluation: 2.34 ms [2.01 ms, 2.67 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.26 s
```

The Yeppp Julia package provides implementations of some common vectorized functions, including log, sin, exp, and sumabs. Refer to <https://github.com/JuliaLang/Yeppp.jl> for full details of its capabilities.

Writing generic library functions with arrays

The suggestions in the previous sections should make your array code fast and high-performance. If you are directly writing code to solve your own problems, this should be enough. However, if you are writing library routines that may be called by other programs, you will need to heed additional concerns. Your function may be called with arrays of different kinds and with different dimensions. To write generic code that is fast with all types and dimensions of arrays, your code needs to be careful in how it iterates over the elements of the arrays.

All Julia arrays are subtypes of the `AbstractArray` type. All abstract arrays must provide facilities for indexation and iteration. However, these can be implemented very differently for different types of arrays. The default array is `DenseArray`, which stores its elements in contiguous memory. As discussed before, these elements can be pointers or values, but in either case, they are stored in contiguous memory. This means that linear indexing is very fast for all these arrays. However, this is not true for all kinds of arrays.

Linear indexing

The term *linear indexing* refers to the ability of indexing a multidimensional array by a single scalar index. So, for example, if we have a three-dimensional array x with 10 elements in each dimension, it can be indexed with a single integer in the range of 1 to 1000. In other words, $x[1], x[2], \dots, x[10], x[11], \dots, x[99]$, and $x[100]$ are consecutive elements of the array. As described earlier, Julia arrays are stored in a major order column, so linear indexing runs through the array in this order. This makes linear indexing particularly cache-friendly because contiguous memory segments are accessed consecutively. In contrast, *cartesian indexing* uses the complete dimensions of the array to index it. The three-dimensional array x is indexed by three integers $x[i, j, k]$.



For example, subarrays can be efficiently indexed using cartesian indexing, but linear indexing is much slower due to the need to compute a div for each indexing operation. While cartesian indexing is useful when the dimensions of an array are known, generic code typically uses linear indexing to work with multidimensional arrays. This, then, may create performance pitfalls.

As an example of a function that can work with generic multidimensional arrays, let's write a simple function that sums all the elements in an array, as follows:

```
function mysum_linear(a::AbstractArray)
    s=zero(eltype(a))
    for i = 1:length(a)
        s=s+a[i]
    end
    return s
end
```

This function works with arrays of any type and dimension, as we can note in the test calls in the following code, in which we call it with a range—a three-dimensional array, a two-dimensional array, and a two-dimensional subarray, respectively:

```
julia> mysum_linear(1:1000000)
500000500000

julia> mysum_linear(reshape(1:1000000, 100, 100, 100))
500000500000

julia> mysum_linear(reshape(1:1000000, 1000, 1000))
500000500000

julia> mysum_linear(sub(reshape(1:1000000, 1000, 1000), 1:500, 1:500) )
62437625000
```

If we benchmark these functions, we will note that calling the same function on a subarray is significantly slower than calling it on a regular dense array.

```
julia> @benchmark mysum_linear(reshape(1:1000000, 1000, 1000))
===== Benchmark Results =====
Time per evaluation: 808.98 μs [728.67 μs, 889.28 μs]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.33 s
```

```
julia> @benchmark mysum_linear(sub(reshape(1:1000000, 1000, 1000), 1:500, 1:500) )
=====
Time per evaluation: 11.39 ms [10.23 ms, 12.55 ms]
Proportion of time in GC: 4.97% [0.75%, 9.19%]
Memory allocated: 7.61 mb
Number of allocations: 498989 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 1.34 s
```

In situations such as this where we want to write generic functions that can be performant with different kinds of arrays, the advice is to not use linear indexing. So, what should we use?

The simplest option is to directly iterate the array rather than iterating its indices. The iterator for each kind of array will choose the most optimal strategy for high performance. Hence, the code to add the elements of a multidimensional array can be written as follows:

```
function mysum_in(a::AbstractArray)
    s = zero(eltype(a))
    for i in a
        s = s + i
    end
end
```

If we benchmark this function, we can see the difference in performance, as follows:

```
julia> @benchmark mysum_in(sub(reshape(1:1000000, 1000, 1000), 1:500, 1:500) )
=====
Time per evaluation: 354.25 μs [347.11 μs, 361.39 μs]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.23 s
```

This strategy is usable when the algorithm only requires the elements of the array and not its indexes. If the indexes need to be available within the loop, they can be written using the `eachindex()` method. Each array defines an optimized `eachindex()` method that allows the iteration of its index efficient. We can then rewrite the sum function as follows, even though, for this particular function, we do not actually need indexes:

```
function mysum_eachindex(a::AbstractArray)
    s = zero(eltype(a))
    for i in eachindex(a)
        s = s + a[i]
    end
end
```

The benchmark numbers demonstrate an order of magnitude improvement in the speed of these functions when not using linear indexing for subarrays. Writing code in this manner, therefore, allows our function to be used correctly and efficiently by all manner of arrays in Julia. Take a look at the following:

```
Julia> @benchmark mysum_eachindex(sub(reshape(1:1000000, 1000, 1000),
1:500, 1:500) )
=====
Benchmark Results =====
Time per evaluation: 383.06 μs [363.04 μs, 403.07 μs]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
Memory allocated: 0.00 bytes
Number of allocations: 0 allocations
Number of samples: 100
Number of evaluations: 100
Time spent benchmarking: 0.22 s
```

Summary

In this chapter, we covered the performance characteristics in Julia of the most important data structure in scientific computing: the array. We discussed why Julia's design enables extremely fast array operations and how to get the best performance in our code when operating on arrays. This brings us to the end of our journey creating the fastest possible code in the Julia. Using all the tips discussed until now, the performance of your code should approach that of well-written C.

Sometimes, however, this isn't enough; we want higher performance. Our data may be larger or our computations intensive. In this case, the only option is to parallelize our processing using multiple CPUs and systems. In the next chapter, we will take a brief look at the features that Julia provides to write parallel systems easily.

7

Beyond the Single Processor

Throughout this book, we discussed ways to make our code run faster and more efficiently. Using the suggestions in the previous chapters, your code now fully utilizes the processor without much overhead or wastage. However, if you still need your computation to finish even earlier, the only solution is distributing the computation over multiple cores, processors, and machines. In this chapter, we will briefly discuss some of the facilities available in Julia for distributed computing. A complete exposition of this topic is probably the subject of another large book – this chapter can only provide a few pointers for further information, such as:

- Parallelism in Julia
- Programming parallel tasks
- Shared memory arrays

Parallelism in Julia

Julia is currently a single-threaded language (although it does perform asynchronous I/O). This means that the Julia code that you write will run sequentially on a single core of the machine. There are a few significant exceptions; Julia has asynchronous I/O that can offload network or file access to a separate operating system thread, and some libraries embedded within Julia, such as OpenBLAS, spawn and manage multiple threads for their computations. Notwithstanding these exceptions, most user-written Julia code is limited to a single core.

Julia, however, contains an easy-to-use multiprocessor mechanism. You can start multiple Julia processes either on a single host or across a network, and you can control, communicate, and execute programs across the entire cluster.

Starting a cluster

The communication between Julia processes is *one-sided* in the sense of there being a master process that accepts the user's inputs and controls all the other processes. Starting a cluster, therefore, involves either a command-line switch while starting the master Julia process or calling methods from REPL. At its simplest, the `-p n` option while starting Julia creates n additional processes on the local host, as can be seen in the following:

```
$ ./julia -p 2

 _ _ _ _ _ | A fresh approach to technical computing
( ) ( ) ( ) | Documentation: http://docs.julialang.org
_ _ _ | | _ _ _ | Type "?help" for help.
| | | | | | / _` | |
| | | | | | ( | | | Version 0.4.3-pre+6 (2015-12-11 00:38 UTC)
_| / \_\_'\_\_|\_\_'\_| | Commit adffe19* (63 days old release-0.4)
|__/ | x86_64-apple-darwin15.2.0
```

The `procs()` method can be used to inspect the cluster. It returns the IDs of all the Julia processes that are available. We can note in the following that we have three processes available—the master and two child processes:

```
julia> procs()
3-element Array{Int64,1}:
 1
 2
 3
```

The `addprocs(n)` method creates additional processes connected to the same master. It behaves similarly to the `-p n` option but is a pure Julia function that can be called from REPL or other Julia code, as follows:

```
julia> addprocs(2)
2-element Array{Int64,1}:
 4
 5

julia> procs()
```

```
5-element Array{Int64,1}:
1
2
3
4
5
```

These commands launch multiple Julia processes on the same machine. This is useful to the extent of running as many Julia processes as the number of cores on this host. Beyond this, you can start processes on other hosts by providing the hostname to the `addprocs` call, as follows:

```
julia> addprocs(["10.0.2.1", "10.0.2.2"])
```

```
7-element Array{Int64,1}:
1
2
3
4
5
6
7
```

This invocation, by default, uses **Secure Shell (SSH)** to connect to and start Julia processes on remote machines. There are, of course, many different configuration options possible for this setup, including the ability to use other protocols to control and communicate between processes. All this and more is described in detail in the manual at <http://docs.julialang.org/en/release-0.4/manual/parallel-computing/#clustermanagers>.

Communication between Julia processes

The primitive facilities provided by Julia to move code and data within a cluster of processes consist of *remote references* and *remote calls*. As the name suggests, a remote reference consists of a reference to data residing on a different Julia process. Thereby, values can be retrieved from (or written to) such a reference.

A remote call, on the other hand, is a request to execute a function on a particular node. Such a call is asynchronous in that a remote call finishes immediately, returning the `RemoteRef` object, which is a reference to its result. The arguments to `remotecall` are the function name, the process number to execute the function in, and the arguments to this function. The caller, then, has the option to `wait()` on the reference until the call completes and then `fetch()` the result into its own process, as shown in the following code:

```
julia> a = remotecall(2,sqrt, 4.0)
RemoteRef{Channel{Any}}(2,1,3)
```

```
julia> wait(a)
RemoteRef{Channel{Any}}(2,1,3)
```

```
julia> fetch(a)
2.0
```

For simple uses, the `remotecall_fetch` function can combine these two steps and return the function result at once, as follows:

```
julia> remotecall_fetch(2, sqrt, 4.0)
2.0
```

Programming parallel tasks

The low-level facilities that we saw in the previous section are quite flexible and very powerful. However, they leave a lot to be desired in terms of ease of use. Julia, therefore, has built-in set of higher-level programming tools that make it much easier to write parallel code. We will discuss some of them in the next section.

@everywhere

The `@everywhere` macro is used to run the same code in all the processes in the cluster. This is useful to set up the environment to run the actual parallel computation later. The following code loads the `Distributions` package and calls the `rand` method on all the nodes simultaneously, as follows:

```
julia> @everywhere using Distributions
```

```
julia> @everywhere rand(Normal())
```

@spawn

The @spawn macro is a simpler way to run a function in a remote process without having to specify the remote node or having to work through ambiguous syntax. Take a look at the following code:

```
julia> a=@spawn randn(5,5)^2
RemoteRef{Channel{Any}} (2,1,240)

julia> fetch(a)
5x5 Array{Float64,2}:
 -0.478348  -0.185402   6.21775   2.62166  -5.4774
 -3.22569   -1.56487   3.03402  -0.305334  -1.75827
 -2.9194    -0.0549954  0.922262  -0.117073  -0.281402
  0.709968   1.87017   -1.7031   0.343585  0.09105
  3.20311    0.49899   -0.202174  -0.337815  -1.81711
```

This macro actually creates a closure around the code being called on the remote node. This means that any variable declared on the current node will be copied over to the remote node. In the preceding code, the random array is created on the remote node. However, in the following code, the random array is created on the current node and copied to the remote node. Even though the two code extracts look similar, they have very different performance characteristics. Take a look at the following code:

```
julia> b=rand(5,5)
5x5 Array{Float64,2}:
 0.409983  0.852665  0.490156  0.481329  0.642901
 0.676688  0.0865577 0.59649   0.553313  0.950665
 0.591476  0.824942  0.440399  0.701106  0.321909
 0.137929  0.0138369 0.273889  0.677865  0.33638
 0.249115  0.710354  0.972105  0.617701  0.969487

julia> a=@spawn b^2
RemoteRef{Channel{Any}} (3,1,242)

julia> fetch(a)
5x5 Array{Float64,2}:
 1.26154   1.29108   1.68222   1.73618   2.01716
 1.00195   1.75952   1.7217    1.7541    1.81713
```

```
1.2381    1.17741   1.48089   1.72401   1.8542  
0.405205  0.593076  0.709137  0.933354  0.744132  
1.48451   1.77305   2.08556   2.21207   2.29608
```

Parallel for

Julia includes an inbuilt **parallel for** loop that can automatically distribute the computation within a for loop across all the nodes in a cluster. This can sometimes allow code to be sped up across machines with little programmer intervention.

In the following code, we will generate a million random numbers and add them. The first function computes each step serially, while the second function attempts to distribute the steps across the cluster. Each step in this loop can be computed independently and should thus be easy to parallelize:

```
function serial_add()  
    s=0.0  
    for i = 1:1000000  
        s=s+randn()  
    end  
    return s  
end  
  
function parallel_add()  
    return @parallel (+) for i=1:1000000  
        randn()  
    end  
end
```

We can note that the parallel function provides a significant performance improvement without the programmer having to manage the task distribution or internode communication explicitly. Now, take a look at the following code:

```
julia> @benchmark serial_add()  
===== Benchmark Results =====  
Time per evaluation: 6.95 ms [6.59 ms, 7.31 ms]  
Proportion of time in GC: 0.00% [0.00%, 0.00%]  
Memory allocated: 0.00 bytes  
Number of allocations: 0 allocations  
Number of samples: 100  
Number of evaluations: 100  
Time spent benchmarking: 0.86 s  
julia> @benchmark parallel_add()
```

```
===== Benchmark Results =====
Time per evaluation: 4.42 ms [4.25 ms, 4.60 ms]
Proportion of time in GC: 0.00% [0.00%, 0.00%]
    Memory allocated: 154.42 kb
    Number of allocations: 2012 allocations
        Number of samples: 100
    Number of evaluations: 100
Time spent benchmarking: 0.63 s
```

Parallel map

The parallel for loop we discussed in the previous section can perform a reduction (the addition in the previous code) and works well even if each step in the computation is lightweight. For code where each iteration is heavyweight, and there is no reduction to be done, the parallel map construct is useful. In the following code, we will create 10 large matrices and then perform a singular-value decomposition on each. We can note that parallelizing this computation can attain a significant speed improvement simply by changing one character in the code:

```
julia> x=[rand(100,100) for i in 1:10];
julia> @benchmark map(svd, x)
===== Benchmark Results =====
Time per evaluation: 327.77 ms [320.38 ms, 335.16 ms]
Proportion of time in GC: 0.13% [0.00%, 0.40%]
    Memory allocated: 5.47 mb
    Number of allocations: 231 allocations
        Number of samples: 29
    Number of evaluations: 29
Time spent benchmarking: 10.18 s
julia> @benchmark pmap(svd, x)
===== Benchmark Results =====
Time per evaluation: 165.30 ms [161.76 ms, 168.84 ms]
Proportion of time in GC: 0.10% [0.00%, 0.40%]
    Memory allocated: 1.66 mb
    Number of allocations: 2106 allocations
        Number of samples: 59
    Number of evaluations: 59
Time spent benchmarking: 10.11 s
```

Distributed arrays

The `DistributedArrays` package provides an implementation of partitioned multidimensional arrays. Detailed package documentation is available at <https://github.com/JuliaParallel/DistributedArrays.jl>. For the moment, it suffices to say that there exist facilities to partition datasets automatically at creation or manually, as well as distributing the computation to each node for operation on the local parts of the arrays.

Shared arrays

Distributed arrays are a fully generic solution that scales across many networked hosts in order to work on data that cannot fit in the memory of a single machine. However, in many circumstances, although the data does fit in the memory, we want multiple Julia processes to improve throughput by fully utilizing all the cores in a machine. In this situation, shared arrays are useful to get different Julia processes operating on the same data.

Shared arrays, as the name suggests, are arrays that are shared across multiple Julia processes on the *same* machine.

Constructing `SharedArray` requires specifying its type, its dimensions, and the list of process IDs that will have access to the array, as follows:

```
S=SharedArray( Float64, (100, 100, 5), pids=[2,3,4,5]);
```

Once a shared array is created, it is accessible in full to all the specified workers (on the same machine). Unlike a distributed array, the data is not partitioned, and hence there is no need for any data transfer between nodes. Therefore, when the data is small enough to fit in the memory but large enough to require multiple nodes to process, shared arrays are particularly useful. Not only are they highly performant in these situations, it is much easier to write code for them.

Threading

Shared arrays can be seen as some kind of shared memory multiprocessing in Julia. They are currently useful as Julia does not have first-class threads that can operate on shared memory. This is, however, being worked on as we speak, and it is likely that in the future versions of Julia, it will be possible to operate on shared memory arrays from multiple threads within the same process.

Summary

This chapter provided a very cursory glimpse into the parallel computing facilities built into the Julia language. While we didn't cover much in detail in this chapter, you have hopefully noted how easy it is to get started with distributed computation in Julia. With a little bit of help from the online documentation, it should be easy to create high performing distributed codebases in Julia.

Module 3

Mastering Julia

*Develop your analytical and programming skills further in Julia
to solve complex data processing problems*

1

The Julia Environment

In this chapter, we explore all you need to get started on Julia, to build it from source or to get prebuilt binaries. Julia can also be downloaded bundled with the Juno IDE. It can be run using IPython, and this is available on the Internet via the <https://juliabox.org/> website. Julia is a high-level, high-performance dynamic programming language for technical computing. It runs on Linux, OS X, and Windows. We will look at building it from source on CentOS Linux, as well as downloading as a prebuilt binary distribution. We will normally be using v0.3.x, which is the stable version at the time of writing but the current development version is v0.4.x and nightly builds can be downloaded from the Julia website.

Introduction

Julia was first released to the world in February 2012 after a couple of years of development at the Massachusetts Institute of Technology (MIT).

All the principal developers—Jeff Bezanson, Stefan Karpinski, Viral Shah, and Alan Edelman—still maintain active roles in the language and are responsible for the core, but also have authored and contributed to many of the packages.

The language is open source, so all is available to view. There is a small amount of C/C++ code plus some Lisp and Scheme, but much of core is (very well) written in Julia itself and may be perused at your leisure. If you wish to write exemplary Julia code, this is a good place to go in order to seek inspiration. Towards the end of this chapter, we will have a quick run-down of the Julia source tree as part of exploring the Julia environment.

Julia is often compared with programming languages such as Python, R, and MATLAB. It is important to realize that Python and R have been around since the mid-1990s and MATLAB since 1984. Since MATLAB is proprietary (® MathWorks), there are a few clones, particularly GNU Octave, which again dates from the same era as Python and R. Just how far the language has come is a tribute to the original developers and the many enthusiastic ones who have followed on. Julia uses GitHub as both for a repository for its source and for the registered packages. While it is useful to have Git installed on your computer, 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. While Julia has no simple built-in graphics, there are several different graphics packages and I will be devoting a chapter later particularly to these.

Philosophy

Julia was designed with scientific computing in mind. The developers all tell us that they came with a wide array of programming skills—Lisp, Python, Ruby, R, and MATLAB. Some like myself even claim to originate as Perl hackers. However, all need a *fast* compiled language in their armory such as C or Fortran as the current languages listed previously are pitifully slow.

So, to quote 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 interactive and we want it compiled."

(Did we mention it should be as fast as C?)"

<http://julialang.org/blog/2012/02/why-we-created-julia>

With the introduction of the **Low-Level Virtual Machine (LLVM)** compilation, it has become possible to achieve this goal and to design a language from the outset, which makes the two-language approach largely redundant.

Julia was designed as a language similar to other scripting languages and so should be easy to learn for anyone familiar to Python, R, and MATLAB. It is syntactically closest to MATLAB, but it is important to note that it is not a drop-in clone. There are many important differences, which we will look at later.

It is important not to be too overwhelmed by considering Julia as a challenger to Python and R. In fact, we will illustrate instances where the languages are used to complement each other. Certainly, Julia was not conceived as such, and there are certain things that Julia does which makes it ideal for use in the scientific community.

Role in data science and big data

Julia was initially designed with scientific computing in mind. Although the term "data science" was coined as early as the 1970s, it was only given prominence in 2001, in an article by William S. Cleveland, *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?

The following might be one definition:

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, 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 in the growth of the popularity of Python and R can be traced directly to the interest in data science.

So, what we set out to achieve in this book is 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 the cloud. Big data was originally the realm of Java largely because of the uptake of the Hadoop/HDFS framework, which, being written in Java, made it convenient to program MapReduce algorithms in it or any language, which runs on the JVM. This leads to an obscene amount of bloated boilerplate coding.

However, here, 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 is beginning to be considered an alternative to Java, but upon inspection, Julia makes an excellent candidate in this category too.

Comparison with other languages

Julia has the reputation for speed. The home page of the main Julia website, as of July 2014, includes references to benchmarks. The following table shows benchmark times relative to C (smaller is better, C performance = 1.0):

	Fortran	Julia	Python	R	MATLAB	Octave	Mathematica	Java Script	Go
fib	0.26	0.91	30.37	411.31	1992.0	3211.81	64.46	2.18	1.0
mandel	0.86	0.85	14.19	106.97	64.58	316.95	6.07	3.49	2.36
pi_sum	0.80	1.00	16.33	15.42	1.29	237.41	1.32	0.84	1.41
rand_mat_stat	0.64	1.66	13.52	10.84	6.61	14.98	4.52	3.28	8.12
rand_mat_mul	0.96	1.01	3.41	3.98	1.10	3.41	1.16	14.60	8.51

Benchmarks can be notoriously misleading; indeed, to paraphrase the common saying: *there are lies, damned lies, and benchmarks.*

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 uses NumPy, and C, Fortran, and Julia use OpenBLAS.

Julia provides a website for checking its performance: <http://speed.julialang.org>.

The source code for all the tests is available on GitHub. This is not just the Julia code but also that used in C, MATLAB, Python, and so on. Indeed, extra language examples are being added, and you will find benchmarks to try in Scala and Lua too:

<https://Github.com/JuliaLang/julia/tree/master/test/perf/micro>.

This table is useful in another respect too, as it lists all the major comparative languages of Julia. No real surprises here, except perhaps the range of execution times.

- **Python:** This has become the de facto data science language, and the range of modules available is overwhelming. Both version 2 and version 3 are in common usage; the latter is NOT a superset of the former and is around 10% slower. In general, Julia is an order of magnitude faster than Python, so often when the established Python code is compiled or rewritten in C.

- **R:** 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:** Another commercial product (® Wolfram Research) for general-purpose mathematical problems. There is 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 and Go:** These are linked together since they both use the Google V8 engine. V8 compiles to native machine code before executing it; hence, the excellent performance timings but both languages are more targeted at web-based applications.

So, Julia would seem to be an ideal language for tackling data science problems. It's important to recognize that many of the built-in functions in R and Python are not implemented natively but are written in C. Julia performs roughly as well as C, so Julia won't do any better than R or Python if most of the work you do in R or Python calls built-in functions without performing any explicit iteration or recursion.

However, when you start doing custom work, Julia will come into its own. It is the perfect language for advanced users of R or Python, who are trying to build advanced tools inside of these languages. The alternative to Julia is typically resorting to C; R offers this through **Rcpp**, and Python offers it through **Cython**.

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

Features

The Julia programming language is free and open source (MIT licensed), and the source is available on GitHub.

To the veteran programmer, it has looks and feels similar to MATLAB. Blocks created by the `for`, `while`, and `if` statements are all terminated by `end` rather than by `endfor`, `endwhile`, and `endif` or by using the familiar `{}` style syntax. However, it is not a MATLAB clone, and sources written for MATLAB will not run on Julia.

The following are some of Julia's features:

- Designed for parallelism and distributed computation (multicore and cluster)
- C functions called directly (no wrappers or special APIs needed)
- Powerful shell-like capabilities for managing other processes
- Lisp-like macros and other meta-programming facilities
- User-defined types are as fast and compact as built-ins
- LLVM-based, **just-in-time (JIT)** compiler that allows Julia to approach and often match the performance of C/C++
- An extensive mathematical function library (written in Julia)
- Integrated mature, best-of-breed C and Fortran libraries for linear algebra, random number generation, **Fast Fourier Transform (FFT)**, and string processing

Julia's core is implemented in C and C++, and its parser in Scheme; the LLVM compiler framework is used for the JIT generation of machine code.

The standard library is written in Julia itself by using Node.js's libuv library for efficient, cross-platform I/O.

Julia has a rich language of types for constructing and describing objects that can also optionally be used to make type declarations. It has the ability to define function behavior across many combinations of argument types via a multiple dispatch, which is the key cornerstone of language design.

Julia can utilize code in other programming languages by directly calling routines written in C or Fortran and stored in shared libraries or DLLs. This is a feature of the language syntax and will be discussed in detail later.

In addition, it is possible to interact with Python via **PyCall** and this is used in the implementation of the **IJulia** programming environment.

Getting started

Starting to program in Julia is very easy. The first place to look at is the main Julia language website: <http://julialang.org>. This is not blotted with graphics, just the Julia logo, some useful major links to other parts of the site, and a quick sampler on the home page.

The Julia documentation is comprehensive of the docs link: <http://docs.julialang.org>. There are further links to the Julia manual, the standard library, and the package system, all of which we will be discussing later. Moreover, the documentation can be downloaded as a PDF file, a zipped file of HTML pages, or an ePub file.

Julia sources

At present, we will be looking at the download link. This provides links to 32-bit and 64-bit distros for Windows, Mac OS X, CentOS, and Ubuntu; both the stable release and the nightly development snapshot. So, a majority of the users getting started require nothing more than a download and a standard installation procedure.

For Windows, this is by running the downloaded .exe file, which will extract Julia into a folder. Inside this folder is a batch file `julia.bat`, which can be used to start the Julia console.

For Mac OS X, the users need to click on the downloaded .dmg file to run the disk image and drag the *app* icon into the Applications folder. On Mac OS X, you will be prompted to continue as the source has been downloaded from the Internet and so is not considered secure.

Similarly, uninstallation is a simple process. In Windows, delete the `julia` folder, and in Mac OS X, delete `Julia.app`. To do a "clean" uninstall, it is also necessary to tidy up a few hidden files/folders, and we will consider this after talking about the package system.

For Ubuntu (Linux), it's a little bit more involved as you need to add a reference to **Personal Package Archive (PPA)** to your system. You will have to have the root privilege for this to execute the following commands:

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

Downloading the example code



You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books that you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

The releases are provided by Elliot Saba, and there is a separate PPA for the nightly snapshots: ppa:staticfloat/julianightlies.

It is only necessary to add PPA once, so for updates, all you need to do is execute the following command:

```
sudo apt-get update
```

Exploring the source stack

Let's look at the top-level folders in the source tree that we get from GitHub:

Folder	Contents
Base	Contains the Julia sources that make up the core
contrib	A miscellaneous set of scripts, configuration files, and so on
deps	Dependencies and patches
doc	The reStructuredText files to build the technical documentation
etc	The juliacrc file
examples	A selection of examples of Julia coding
src	The C/C++, Lisp, and Scheme files to build the Julia kernel
test	A comprehensive test suite
ui	The source for the console REPL

To gain some insight into Julia coding, the best folders to look at are `base`, `examples`, and `test`.

1. The `base` folder contains a great portion of the standard library and the coding style exemplary.
2. The `test` folder has some code that illustrates how to write test scripts and use the `Base.Test` system.
3. The `examples` folder gives Julia's take on some well-known old computing chestnuts such as *Queens Problem*, *Wordcounts*, and *Game of Life*.

If you have created Julia from source, you will have all the folders available in the Git/build folder; the build process creates a new folder tree in the folder starting with `usr` and the executable is in the `usr/bin` folder.

Installing on a Mac under OS X creates Julia in `/Applications/Julia-[version].app`, where `version` is the build number being installed. The executables required are in a subfolder of `Contents/Resources/julia/bin`. To find the Julia sources, look into the `share` folder and go down one level in to the `julia` subfolder.

So, the complete path will be similar to `/Applications/julia-0.2.1.app/Contents/Resources/julia/share/julia`. This has the Julia files but not the C/C++ and Scheme files and so on; for these, you will need to checkout a source tree from GitHub.

For Windows, the situation is similar to that of Mac OS X. The installation file creates a folder called `julia-[build-number]` where `build-number` is typically an alpha string such as `e44b593905`. Immediately under it are the `bin` and `share` folders (among others), and the `share` folder contains a subfolder named `julia` with the Julia scripts in it.

Juno

Juno is an IDE, which is bundled, for stable distributions on the Julia website. There are different versions for most popular operating systems.

It requires unzipping into a subfolder and putting the Juno executable on the run-search path, so it is one of the easiest ways to get started on a variety of platforms. It uses Light Table, so unlike IJulia (explained in the following section), it does not need a helper task (viz. Python) to be present.

The driver is the `Jewel.jl` package, which is a collection of IDE-related code and is responsible for communication with Light Table. The IDE has a built-in workspace and navigator. Opening a folder in the workspace will display all the files via the navigator.

Juno handles things such as the following:

- Extensible autocomplete
- Pulling code blocks out of files at a given cursor position
- Finding relevant documentation or method definitions at the cursor
- Detecting the module to which a file belongs
- Evaluation of code blocks with the correct file, line, and module data

Juno's basic job is to transform expressions into values, which it does on pressing `Ctrl + Enter`, (`Cmd + Enter` on Mac OS X) inside a block. The code block is evaluated as a whole, rather than line by line, and the final result returned.

By default, the result is "collapsed." It is necessary to click on the bold text to toggle the content of the result. Graphs, from Winston and Gadfly, say, are displayed in line within Juno, not as a separate window.

IJulia

IJulia is a backend interface to the Julia language, which uses the IPython interactive environment. It is now part of Jupyter, a project to port the agnostic parts of IPython for use with other programming languages.

This combination allows you to interact with the Julia language by using IPython's powerful graphical notebook, which combines code, formatted text, math support, and multimedia in a single document.

You need version 1.0 or later of IPython. Note that IPython 1.0 was released in August 2013, so the version of Python required is 2.7 and the version pre-packaged with operating-system distribution may be too old to run it. If so, you may have to install IPython manually.

On Mac OS X and Windows systems, the easiest way is to use the Anaconda Python installer. After installing Anaconda, use the `conda` command to install IPython:

```
conda update conda conda update ipython
```

On Ubuntu, we use the `apt-get` command and it's a good idea to install `matplotlib` (for graphics) plus a cocktail of other useful modules.

```
sudo apt-get install python-matplotlib python-scipy python-pandas python-sympy python-nose
```

IPython is available on Fedora (v18+) but not yet on CentOS (v6.5) although this should be resolved with CentOS 7. Installation is via yum as follows:

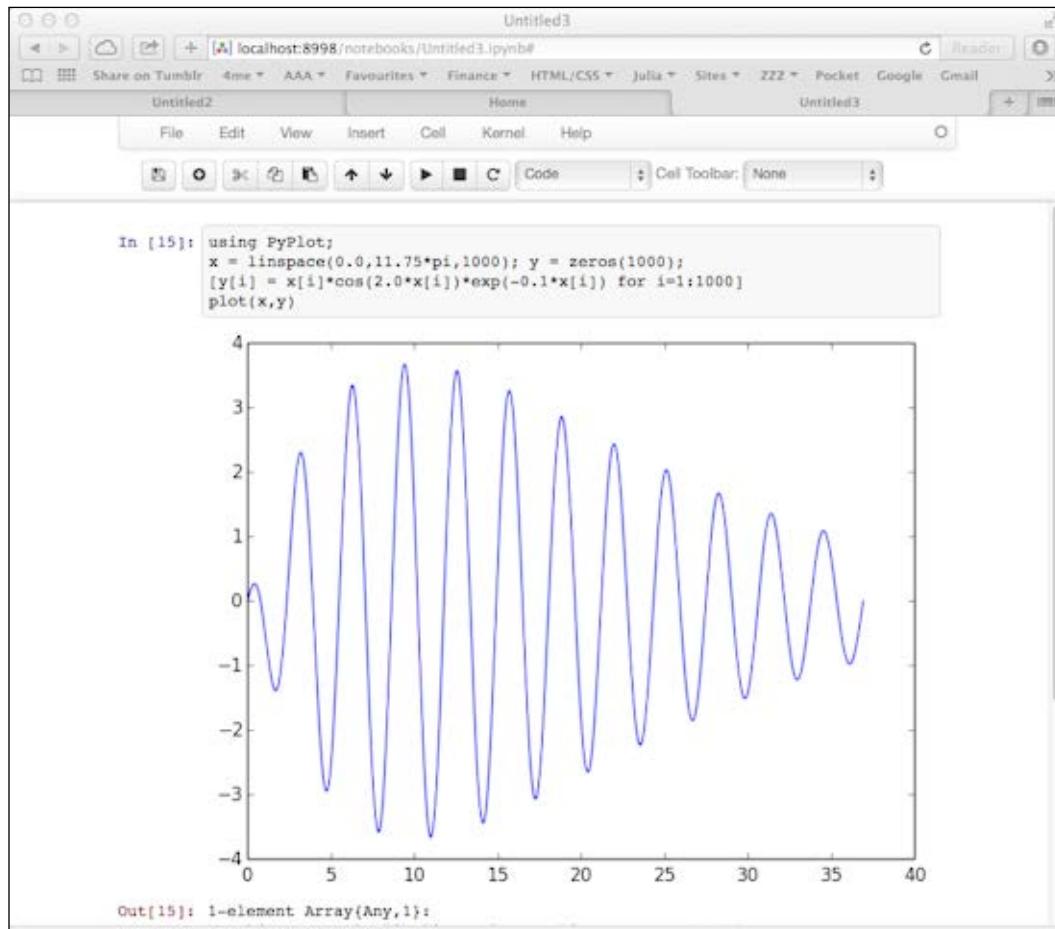
```
sudo yum install python-matplotlib scipy python-pandas sympy python-nose
```

The IPython notebook interface runs in your web browser and provides a rich multimedia environment. Furthermore, it is possible to produce some graphic output via Python's `matplotlib` by using a Julia to Python interface. This requires installation of the IJulia package.

Start IJulia from the command line by typing `ipython notebook --profile julia`, which opens a window in your browser.

This can be used as a console interface to Julia; using the `PyPlot` package is also a convenient way to plot some curves.

The following screenshot displays a damped sinusoid of the form $x * \exp(-0.1x) * \cos(2.0x)$:



A quick look at some Julia

To get flavor, look at an example that uses random numbers to price an Asian derivative on the options market.

A share option is the right to purchase a specific stock at a nominated price sometime in the future. The person granting the option is called the grantor and the person who has the benefit of the option is the beneficiary. At the time the option matures, the beneficiary may choose to exercise the option if it is in his/her interest and the grantor is then obliged to complete the contract.

In order to set up the contract, the beneficiary must pay an agreed fee to the grantor. The beneficiary's liability is therefore limited by this fee, while the grantor's liability is unlimited. The following question arises: How can we arrive at a price that is fair to both the grantor and the beneficiary? The price will be dependent on a number of factors such as the price that the beneficiary wishes to pay, the time to exercise the option, the rate of inflation, and the volatility of the stock.

Options characteristically exist in one of two forms: call options and put options.

Call options, which give the beneficiary the right to require the grantor to sell the stock to him/her at the agreed price upon exercise, and put options, which give the beneficiary the right to require the grantor to buy the stock at the agreed price on exercise. The problem of the determination of option price was largely solved in the 1970s by Fisher Black and Myron Scholes by producing a formula for the price after treating the stock movement as random (Brownian) and making a number of simplifying assumptions.

We are going to look at the example of an Asian option, which is one for which there can be no formula. This is a type of option (sometimes termed an average value option) where the payoff is determined by the average underlying price over some preset period of time up to exercise rather than just the final price at that time.

So, to solve this type of problem, we must simulate the possible paths (often called random walks) for the stock by generating these paths using random numbers. We have seen a simple use of random numbers earlier while estimating the value of Pi. Our problem is that the accuracy of the result typically depends on the square of the number of trials, so obtaining an extra significant figure needs a hundred times more work. For our example, we are going to do 100000 simulations, each 100 steps representing a daily movement over a period of around 3 months. For each simulation, we determine at the end whether based on the average price of the stock, there would be a positive increase for a call option or a negative one for a put option. In which case, we are "in the money" and would exercise the option. By averaging all the cases where there is a gain, we can arrive at a fair price.

The code that we need to do this is relatively short and needs no special features other than simple coding.

Julia via the console

We can create a file called `asian.jl` by using the following code:

```
function run_asian(N = 100000, PutCall = 'C';)
# European Asian option.
# Uses geometric or arithmetic average.
```

```

# Euler and Milstein discretization for Black-Scholes.

# Option features.

    println("Setting option parameters");
    S0 = 100;          # Spot price
    K = 100;           # Strike price
    r = 0.05;          # Risk free rate
    q = 0.0;           # Dividend yield
    v = 0.2;           # Volatility
    tma = 0.25;        # Time to maturity

    Averaging = 'A';   # 'A'ithmetic or 'G'eometric
    OptType = (PutCall == 'C' ? "CALL" : "PUT");
    println("Option type is $OptType");

# Simulation settings.

    println("Setting simulation parameters");
    T = 100;           # Number of time steps
    dt = tma/T;        # Time increment

# Initialize the terminal stock price matrices
# for the Euler and Milstein discretization schemes.

S = zeros(Float64,N,T);
for n=1:N
    S[n,1] = S0;
end

# Simulate the stock price under the Euler and Milstein schemes.

# Take average of terminal stock price.

    println("Looping $N times.");
    A = zeros(Float64,N);
    for n=1:N
        for t=2:T
            dW = (randn(1)[1])*sqrt(dt);
            z0 = (r - q - 0.5*v*v)*S[n,t-1]*dt;
            z1 = v*S[n,t-1]*dW;
            z2 = 0.5*v*v*S[n,t-1]*dW*dW;
            S[n,t] = S[n,t-1] + z0 + z1 + z2;
        end
    end

```

```
    end

    if cmp(Averaging,'A') == 0
        A[n] = mean(S[n,:]);
    elseif cmp(Averaging,'G') == 0
        A[n] = exp(mean(log(S[n,:])));
    end
end

# Define the payoff
P = zeros(Float64,N);
if cmp(PutCall,'C') == 0
    for n = 1:N
        P[n] = max(A[n] - K, 0);
    end
elseif cmp(PutCall,'P') == 0
    for n = 1:N
        P[n] = max(K - A[n], 0);
    end
end

# Calculate the price of the Asian option
AsianPrice = exp(-r*tma)*mean(P);
@printf "Price: %10.4f\n" AsianPrice;
end
```

We have wrapped the main body of the code in a function `run_asian(N, PutCall)` `end` statement. The reason for this is to be able to time the execution of the task in Julia, thereby eliminating the startup times associated with the Julia runtime when using the console.

The stochastic behavior of the stock is modeled by the `randn` function; `randn(N)` provides an array of N elements, normally distributed with zero mean and unit variance.

All the work is done in the inner loop; the `z`-variables are just written to decompose the calculation.

To store the averages for each track, use the `zeros` function to allocate and initialise the array.

The option would only be exercised if the average value of the stock is above the "agreed" prices. This is called the payoff and is stored for each run in the array `P`.

It is possible to use arithmetic or geometric averaging. The code sets this as arithmetic, but it could be parameterized.

The final price is set by applying the mean function to the `P` array. This is an example of vectorized coding.

So, to run this simulation, start the Julia console and load the script as follows:

```
julia> include("asian.jl")
julia> run_asian()
```

To get an estimate of the time taken to execute this command, we can use the `tic()`/`toc()` function or the `@elapsed` macro:

```
include("asian.jl")
tic(); run_asian(1000000, 'C'); toc();
Option Price: 1.6788 elapsed: time 1.277005471 seconds
```

If we are not interested in the execution times, there are a couple of ways in which we can proceed.

The first is just to append to the code a single line calling the function as follows:

```
run_asian(1000000, 'C')
```

Then, we can run the Asian option from the command prompt by simply typing the following: `julia asian.jl`.

This is pretty inflexible since we would like to pass different values of the number of trials `N` and to determine the price for either a call option or a put option.

Julia provides an `ARGS` array when a script is started from the command line to hold the passed arguments. So, we add the following code at the end of `asian.jl`:

```
nArgs = length(ARGS)
if nArgs >= 2
    run_asian(ARGS[1], ARGS[2])
elseif nArgs == 1
    run_asian(ARGS[1])
else
    run_asian()
end
```

Julia variables are case-sensitive, so we must use `ARGS` (uppercase) to pass the arguments.

Because we have specified the default values in the function definition, this will run from the command line or if loaded into the console.

Arguments to Julia are passed as strings but will be converted automatically although we are not doing any checking on what is passed for the number of trials (N) or the `PutCall` option.

Installing some packages

We will discuss the package system in the next section, but to get going, let's look at a simple example that produces a plot by using the `ASCIIPlots` package. This is not dependent on any other package and works at the console prompt.

You can find it at <http://docs.julialang.org> and choose the "Available Packages" link. Then, click on **ASCIIPlots**, which will take you to the GitHub repository.

It is always a good idea when looking at a package to read the markdown file: `README.md` for examples and guidance.

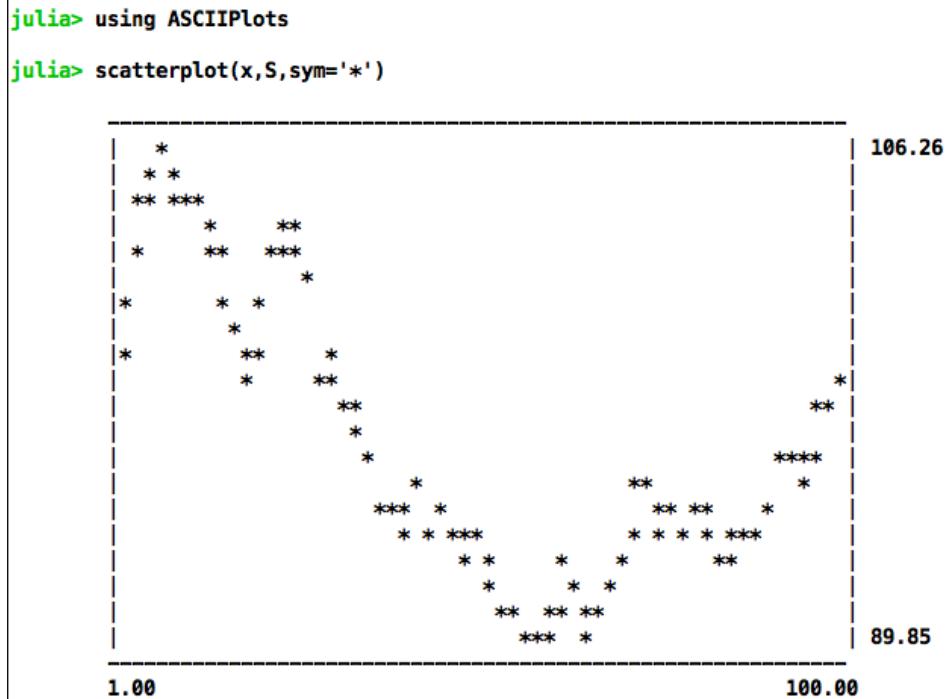
Let's use the same parameters as before. We will be doing a single walk so there is no need for an outer loop or for accumulating the price estimates and averaging them to produce an option price.

By compacting the inner loop, we can write this as follows:

```
using ASCIIPlots;

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   = 100;       # Number of time steps
dt  = tma/T;    # Time increment
S   = zeros(Float64,T);
S[1] = S0;
```

```
dW = randn(T)*sqrt(dt)
[ S[t] = S[t-1] * (1 + (r - q - 0.5*v*v)*dt + v*dW[t] +
0.5*v*v*dW[t]*dW[t]) for t=2:T ]
x = linspace(1,T);
scatterplot(x,S,sym='*' );
```



Note that when adding a package the statement using its name as an `ASCIIPlots` string, whereas when using the package, it does not.

A bit of graphics creating more realistic graphics with Winston

To produce a more realistic plot, we turn to another package called Winston.

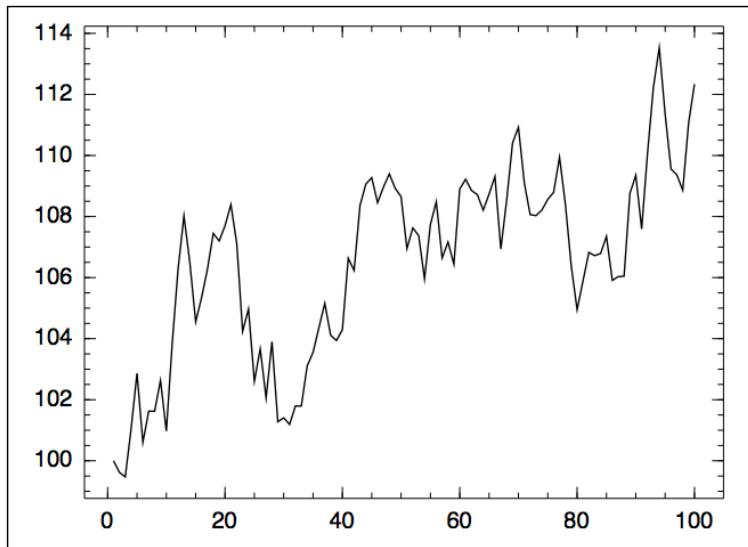
Add this by typing `Pkg.add("Winston")`, which also adds a number of other "required" packages. By condensing the earlier code (for a single pass), it reduces the earlier code to the following:

```
using Winston;

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   = 100;          # Number of time steps
dt  = tma/T;        # Time increment
S   = zeros(Float64,T)
S[1] = S0;
dW = randn(T)*sqrt(dt);
[ S[t] = S[t-1] * (1 + (r - q - 0.5*v*v)*dt + v*dW[t] +
0.5*v*v*dW[t]*dW[t]) for t=2:T ]

x = linspace(1, T, length(T));
p = FramedPlot(title = "Random Walk, drift 5%, volatility 2%")
add(p, Curve(x,S,color="red"))
display(p)
```

1. Plot one track, so only compute vector s of T elements.
2. Compute stochastic variance dw in a single vectorized statement.
3. Compute track s by using "list comprehension."
4. Create array x by using `linspace` to define a linear abscissa for the plot.
5. Use the `Winston` package to produce the display, which only requires three statements: to define the plot space, add a curve to it, and display the plot as shown in the following figure:



My benchmarks

We compared the Asian option code above with similar implementations in the "usual" data science languages discussed earlier.

The point of these benchmarks is to compare the performance of specific algorithms for each language implementation. The code used is available to download.

Language	Timing (c = 1)	Asian option
C	1.0	1.681
Julia	1.41	1.680
Python (v3)	32.67	1.671
R	154.3	1.646
Octave	789.3	1.632

The runs were executed on a Samsung RV711 laptop with an i5 processor and 4GB RAM running CentOS 6.5 (Final).

Package management

We have noted that Julia uses Git as a repository for itself and for its package and that the installation has a built-in package manager, so there is no need to interface directly to GitHub. This repository is located in the Git folder of the installed system.

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

Listing, adding, and removing

After installing Julia, the user can create a new repository by using the `Pkg.init()` command. This clones the metadata from a "well-known" repository and creates a local folder called `.julia`:

```
julia> Pkg.init()
INFO: Initializing package repository C:\Users\Malcolm
INFO: Cloning METADATA from git://Github.com/JuliaLang
```

The latest versions of all installed packages can be updated with the `Pkg.update()` command.

Notice that if the repository does not exist, the first use of a package command such as `Pkg.update()` or `Pkg.add()` will call `Pkg.init()` to create it:

```
julia> Pkg.update()
Pkg.update()
INFO: Updating METADATA...
INFO: Computing changes...
INFO: No packages to install, update or remove.
```

We previously discussed how to install the `ASCIIPlots` package by using the `Pkg.add("ASCIIPlots")` command. The `Pkg.status()` command can be used to show the current packages installed and `Pkg.rm()` to remove them:

```
julia> Pkg.status()
Pkg.status()
Required packages:
- ASCIIPlots 0.0.2

julia> Pkg.rm("ASCIIPlots")
Pkg.rm("ASCIIPlots")
INFO: Removing ASCIIPlots INFO: REQUIRE updated.
```

After adding `ASCIIPlots`, we added the `Winston` graphics package. Most packages have a set of others on which they depend and the list can be found in the `REQUIRE` file.

For instance, `Winston` requires the `Cairo`, `Color`, `IniFile`, and `Tk` packages. Some of these packages also have dependencies, so `Pkg.add()` will recursively resolve, clone, and install all of these. The `Cairo` package is interesting since it requires **Homebrew** on Mac OS X and **WinRPM** on Windows.

`WinRPM` further needs `URLParse`, `HTTPClient`, `LibExpat`, and `ZLib`. So, if we use `Pkg.status()` again on a Windows installation, we get the following:

```
julia> Pkg.status()
Required packages:
- ASCIIPlots          0.0.2
- Winston              0.11.0

Additional packages:
```

- BinDeps	0.2.14
- Cairo	0.2.13
- Color	0.2.10
- HTTPClient	0.1.0
- IniFile	0.2.2
- LibCURL	0.1.3
- LibExpat	0.0.4
- Tk	0.2.12
- URIParser	0.0.2
- URLParse	0.0.0
- WinRPM	0.0.13
- Zlib	0.1.5

All the packages installed as dependencies are listed as additional packages. Removing the `Winston` package will also remove all the additional packages. When adding complex packages, you may wish to add some of the dependent ones first. So, with `Winston`, you can add both `Cairo` and `Tk`, which will then show the required rather than the additional packages.

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 the mathematical analyst. Registered packages are available on GitHub, and the list of these packages can be referenced via <http://docs.julialang.org/>.

Because the core language is still under review from release to release, some features are deprecated, others changed, and the 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. Furthermore, it may be that a package may not work under different operating systems. In general, use under the Linux operating system fares the best and under Windows fares the worst.

How then should we select a package? The best indicators are the version number; packages designated v0.0.0 should always be viewed with some suspicion. Furthermore, the date of the last update is useful here. The docs website also lists the individual contributors to each individual package with the principal author listed first. Ones with multiple developers are clearly of interest to a variety of contributors and tend to be better discussed and maintained. There is strength here in numbers. The winner in this respect seems to be (as of July 2014) the `DataFrames` package, which is up to version 0.3.15 and has attracted the attention of 33 separate authors.

Even with an old relatively untouched package, there is nothing to stop you 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. Furthermore, the principal author is likely to be delighted that someone else is finding the package useful and taking an interest in the work.

It is not possible to create a specific taxonomy of Julia packages but certain groupings emerge, which build on the backs of the earlier ones. We will be meeting many of these later in this book, but before that, it may be useful to quickly list a few.

Statistics and mathematics

Statistics is seen rightly 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 add that of SciPy and Pandas.

Statistics is well provided in Julia on GitHub by both the <https://github.com/JuliaStats> group and a Google group called <https://groups.google.com/forum/#!forum/julia-stats>.

Much of the basic statistics is provided by `Stats.jl` and `StatsBase.jl`. There are various means of working with R-style data frames and loading some of the datasets available to R. The distributions package covers the probability distributions and the associated functions. Moreover, there is support for time series, cluster analysis, hypothesis testing, MCMC methods, and more.

Mathematical operations such as random number generators and exotic functions 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 <https://github.com/JuliaOpt> group, which lists the packages under the umbrella of optimization.

Data visualization

Graphics 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 graphics support in the core, but at present, this is largely the realm of package developers.

While it was true that v0.1.x offered very limited and flaky graphics, v0.2.x vastly improved the situation and this continues with v0.3.x.

Firstly, there is a module in the core called `Base.Graphics`, which acts as an abstract layer to packages such as `Cairo` and `Tk/Gtk`, which serve to implement much of the required functionality.

Layered on top of these are a couple of packages, namely `Winston` (which we have introduced already) and `Gadfly`. Normally, as a user, you will probably work with one or the other of these.

`Winston` 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.

`Gadfly` is a system for plotting and visualization equivalent to the `ggplot2` module in R. It can be used to render the graphics output to PNG, PostScript, PDF, and SVG files. `Gadfly` works best with the following C libraries installed: `cairo`, `pango`, and `fontconfig`. The PNG, PS, and PDF backends all require `cairo`, but without it, it is still possible to create displays to SVG and Javascript/D3.

There are a couple of different approaches, which are worthy of note: `Gaston` and `PyPlot`.

`Gaston` is an interface to the `gnuplot` program on Linux. You need to check whether `gnuplot` is available, and if not, it must be installed in the usual way via `yum` or `apt-get`. For this, you need to install `XQuartz`, which must be started separately before using `Gaston`.

`Gaston` can do whatever `gnuplot` is capable of. There is a very comprehensive script available in the package by running `Gaston.demo()`.

We have discussed `Pyplot` briefly before when looking at IJulia. The package uses Julia's `PyCall` package to call the `Matplotlib` Python module directly and can display plots in any Julia graphical backend, including as we have seen, inline graphics in IJulia.

Web and networking

Distributed computing is well represented in Julia. TCP/IP sockets are implemented in the core. Additionally, there is support for Curl, SMTP and for WebSockets. HTTP protocols and parsing are provided for with a number of packages, such as `HTTP`, `HttpParser`, `HttpServer`, `JSON`, and `Mustache`.

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

Database and specialist packages

The database is supported mainly through the use of the ODBC package. On Windows, ODBC is the standard, while Linux and Mac OS X require the installation of unixODBC or iODBC. There is currently no native support for the main SQL databases such as Oracle, MySQL, and PostgreSQL.

The package SQLite provides an interface to that database and there is a Mongo package, which 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.

A couple of specialist Julia groups are JuliaQuant and JuliaGPU.

JuliaQuant encompasses a variety of packages for quantitative financial modeling. This is an area that has been heavily supported by developers in R, MATLAB, and Python, and the Quant group is addressing the same problems in Julia.

JuliaGPU is a set of packages supporting OpenCL and CUDA interfacing to GPU cards for high-speed parallel processing.

Both of these are very much works in progress, and interest and support in the development of the packages would be welcome.

How to uninstall Julia

Removing Julia is very simple; there is no explicit uninstallation process. It consists of deleting the source tree, which was created by the build process or from the DMG file on Mac OS X or the EXE file on Windows. Everything runs within this tree, so there are no files installed to any system folders.

In addition, we need to attend to the package folder. Recall that under Linux and Mac OS X this is a hidden folder called `.julia` in the user's home folder. In Windows, it is located in the user's profile typically in `C:\Users\[my-user-name]`. Removing this folder will erase all the packages that were previously installed.

There is another hidden file called `.julia_history` that should be deleted; it keeps an historical track of the commands listed.

Adding an unregistered package

The official repository for the registered packages in Julia is here:

<https://Github.com/JuliaLang/METADATA.jl>.

Any packages here will be listed using the package manager or in Julia Studio.

However, it is possible to use an unregistered package by using `Pkg.clone(url)`, where the `url` is a Git URL from which the package can be cloned. The package should have the `src` and `test` folders and may have several others. If it contains a `REQUIRE` file at the top of the source tree, that file can be used to determine any dependent registered packages; these packages will be automatically installed.

If you are developing a package, it is possible to place the source in the `.julia` folder alongside packages added with `Pkg.add()` or `Pkg.clone()`. Eventually, you will wish to use GitHub in a more formal way; we will deal with that later when considering package implementation.

What makes Julia special

Julia, as a programming language, has made rapid progress since its launch in 2012, which is a testimony to the soundness and quality of its design and coding. It is true that Julia is fast, but speed alone is not sufficient to guarantee a progression to a main stream language rather than a cult language.

Indeed, the last few years have witnessed the decline of Perl (largely self-inflicted) and a rapid rise in the popularity of R and Python. This we have attributed to a new breed of analysts and researchers who are moving into the fields of data science and big data and who are looking for tool kits that fulfill their requirements.

To occupy this space, Julia needs to offer some features that others find hard to achieve. Some such features are listed as follows:

Parallel processing

As a language aimed at the scientific community, it is natural that Julia should provide facilities for executing code in parallel. In running tasks on multiple processors, Julia takes a different approach to the popular **message passing interface (MPI)**. In Julia, communication is one-sided and appears to the programmer more as a function call than the traditional message send and receive paradigm typified by **pyMPI** on Python and **Rmpi** on R.

Julia provides two in-built primitives: remote references and remote calls. A remote reference is an object that can be used from any processor to refer to an object stored on a particular processor. A remote call is a request by one processor to call a certain function or certain arguments on another, or possibly the same, processor.

Sending messages and moving data constitute most of the overhead in a parallel program, and reducing the number of messages and the amount of data sent is critical to achieving performance and scalability. We will be investigating how Julia tackles this in a subsequent chapter.

Multiple dispatch

The choice of which method to execute when a function is applied is called dispatch.

Single dispatch polymorphism is a familiar feature in object-orientated languages where a method call is dynamically executed on the basis of the actual derived type of the object on which the method has been called.

Multiple dispatch is an extension of this paradigm where dispatch occurs by using all of a function's arguments rather than just the first.

Homoiconic macros

Julia, like Lisp, represents its own code in memory by using a user-accessible data structure, thereby allowing programmers to both manipulate and generate code that the system can evaluate. This makes complex code generation and transformation far simpler than in systems without this feature.

We met an example of a macro earlier in `@printf`, which mimics the C-like `printf` statements. Its definition is given in the `base/printf.jl` file.

Interlanguage cooperation

We noted that Julia is often seen as a competitor to languages such as C, Python, and R, but this is not the view of the language designers and developers.

Julia makes it simple to call C and Fortran functions, which are compiled and saved as shared libraries. This is by the use of the in-built `call`. This means that there is no need for the traditional "wrapper code" approach, which acts on the function inputs, transforms them into an appropriate form, loads the shared library, makes the call, and then repeats the process in reverse on the return value. Julia's JIT compilation generates the same low-level machine instructions as a native C call, so there is no additional overhead in making the function call from Julia.

Additionally, we have seen that the PyCall package makes it easy to import Python libraries, and this has been seen to be an effective method of displaying the graphics from Python's `matplotlib`. Further, inter-cooperation with Python is evident in the provision of the IJulia IDE and an adjunction to IPython notebooks.

There is also work on calling R libraries from Julia by using the `Rif` package and calling Java programs from within Julia by using the `JavaCall` package. These packages present the exciting prospect of opening up Julia to a wealth of existing functionalities in a straightforward and elegant fashion.

Summary

This chapter introduced you to Julia, how to download it, install it, and build it from source. We saw that the language is elegant, concise, and powerful. The next three 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. We also discussed the use of two interactive development environments (IDEs), Juno and IJulia, as an alternative to REPL.

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

2

Developing in Julia

Julia is a feature-rich language. It was designed to appeal to the novice programmer and purist alike. Indeed 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 the reader 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 two 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.

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.

At the present time Julia has not yet reached the version 1.0 status and some of the syntax may be deprecated and later changed. However, we believe that most of the material presented here will stand the test of time.

Integers, bits, bytes, and bools

Julia is a strongly typed language allowing the programmer to specify a variable's type precisely. However in common with most interpreted languages it does not require the type to be declared when a variable is declared, rather it infers it from the form of the declaration.

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.

So typically:

```
julia> x = 2;    typeof(x) # => gives Int64
julia> x = 2.0;  typeof(x) # => gives Float64
```

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 built-in constant `WORD_SIZE`.

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

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

Integers

The integer type can be any of `Int8`, `Int16`, `Int32`, `Int64` and `Int128`, so the maximum integer can occupy 16 bytes of storage and can be anywhere within the range of -2^{127} to $(+2^{127} - 1)$.

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

```
julia> x=BigInt(2^32); x^6
6277101735386680763835789423207666416102355444464034512896
```

There are a few more things to say about integers:

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` is $(2^{128} - 1)$.

We can use the `typemax()` and `typemin()` functions to output the ranges of the `Int` and `Uint` types.

```
julia> for T = {Int8,Int16,Int32,Int64,Int128,Uint8,Uint16,Uint32,Uint64,
Uint128}println("#(lpad(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:

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

The reason is that integer overflow 'wraps' around, so squaring 2^{32} gives 0 not 2^{64} , since my WORD_SIZE 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` to see what its parent type is.

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

The definition of `DataType` is 'hinted' at in the core file `boot.jl`; hinted at because the actual definition is implemented in C and the Julia equivalent is commented out.

The definitions of the integer types can also be found in `boot.jl`, this time not commented out. Typically:

```
abstract Number
abstract Real <: Number
abstract Integer <: Real
abstract Signed <: Integer
bitstype 64 Int64 <: Signed
```

Where the `<:` operator corresponds to a subclass of the parent and `bitstype` again is defined in the core, nominally as `#bitstype {32|64} Ptr{T}`.

If we type:

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

So division of two integers produces a real result. In interactive mode we can use the symbol `ans` to correspond to the last answer, that is, `typeof(ans)` gives `Float64`.

To get the integer divisor we use the function `div(x,y)` 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).

Logical and arithmetic operators

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

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 `UInt16`.

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

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

Also we can perform arithmetic shifts using `<<` (LEFT) and `>>` (RIGHT) operators. Note because `x` is of type `UInt16` the shift operator retains that size, so:

```
julia> x = 0xbb31; x<<8
```

This gives `0x3100` (the top two nibbles being discarded) and `typeof(ans)` is `UInt16`.

Arithmetic shifts preserve the sign bit. This is not relevant when dealing with unsigned integers but bit-wise operators can be applied to signed integers too. Logical and arithmetic left shifts produce the same result but right shifts do not. In this case Julia provides the `>>>` operator to apply a right logical shift:

```
julia> x = 0xbb31; y = int16(x) # => 17615
```

`y>>4` gives -1101 and `y>>>4` gives 2995.

Booleans

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

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

Many languages treat 0, empty strings, NULLS 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 type `Int` and `p` of type `Bool`) will:

```
julia> x = 0xbb31; p = (2 < 3); x + p # => 0x000000000000bb32
julia>typeof(ans) # => UInt(64)
```

Arrays

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

One simple way to create an array is to enumerate its values:

```
julia> A = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377];
15-element Array{Int64,1}
```

These are the first 15 values of the Fibonacci series and because all values are listed as integers the array created is of type `Int64`. The other number refers to the number of dimensions of the array, in this case 1.

In conjunction of 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]; B = [1:3:15]; C =[1:0.5:5];
```

Here `A` is `[1,2,3,4,5,6,7,8,9,10]`, `B` is `[1,4,7,10,13]` and `C` is `[1.0,1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0]`

Because the step in `C` is specified as a float value the array is of type `Float64` not `Int64`.

Julia also provides functions such as `zeros`, `ones` and `rand` which provide array results. Normally these are returned a float-point value so a little bit of work is required to provide integer results.

```
julia> A = int(zeros(15)); B = int(ones(15)); C = rand(1:100,15);
```

Another method of creating and populating an array is by using a list comprehension. If we recall the first example of the Fibonacci series, we can get the same result by creating an uninitialized array of 15 integers by using `Array(Int64, 15)` then by assigning the first couple of values and use the definition of the Fibonacci series to create the remaining values:

```
julia> A = Array(Int64,15); A[1]=0;A[2]=1; [A[i] = A[i-1] + A[i-2] for i = 3:length(A)]
```

Finally it is possible to create a completely empty array by using `Int64 []`. Since array sizes are fixed this would seem a little futile but certain functions can be used to alter the size of the array. In particular the `push! ()` function can add a value to the array and increase its length by one:

```
julia> A = Int64[];  
julia> push!(A,1); push!(A,2); push!(A,3) # => [1,2,3]
```

The corresponding `pop! (A)` function will return the value 3, and result in `A = [1, 2]`.

Note that the use of the tailing ! borrows the syntax form Lisp-like conventions and is purely arbitrary. Since functions are first class variables this is the reason that ! is an accepted character in variable names but it's a good idea to respect the convention and not use ! in reference to common variables.

Consider the following two array definitions:

```
julia> A = [1,2,3];  
3-element Array{Int64,1}  
julia> A = [1 2 3];  
1x3 Array{Int64,2}
```

The first, with values separated by commas, produces the usual one-dimensional data structure. The second, where there are no commas, produces a matrix or 1 row and 3 columns, hence the definition: `1x3 Array{Int64,2}`.

To define more rows we separate the values with semicolons as:

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

If we type:

```
for i in (1:length(A)) @printf("%ld: %ld\n", i, A[i]); end  
1 : 1  
2 : 4  
3 : 2  
4 : 5  
5 : 3  
6 : 6
```

In Julia indexing is in column order and the array/matrix can be indexed in a one-dimensional or two-dimensional.

```
julia> A[1,2] # => 2  
julia> A[2] # => 4  
julia> A[5]# => 3
```

In fact it is possible to reshape the array to change it from a 2×3 matrix to a 3×2 one:

```
julia> B = reshape(A,3,2)  
3x2 Array{Int64,2}:  
1 5  
4 3  
2 6
```

Operations on matrices

We will be meeting matrices and matrix operations through this book but let us look at the simplest of operations:

Taking A and B as defined previously the normal matrix rules apply.

We'll define C as the transpose of B so:

```
julia> C = transpose(B)  
2x3 Array{Int64,2}:  
1 4 2  
5 3 6  
julia> A + C  
2x3 Array{Int64,2}:
```

```
2   6   5
9   8   12
julia> A*B
2x2 Array{Int64,2}:
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:

```
julia> A / C
2x2 Array{Float64,2}
0.332273  0.27663
0.732909  0.710652
```

Elemental operations

As well as the 'common' matrix operations, Julia defines a set which works on the elements of the matrix. So although $A * C$ is not allowed because number of columns of A is not equal to number of rows of C but the following are all valid since A and C are the same shape:

```
julia> A .* C
2x3 Array{Int64,2}:
1   8   6
20  15  36

julia> A ./ C
2x3 Array{Float64,2}:
1.0  0.5      1.5
0.8  1.66667  1.0

julia> A .== C
2x3 BitArray{2}:
true  false  false
false false  true
```

A simple Markov chain – cat and mouse

Suppose there is a row of five adjacent boxes, with a cat in the first box and a mouse in the fifth box. At each 'tick' the cat and the mouse both jump to a random box next to them. On the first tick the cat must jump to box 2 and the mouse to box 4 but on the next ticks they may jump to the box they started in or to box 3.

When the cat and mouse are in the same box the cat catches the mouse and the Markov chain terminates. Because there is an odd number of boxes between the cat and mouse it's easy to see that they will not jump past each other.

So Markov chain that corresponds to this contains the only five possible combinations of (Cat, Mouse).

```
State 1: (1,3) State 2: (1,5) State 3: (2,4) State 4: (3,5) State 5:  
(2,2), (3,3) & (4,4) # => cat catches the mouse
```

The matrix $P = [0\ 0\ .5\ 0\ .5; 0\ 0\ 1\ 0\ 0; .25\ .25\ 0\ .25\ .25; 0\ 0\ .5\\ 0\ .5; 0\ 0\ 0\ 0\ 1]$ represents the probabilities of the transition from one state to the next and the question is how long has the mouse got before it's caught. Its best chance is starting in State 2 = (1,5).

The matrix P is a stochastic matrix where all the probabilities along any row add up to 1.

This is actually an easy problem to solve using some matrix algebra in a few lines in Julia and for a full discussion of the problem look at the Wikipedia discussion.

```
I = eye(4);  
P = [0 0 .5 0; 0 0 1 0; .25 .25 0 .25; 0 0 .5 0];  
ep = [0 1 0 0]*inv(I - P)*[1,1,1,1];  
println("Expected lifetime for the mouse is $ep ticks")  
Expected lifetime for the mouse is [4.5] ticks
```

Consider how this works:

1. The `eye()` function returns a square(real) matrix with leading diagonal unity and the other values of zero.
2. The matrix P can be reduced to 4×4 since when in state 5 the Markov chain terminates.
3. The `inv(I - P) * [1,1,1,1]` returns the expected lifetime (no disrespect) of the mouse in all states so multiplying with `[0 1 0 0]` gives the expectation when starting in state 2.0.

Char and strings

So far we have been dealing with numeric and boolean datatypes. In this section we will look at character representation and how Julia handles ASCII and UTF-8 strings of characters. We will also introduce the concept of regular expressions, widely used in pattern matching and filtering operations.

Characters

Julia has a built-in type `Char` to represent a character. A character occupies 32 bits not 8, so a character can represent a UTF-8 symbol and may be assigned in a number of ways:

```
julia> c = 'A'  
julia> c = char(65)  
julia> c = '\u0041'
```

All these represent the ASCII character capital A.

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_char()` function:

```
julia> c = '\Udff3';  
julia> isValid_char(c; ) # => gives false.
```

Julia uses the special C-like syntax for certain ASCII control characters such as '`\b`', '`\t`', '`\n`', '`\r`', '`\f`' for backspace, tab, newline, carriage return and form feed. Otherwise the backslash acts as an escape character, so `int('\'s')` gives 115 whereas `int('\'t')` gives 9.

Strings

The type of string we are most familiar with comprises a list of ASCII characters which, in Julia, are normally delimited with double quotes, that is:

```
julia> s = "Hello there, Blue Eyes"; typeof(s)  
ASCIIString (constructor with 2 methods)
```

In fact a string is an abstraction not a concrete type and `ASCIIString` is only one such abstraction. Looking at `Base::boot.jl` we see:

```
abstract String  
abstract DirectIndexString <: String  
immutable ASCIIString <: DirectIndexString
```

```
    data::Array{UInt8,1}
end
immutable UTF8String <: String
    data::Array{UInt8,1}
end
typealias ByteString Union(ASCIIString,UTF8String)
```

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.

ASCII strings are also indexable so from `s` as defined previously: `s[14:17]` gives "Blue". The values in the range are inclusive and if we wish we can change the increment as `s[14:2:17]` which gives "Bu" or reverse the slice as `s[17:-1:14]` which gives "eulB". Omitting the end of the range is equivalent to running to the end of the string: `s[14:]` which gives "Blue Eyes".

However `s[:14]` is somewhat unexpected and gives the character `B` not the string upto and including `B`. This is because the `:` 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]`.

Strings allow for the special characters such as '`\n`', '`\t`', and so on. If we wish to include the double quote we can escape it 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 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 also Julia uses the `*` operator (rather than `+` or some other symbol):

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

The following two expressions are directly equivalent:

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

Unicode support

We saw from the definition above that apart from ASCII strings Julia defines UTF-8 strings. In fact UTF-8 is not all that Julia supports and adding support for new encodings is quite easy. In particular, Julia also provides `UTF16String` and `UTF32String` types, constructed by the `utf16(s)` and `utf32(s)` functions respectively, for UTF-16 and UTF-32 encodings.

Julia provides a function `endof()` which can be used to determine the end of a string and a symbol `end` to denote the character in the last index position.

Because of variable-length encodings, the number of characters in a string which is given by `length(s)` is not always the same as the last index. If you iterate through the indices 1 through `endof(s)` and index into `s`, the sequence of characters returned when errors aren't thrown is the sequence of characters comprising the string. Thus we have the identity that `length(s) <= endof(s)`, since each character in a string must have its own index:

```
julia> s = "\u2200 x \u2203 y"    # this is the mathematical expression
.....
julia>typeof(s) # => UTF8String
julia>endof(s) # => 11
julia> length(s) # => 7
julia> s[end]    # => 'y'
```

In this case, since the string `s` has two UTF characters each occupying 3 bytes the only valid indices are `s[1], s[4], s[5], s[6], s[7], s[11]`, so for example `s[7]` will return the character '`\u2203`'.

Regular expressions

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

There is an old adage: "*I had a problem and decided to solve it using regular expressions, now I have two problems*".

Regular expressions are used for pattern matching, numerous books have been written on them and support is available in a variety of our programming languages post-Perl, notably Java and Python.

Julia supports regular expressions via a special form of string prefixed with an `r`.

Suppose we define the pattern `empat` as:

```
empat = r"^[_a-zA-Z-]+(\.[_a-zA-Z-]+)*@[a-zA-Z-]+(\. [a-zA-Z-]+)*(\. [a-zA-Z]{2,4})$"
```

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

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

The pattern is for a valid e-mail address and in the second case the space in "Fredrick Flintstone" is not valid 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 function `match()`:

```
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 # => "@bedrock"
julia>m.offset # => 14
julia>m.captures # => 0-element Array{Union{SubString{UTF8String}, Nothing},1}
```

Byte array literals

Another special form is the byte array literal: `b"..."` which enables string notation express arrays of `UInt8` values.

The rules for byte array literals are the following:

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

Consider the following two examples:

```
julia> A = b"HEX:\xefcc" # => 7-element Array{UInt8,1}:[0x48,0x45,0x58,0x3a,0xef,0x63,0x63]
julia> B = b"\u2200 x \u2203 y" #=> 11-element Array{UInt8,1}:[0xe2,0x88,0x80,0x20,0x78,0x20,0xe2,0x88,0x83,0x20,0x79]
```

Version literals

Version numbers can easily be expressed with non-standard string literals such as `v"..."`.

Version number literals create `VersionNumber` objects which follow the specifications of semantic versioning (<http://semver.org>), 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: `v"0.3.1-rc1"`; the major version is "0", minor version "3", patch level "1" and release candidate is 1. Only the major version needs to be provided and the others assume default values. So `v"1"` is equivalent to `v"1.0.0"`.

We met the use of version numbers previously when using the package manager to pin a package to a specific version: `Pkg.pin("NumericExtensions", v"0.2.1")`.

An example

Let us look at some code to play the game *Bulls and Cows*. A computer program *moo*, written in 1970 at MIT in the PL/I, was among the first Bulls and Cows computer implementation.

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

The computer enumerates a four digit random number from the digits 1 to 9, without duplication. The player inputs his/her guess and the program should validate the player's guess, reject guesses that are malformed, then print the 'score' in terms of number of bulls and cows.

The score is computed as follows:

- 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
- Otherwise the program accepts a new guess, incrementing the number of 'tries'

Coding it up in Julia:

```
function bacs ()
    bulls = cows = turns = 0
    A = {}
    srand(int(time()))
    while length(unique(A)) < 4
        push!(A,rand('1':'9'))
    end
    bacs_number = unique(A)
    println("Bulls and Cows")
    while (bulls != 4)
        print("Guess? ")
        if eof(STDIN)
            s = "q"
        else
            s = chomp(readline(STDIN))
        end
        if (s == "q")
            print("My guess was "); [print(bacs_number[i]) for i=1:4]
            return
        end
        guess = collect(s)
        if !(length(unique(guess)) == length(guess) == 4 &&
            all(isdigit,guess))
            print("\nEnter four distinct digits or q to quit: ")
            continue
        end
        bulls = sum(map(==, guess, bacs_number))
        cows = length(intersect(guess,bacs_number)) - bulls
        println("$bulls bulls and $cows cows!")
        turns += 1
    end
    println("You guessed my number in $turns turns.")
end
```

The preceding code can be explained as follows:

1. We define an array A as A = {} rather than A = []. This is because although arrays were described as homogeneous collections, Julia provides a type Any which can, as the name suggests, store any form of variable. This is similar to the Microsoft variant datatype.

```
julia> A = {"There are ",10, " green bottles", " hanging on the
wall.\n"}
```

```
julia> [print(A[i]) for i = 1:length(A)]  
There are 10 green bottles hanging on the wall.
```

2. Integers are created as characters using the `rand()` function and pushed onto `A` with `push!()`.
3. The array `A` may consist of more than 4 entries so a `unique()` function is applied which reduces it to 4 by eliminating duplicates and this is stored in `bacs_number`.
4. User input is via `readline(STDIN)` 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.
5. A `collect()` function applied is applied to return a 4-element array of type `Char` and it is checked that there are 4 elements and that these are all digits.
6. The number of `bulls` is determined by comparing each entry in `guess` and `bacs_number`. This is achieved by using a `map()` function to applying the `==` operator, if 4 bulls then we are done. Otherwise it's possible to construct a new array as the intersection of `guess` and `bacs_number` which will contain all the elements which match. So subtracting the number of 'bulls' leaves the number of `cows`.

Real, complex, and rational numbers

Now we will consider how to handle real and complex numbers in Julia and also introduce an alternate representation of fixed-point reals as a fraction comprising two integers, the `Rational` datatype.

Further we will discuss the use of the `Big()` function to handle integers and real numbers which are too large to be represented by the primitive Julia numeric types.

Reals

We have met real numbers a few times already. The generic type is `FloatingPoint` which is sub-classed from `Real`:

```
abstract Real <: Number  
abstract FloatingPoint <: Real  
bitstype 16 Float16 <: FloatingPoint  
bitstype 32 Float32 <: FloatingPoint  
bitstype 64 Float64 <: FloatingPoint
```

A float can be defined as `x = 100.0` or `x = 1e2` or `x = 1f2`; all represent the number 100.

The first will be of the type equivalent to `WORD_SIZE`, the second of type `Float64` and the third (using `f` rather than the `e` notation) of type `Float32`.

There is also a `p` notation which can be used with hexadecimals, that is `x = 0x10p2` corresponds to `64.0`.

Operators and built-in functions

Julia provides comprehensive operator and function support for real numbers. There is a wealth of mathematical functions built-in. In addition to the 'usual' ones such as `exp()`, `log()`, `sin()`, `cos()`, and so on, there is support for gamma, bessel, zeta and hankel functions and many others.

The reader should look at the section of the manual on *Mathematical Operations* (<http://docs.julialang.org/en/release-0.3/manual/mathematical-operations/>) for a comprehensive list.

One feature to note is that the multiplication operator `*` can be omitted in places where there is no ambiguity. If `x` is a variable `2.0x` and `2.0*x` are both valid. This is useful in cases when dealing with pre-defined constants such as `pi`, where `2pi` is equal to `6.2831`.

Special values

In dealing with real numbers Julia defines three special values `Inf`, `-Inf` and `NaN`. `Inf` and `-Inf` refer to values greater (or less) than all finite floating-point values and `NaN` is "not a number" which is a value not equal to any floating-point value (including itself). So `1.0/0.0` is `Inf` and `-1.0/0.0` is `-Inf`, whereas `0.0/0.0` is `NaN`, as is `0.0 * Inf`.

Observe that `typemin(Float64)` and `typemax(Float64)` are defined as `-Inf` and `Inf` respectively rather than the minimum/maximum representation.

BigFloats

Earlier, in regard to integers, we met `BigInts`.

Unsurprisingly, there are also `BigFloats` which can be used for arbitrary precision arithmetic:

```
julia>h_atoms_in_universe = 1.0*10.0^82 # => 1.0e82
julia>x = BigFloat(h_atoms_in_universe)
9.999999999999963406796563088657421102714322527356779368036384342708650
1542887e+81 with 256 bits of precision
```

The default precision, nominally 256, and rounding mode of `BigFloat` can be changed using `with_bigfloat_precision()` and `with_rounding()` functions.

Notice that it is also possible to apply the `big()` function to achieve a similar result and the function returns either a `BigInt` or `BigFloat`, corresponding to the type of its argument:

```
julia> k = big(7^7);      typeof(k); # => BigInt  
julia> k = big(7.0^7);  typeof(k); # => BigFloat
```

Rationals

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

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

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

The functions `num()` and `den()` return the numerator and denominator of a rational and `float()` can be used to convert a rational to a float.

```
julia> x = 17//100;  
num(x) # => 17  
den(x) # => 100  
float(x) # => 0.17
```

Complex numbers

There are two ways to define a complex number in Julia. First using the type definition `Complex` as its associated constructor `complex()`.

```
c = complex(1, 2); typeof(c) # => Complex{Float64};
```

Because the complex number consists of an ordered pair of two reals, its size is `complex128`. Similarly `complex64` has `Float32` arguments and `complex32` has `Float16` arguments.

The number `Complex(0.0, 1.0)` corresponds to the imaginary number `i`, that is `sqrt(-1.0)`, but Julia uses the symbol `im` rather than `i` to avoid confusion with a variable `i`, 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 the normal arithmetic operations:

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

Also complex functions `real()`, `imag()`, `conj()`, `abs()`, and `angle()`.

`abs` and `angle` can be used to convert the complex arguments to polar form, that is:

```
julia> c = 1 + 2im; abs(c) # => 2.23606
julia> angle(c) # => 1.107148 (radians)
```

Complex versions of many mathematical apply:

```
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
```

Juliasets

The Julia documentation provides the example of generating a Mandelbrot set. Given the name of the language we will instead look at the code to create a related fractal - the Juliaset.

Both the Mandelbrot set and Julia set (for a given constant z_0) are the sets of all z (complex numbers) for which the iteration $z \rightarrow z^2 + z_0$ does not diverge to infinity. The Mandelbrot set is those z_0 for which the Julia set is connected.

We create a file `jset.jl` 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 n_{\max} 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 n_{\max} .

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 until *Chapter 7, Graphics* it is quite easy to create a simple disk file using the portable bitmap (netpbm) format. This consists of a "magic" number $P_1 - P_6$, 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.

Then follows the image values (height \times width) which may be ASCII for P_1 , P_2 , P_3 or binary for P_4 , P_5 , P_6 . There are three different types of portable bitmap; B/W (P_1/P_4), Grayscale (P_2/P_5) and Colour (P_3/P_6).

The function `create_pgmfile()` 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 indices i, j 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.

So the main program looks like:

```
include("jset.jl")
include("pgmfile.jl")
h = 400; w = 800; M = Array(Int64, h, w);
```

```
c0 = -0.8+0.16im;
pgm_name = "juliaset.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 $(t1-t0) seconds.\n");
```

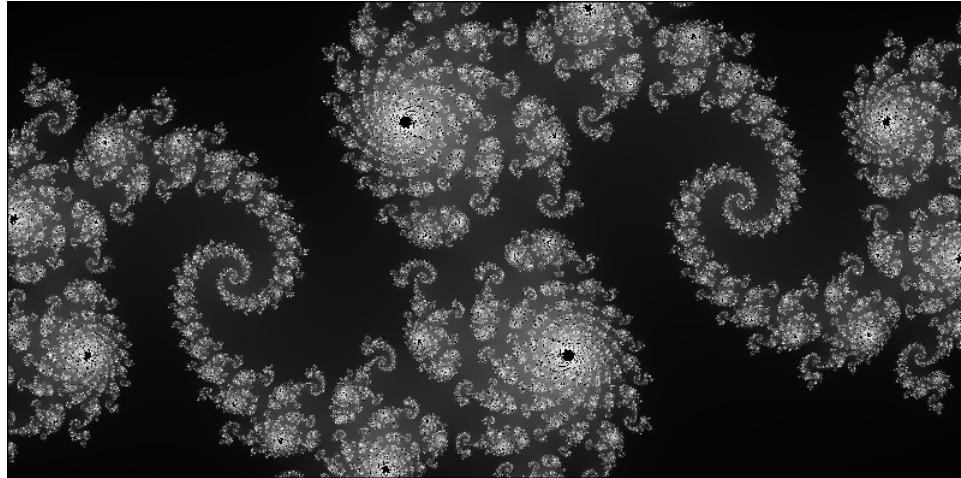
Consider how the previous code works:

1. We define an array `M` of type `Int64` to hold the return values from the `juliaset` function.
2. The constant `c0` is arbitrary, different values of `c0` will produce different Julia sets.
3. The starting complex number is constructed from the `(x,y)` coordinates and scaled to the half width.
4. We 'cheat' a little by defining the maximum number of iterations as 256. This is because we are writing byte values (`UInt8`) and the value which remains bounded will be 256 and since overflow values wrap around will be output as 0 (black).

The script defines a main program in a function `jmain()`:

```
julia>jmain
Written juliaset.pgm
Finished in 0.458 seconds # => (on my laptop)
```

The following screenshot shows the resulting image:



Composite types

A composition type is a collection of named fields, grouped together and treated as a single entity; these are termed records and structures in some programming languages.

If the type can also have functions (methods) associated with them the resulting collection is termed an object and the languages which support them (Java, C++, Python, Ruby, and so on) as object-oriented.

In Julia, functions are not bundled up with the data structures they operate on. The choice of the method a function uses is termed dispatch. When the types of ALL of a function's arguments are considered when determining the method employed, this is termed multiple dispatch and Julia uses this rather than the single dispatch we associated with object methods. We will be considering the implication of multiple dispatch in detail in the next chapter.

Composite type details are defined with the `type` keyword, followed by a list of field names, optionally annotated with the `::` operator and terminated with `end`. If the type of the field is not specified `Any` is assumed.

Consider a simple type definition for membership of a meetup group:

```
type Member
    fullname::ASCIIString
    email::ASCIIString
```

```
meetup::ASCIIString
age::Int
organiser::Bool
mobile::ASCIIString
end
me = ("Malcolm Sherrington", "malcolm@ljuug.org", "London Julia User
Group", 55, true, "07777 555555")
julia> names(me)
6-element Array{Any,1}: # => [:fullname,:email,:group,:mobile,:organiser,
:mobile]
julia>me.fullname #=> Malcolm Sherrington
julia>me.mobile #=> "07777 555555"  (-- not really my number --
```

More about matrices

Previously we looked at simple operations with two-dimensional arrays and two-dimensional matrices.

In this section I will also discuss some further topics to cover multi-dimensional arrays, broadcasting and of handling sparse arrays but first I wish to introduce the concept of vectorized and devectorized code.

Vectorized and devectorized code

Consider the following code to add two vectors:

```
function vecadd1(a,b,c,N)
    for i = 1:N
        c = a + b
    end
end

function vecadd2(a,b,c,N)
    for i = 1:N, j = 1:length(c)
        c[j] = a[j] + b[j]
    end
end

julia> A = rand(2); B = rand(2); C = zeros(2);
```

```
julia> @elapsed vecadd1(A,B,C,100000000)
@elapsed vecadd1(A,B,C,100000000) # => 18.418755286

julia> @elapsed vecadd2(A,B,C,100000000)
@elapsed vecadd2(A,B,C,100000000) # => 0.524002398
```

Why the difference in timings? The function `vecadd1()` uses the array plus operation to perform the calculation whereas `vecadd2()` explicitly loops through the arrays and performs a series of scalar additions. The former is an example of vectorized coding and the latter devectorized, the current situation in Julia is that devectorized code is much quicker than vectorized.

With languages such as R, MATLAB, and Python (using NumPy) vectorized code is faster than devectorized but the reverse is the case in Julia. The reason is that in R (say) vectorization is actually a thin-wrapper around native-C code and since Julia performed is similar to C, calculations which are essentially concerned JUST with array operations will be comparable with those in Julia.

There is little doubt that coding with vector operations is neater and more readable and the designers of Julia are aware of the benefit of improving on timings for vector operations. That it has not been done is tantamount to the difficulty in optimizing code under all circumstances.

Multidimensional arrays

To illustrate the indexing and representation of arrays of higher dimensions, let us generate a three-dimensional array of 64 random numbers laid out in a 4 by 4 by 4 arrangement:

```
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
```

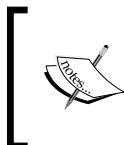
```

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

```



- Use of slice : to display the 3-D matrix
- Can reshape this into a 8x8 2-D matrix
- Values are ordered by the third index, then the second and finally the first



Notice that this can be recast into a two-dimensional array (that is, a matrix) by using the `reshape()` function:

```
julia> B = reshape(A, 8, 8)
8x8 Array{Float64,2}:
 0.522564  0.452363  0.570806  0.358262 ...  0.15706   0.252962
 0.817504
 0.992522  0.885484  0.810382  0.926146      0.86354   0.164883
 0.925754
 0.378972  0.757072  0.634989  0.773742      0.00568161 0.125772
 0.593097
 0.383636  0.304271  0.700649  0.321075      0.0476718  0.865795
 0.315774
 0.852847  0.444234  0.912306  0.494621      0.241825   0.432026
 0.274034
 0.450827  0.0693068 0.235757  0.915814 ...  0.715799   0.209135
 0.876917
 0.365945  0.807745  0.196174  0.158593      0.0868379  0.998318
 0.614772
 0.383711  0.389717  0.843975  0.306428      0.917008   0.204839
 0.520044
```

Similarly we can reshape the array `A` into a one-dimensional array:

```
julia> C = reshape(A,64); typeof(C) # => Array{Float64,1}
julia> transpose(C) ; # we can also write this as C'
1x64 Array{Float64,2}:
 0.522564  0.992522  0.378972  0.383636 ...  0.876917  0.614772  0.520044
```

The reshaped array `C` is of a single dimension but its transpose has two. So the former is a vector and the latter a matrix.

Broadcasting

We saw earlier that it was possible to carry out binary operations on individual elements of two arrays by using the operators `.+ .*`, and so on.

It is sometimes useful to perform operations on arrays of different sizes, such as adding a vector to each column of a matrix. One way to do this is to replicate the vector to the size of the matrix and then apply a `.+` operation but this becomes inefficient when the matrix is large.

Julia provides a function `broadcast()`, which expands singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory and applies the given function element-wise.

The following code generates a 4×3 matrix of Gaussian random numbers, adds 1 to each element and then raises elements of each *row* to the power $1, 2, 3, 4$ respectively:

```
julia> P = randn(4,3)
4x3 Array{Float64,2}:
 0.859635  -1.05159   1.05167
 0.680754  -1.97133   0.556587
 -0.913965   1.05069  -0.215938
 0.165775   1.72851  -0.884942

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

julia> broadcast(^,P + 1.0,Q)
4x3 Array{Float64,2}:
 1.85963      -0.0515925  2.05167
```

```

2.82494      0.943481   2.42296
0.000636823  8.62387    0.482005
1.84697      55.4247    0.000175252

```

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:

```

S1 = sparse(I, J, X[, m, n, combine])
S2 = sparsevec(I, X[, m, combine])
S3 = sparsevec(D::Dict[, m])

```

Where `S1` of will dimensions `m x n` and `S[I[k], J[k]] = X[k]`.

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

`S2` is a special case where a sparse vector is created and `S3` uses an associative array (dictionary) to provide the same thing. The sparse vector is actually an `m x 1` size matrix and in the case of `S3` row values are keys from the dictionary and the nonzero values are the values from the dictionary (see the section *Dictionaries, sets, and others* for more information on associative arrays).

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

```

julia> T = sprand(5,5,0.1)
# The 0.1 means only 10% for the numbers generated will be deemed as
# nonzero
5x5 sparse matrix with 3 Float64 nonzeros:
[1, 1] = 0.0942311
[3, 3] = 0.0165916
[4, 5] = 0.179855

julia> T*T

```

```
5x5 sparse matrix with 2 Float64 non-zeros:  
[1, 1] = 0.00887949  
[3, 3] = 0.000275282
```

The function `full()` converts the sparse matrix to a dense one:

```
julia> S = full(T); typeof(S) # => 5x5 Array{Float64,2}
```

Data arrays and data frames

Users of R will be aware of the success of data frames when employed in analyzing datasets, a success which has been mirrored by Python with the `pandas` package. Julia too adds data frame support through use of a package `DataFrames`, which is available on GitHub, in the usual way.

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

- `NA`: 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 in *Chapter 4, Interoperability*.

However, to get a flavor of processing data with these packages:

```
julia> Pkg.add("DataFrames")  
# if not already done so, adding DataFrames will add the DataArray and  
Blocks framework too.  
julia> using DataFrames  
julia> d0 = @data([1., 3., 2., NA, 6.])  
5-element DataArray{Float64,1}:  
 1.0  
 3.0  
 2.0  
 NA  
 6.0
```

Common operations such as computing `mean(d)` or `var(d)` [variance] will produce `NA` because of the missing value in `d[4]`:

```
julia> isna(d0[4]) # => true
```

We can create a new data array by removing all the NA values and now statistical functions can be applied as normal:

```
julia> d1 = removeNA(d0) # => 4-element Array{Float64,1}
julia> (mean(d1), var(d1)) # => (3.0, 4.66667)
```

Notice that if we try to convert a data array to a normal array, this will fail for d0 because of the NA values but will succeed for d1:

```
julia> convert(Array,d0) # =>MethodError(convert,(Array{T,N},[1.0,3.0,2.0
,NA,6.0]))
julia> convert(Array,d1) # => 4-element Array{Float64,1}:
```

Dictionaries, sets, and others

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

Dictionaries

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

Let us look at a simple datatype to hold a user's credentials: ID, password, e-mail, and so on. We will not include a username as this will be the key to a credential datatype. In practice this would not be a great idea, as users often forget their usernames as well as their passwords!

To implement this we use a simple module. This includes a type and some functions which operate on that type. Note the inclusion of the `export` statement which makes the type `UserCreds` and the functions visible.

```
moduleAuth

typeUserCreds
    uid::Int
    password::ASCIIString
    fullname::ASCIIString
    email::ASCIIString
    admin::Bool
end

function matchPwds(_mc)::Dict{ASCIIString,UserCreds}, _
    name::ASCIIString, _pwd::ASCIIString)
```

```
    return (_mc[_name].password == base64(_pwd) ? true : false)
end

isAdmin(_mc::Dict{ASCIIString,UserCreds}, _name::ASCIIString) = _mc[_name].admin;
export UserCreds, matchPwds, isAdmin;

end
```

We can use this to create an empty authentication array (AA) and add an entry for myself. We will be discussing security and encryption later, so at present we'll just use the `base64()` function to scramble the password:

```
julia> using Auth
julia> AA = Dict{ASCIIString,UserCreds}();
julia> AA["malcolm"] = UserCreds(101,base64("Pa55word"),"Malcolm
Sherrington","malcolm@myemail.org",true);
julia> println(matchPwds(AA, "malcolm", "Pa55word") ? "OK" : "No, sorry")
OK
```

Adding the user requires the scrambling of the password by the user, otherwise `matchPwds` will fail.

To overcome this we can override the default constructor `UserCreds()` 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.

The `using Auth` statement looks for `auth.jl` in directories on the `LOAD_PATH` but will also include the current directory. On a Linux system where v"0.3" is installed on `/opt` typically would be:

```
julia>println(LOAD_PATH)
Union(UTF8String,ASCIIString)
[/opt/julia//usr/local/share/julia/site/v0.3","/opt/julia/usr/share/
julia/site/v0.3"]
```

We can add to the `LOAD_PATH` with `push!:`

```
Julia>push!(LOAD_PATH, "/home/malcolm/jlmodules); # => if we add this
statement to the startup file .juliarc.jlit will happen whenever Julia
starts up.
```

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

```
julia> BB = [ "malcolm" => UserCreds(101,base64("Pa55word"),  
"Malcolm Sherrington","malcolm@myemail.org",true)];
```

So the values can be referenced via the key:

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

The . notation is used to reference the fields:

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

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

Attempting to retrieve a value with a key does not exist, such as AA["james"], will produce an error. We need to trap this in the module routines such as matchPwd and isAdmin using the try / catch / finally syntax.

So the isAdmin function in auth.jl could be rewritten as:

```
function isAdmin2(_mc::Dict{ASCIIString,UserCreds}, _  
name::ASCIIString)  
    check_admin::Bool = false;  
    try  
        check_admin = _mc[_name].admin  
    catch  
        check_admin = false  
    finally  
        return check_admin  
    end  
end
```

Sets

A set is a collection of distinct objects and the "Bulls and Cows" example earlier could have been implemented using sets rather than strings. Julia implements its support for sets in Base.Set (file: set.jl) and the underlying data structure is an associative array.

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

```
julia> S0 = Set(); # => Set{Any}()  
julia> S1 = Set(1,2,4,2); # => Set{Int64}(4,2,1)
```

Elements can be added to a set using the `push!()` function; recall `!` implies that the data structure is altered:

```
julia> push!(S0,"Malcolm"); push!(S0,21); # => Set{Any}("Malcolm",21)  
julia> push!(S1,"Malcolm"); # =>MethodError(convert,(Int64,"Malcolm") ,  
since set elements need to be type {Int64})
```

The usual set operations apply such as union, intersection, difference (`setdiff()`) and complement. It is possible to test for subset, note that:

```
julia> S0 = Set(1.0,2.5,4.0); S1 = Set(1.0); issubset(S1,S0) # => true  
julia> S0 = Set(1.0,2.5,4.0); S1 = Set(1); issubset(S1,S0) # => false
```

When dealing with integers there is a special type of set `IntSet` which produces an ordered set:

```
julia> K = IntSet(2,2,3,1,5,13,7,3,11) # =>IntSet(1, 2, 3, 5, 7, 11, 13)
```

Normal sets of type `{Int}` can be mixed with an `{IntSet}` and the result is a regular set not an `IntSet`:

```
julia> L = Set(3,1,11)  
julia>setdiff(K,L) # => 5-element Array{Int64,1}: (2, 3, 7, 11, 13)
```

Other data structures

The package `DataStructures` implements a rich bag of data structures including deques, queues, stacks, heaps, ordered sets, linked lists, digital trees, and so on.

The `Deque` type is a double-ended queue with allows 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 FILO and FIFO access respectively. `Deques` expose the `push!()`, `pop!()`, `shift!()`, and `unshift!()` functions.

A stack will use `push!()` and `pop!()` to add and retrieve data, a queue will use `push!()` and `unshift!()`. Queues encapsulate these processes as `enqueue!()` and `dequeue!()`.

Consider the following simple example to illustrate using stacks and queues:

```
julia> usingDataStructures
julia> S = Stack(Char,100); # => Stack{Deque{Char}}(Deque[])
julia> Q = Queue(Char,100); # => Queue{Deque{Char}}(Deque[])
julia> greet = "Here's looking at you kid!";
julia> for i = 1:endof(greet)
    push!(S,greet[i]) enqueue!(Q,greet[i]) end
julia> for i = 1:endof(greet) print(pop!(S)) end
!dikuoy ta gnikools'ereH
julia> for i = 1:endof(greet) print(dequeue!(Q)) end
```

Here's looking at you kid!

Summary

In this chapter we began looking at programming in Julia with a consideration of various scalar, vector and matrix data types comprising integers, real numbers, characters and strings, as well as the operators acting upon them.

We then moved on to more data types such as rational numbers, big integers and floats and complex numbers.

Finally we looked at some complex data structures such as data arrays, data frames, dictionaries and sets.

The next chapter follows on with a discussion of user-defined data types and introduces the idea of multiple dispatch, which is a central feature within the Julia language.

3

Types and Dispatch

In this chapter and the next, we will discuss the features that make Julia appealing to the data scientist and the scientific programmer.

Julia was conceived to meet the frustrations of the principal developers with the existing programming languages; it is well designed and beautifully written. Moreover, much of the code is written in Julia, so it is available for inspection and change. 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.

Functions

We have met functions in previous chapters and know how a `function() ... end` block works and that there is a convenient one-line syntax for the simplest of cases:

That is, `sq(x) = x*x` is exactly equivalent to the following:

```
function sq(x)
    y = x*x
    return y
end
```

The variable `y` is not needed (of course). It is local to the `sq()` function and has no existence outside the function call. So, the last statement could be written as `return x*x` or even just as `x*x`, since functions in Julia return their last value.

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, the preceding `sq(x)` function would work with any data structures where the * operator is defined as in the form `sq(x::Int) = x*x` and would only work for integers.

Surprisingly, perhaps, `sq()` does work for strings since the * operator is used for string concatenation rather than +.

```
julia> sq(x) = x*x  
julia> sq(" Hello ") ; # =>" Hello Hello "
```

To apply a function to a list of values, we can use the `map()` construct. Let's slightly modify the `sq()` function by using the .* operator instead of *. For scalars, this has no effect, but we can now operate with vectors and matrices.

```
julia> sq(x) = x.*x  
julia> map(sqAny[1, 2.0, [1,2,3]])  
3-element Array{Any,1}:  
1  
4.0  
[1,4,9]
```

Now, `sq()` will not work on strings, but since operators in Julia can be expressed in both infix and functional notations, the function can be easily rectified by defining the following:

```
julia> .*(s1::String, s2::String) = string(s1,s2)  
.*(generic function with 25 methods)
```

Now, our revised `sq()` function will accept strings as well.

Notice the output from the operator overload: `.*` (generic function with 25 methods).

This is because before this call, there were 24 methods as defined by the standard library and possibly overloads due to package addition. We can list the methods of a function by using the `methods()` call:

```
julia> methods(.*)  
# 25 methods for generic function ".*":  
.*(x::Real,r::Ranges{T}) at range.jl:293  
.*(r::Ranges{T},x::Real) at range.jl:294  
.*(T<:Number,S<:Number}(r::Ranges{T<:Number},s::Ranges{S<:Number}) at  
range.jl:324  
.*(x::Number,y::Number) at operators.jl:80  
. . . . .  
. . . . .  
. . . . .  
.*(s1::String,s2::String) at none:1
```

We can see that not only does this list the methods for which `.*()` is defined, but it also lists the modules in which the definition occurs and the line number. Our definition for strings is there too but at the console not in a file.

It may well be that previously, we defined the `sq()` function in order to apply it to the vector. Julia has a special syntax to set up an anonymous function by using the `->` operator.

```
julia> map(x -> x.*x, ([1 2 3]))  
1x3 Array{Int64,2}:  
1 4 9
```

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 according to Collatz, which states that the hailstone sequence for any starting number always terminates.

```
function hailstone(n)
    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

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 in order to converge, but all integer values seem to do so eventually.

```
for i = 1000:1000:6000
    (m,s) = hailstone(i)
    println("hailstone($i) => $m 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`) to 1. 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 `!` 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 an `if else end` block, initially seen in C but borrowed by many languages including Julia.

`n >> 1` is a bitshift right, so it effectively halves `n` when `n` is even

The sequence continues until an odd prime occurs, when it is tripled and one added, which results in a new even number and the process continues. While it is easy to see why the conjecture is true, the jury is still out on whether it has been proved or not.

It is worth noting that Julia orders its logical statements from left to right, so the operator or `(||)` is equivalent to `orelse` and the operator `(&&)` to `andthen`.

This leads to another couple of constructs, termed short-circuit evaluation, becoming popular with Julia developers:

```
(condition) || (statement) # => if condition then true else statement  
(condition) && (statement) # => if condition then statement else 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, namely the number of iterations and the generated array, which must be assigned to a tuple.

Passing arguments

Most function calls in Julia can involve a set of one or more arguments, and it is possible to designate an argument as being optional and provide a default value.

It is useful if the number of arguments may be of varying length and we may wish to specify an argument by name rather than by its position in the list.

How this is done is discussed as follows:

Default and optional arguments

In the previous examples, all arguments to the function were required and the function call was produced unless all were 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, which might occur or possibly trap the error and raise an exception.

For example, multiplying two integers results in an integer and two real numbers in a real number. If we multiply an integer with a real number, we get a real number. The integer is said to be promoted to a real number.

Similarly, when a real number is multiplied with a complex number, the real number is promoted to a complex number and the result is a complex number.

When a real number 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> a = [1.0,2,3]; println(sq(a))
ERROR: '*' has no method matching *(::Array{Float64,1},
::Array{Float64,1})

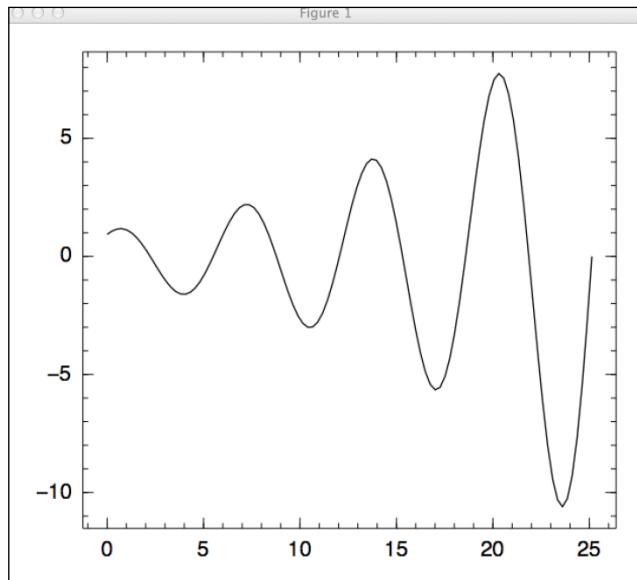
in sq at none:1
```

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, such as the following:

```
f(x, p = 0.0) = exp(p*x)*sin(x);
t = linspace(0.0,8pi);
w = zeros(length(t));
for i = 1:length(w) w[i] = f(t[i], 0.1) end
using PyPlot
plot (t, w)
```

The following figure shows a plot of this function ($p = 0.1$) using `PyPlot` to display the result. If $p = 0.0$ (default), clearly, we just get a sine wave:



Notice in the call, p is given the value $p=0.1$; however, we could still pass a value such as $p = 3$ as this would be promoted in the function body.

Let us now look at the methods for f :

```
julia> methods(f)
# 2 methods for generic function "f":
f(x) at none:1
f(x,p) at none:1
```

In fact, we could pass a rational number or even a complex number:

```
julia> f(2.0,3//4); # => 0.22313016
julia> f(2.0,2+3im); # => 0.01758613 + 0.0051176733im
```

Because of the complex argument, the result in the second case is a complex number too.

Optional arguments must come after the required ones as otherwise the meaning would be ambiguous. Further, when there are two optional parameters, values for the all preceding ones must be provided in order to specify the ones further down the list.

So, we define the following linear function: $f(x, y, a=2.5, b=4.0, c=1.0) = a*x + y *b +c.$

```
julia> f(1,1);      # =>7.5 : all parameters are defaulted
julia> f(1,1,2);   #=> 7.0 : sets equal to 2.0
julia> f(1,1,2.5,4.0,3.0); # => 9.5
```

This sets $c = 3.0$, but both a and b must be specified too.



Therefore, for long argument lists, 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 such a function exists, and `plus (+)` is an example of one such function.

The definition takes the following 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 code illustrates:

```
function g(a ,b, c...)
    n = length(c)
    if  n > 0 then
        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 rational numbers will work with this definition as they get promoted to real numbers:

```
julia> g(1.0, 2.0, 3//5, 5//7)
2-element Array{Float64,1}:
 2.2
 2.42857
```

Since functions are first-class objects these may be passed as arguments, so slightly modifying the definition of `g` gives a (very poor) map function:

```
function g(a ,b...)
    n = length(b)
    if n == 0 then
        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 in the next chapter, `NA`.

Named parameters

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 parameter c , 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.,1.,c=1.); # => 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 as follows:

```
function h(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> h(1.0,2.0,3.0, sigma=0.5)
3-element Array{Float64,1}:
 0.342006
 0.70062
 1.47813
```

So, h returns a Gaussian variable with mean μ and standard deviation σ .

Scope

In the previous example, we used the `?:` notation as a shorthand for `if-then-else`. However, it was necessary to wrap the code following the colon in `begin-end`.

This is a form of block, as are `f` statements and `for` and `while` loops. In fact, Julia signals the end of the most recent block by way of the `end` statement.

Other examples of blocks that we have met so far are those introduced by the `module`, `function`, and `type` definitions and by the `try` and `catch` statements.

The question we need to consider is as follows: If a variable is declared inside a block, is it visible outside it? This is controlled by Julia's scoping rules.

Since `if-then-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.

Certain constructs will introduce new variables into the current innermost scope. When a variable is introduced into a scope, it is also inherited by an inner scope unless the scope explicitly overrides it.

The following rules are reasonably clear:

- A `local` or `const` declaration introduces a new local variable
- A `global` declaration 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 of `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

Next, we clarify the last statement a function `f()` as follows:

```
function f()
    x = y = 0;
    while (x < 5)
        y = x += 1;
    end
    println(y)
end
f() ; # => 5

function f()
    x = y = 0;
    while (x < 5)
        local y = x += 1;
```

```
    end  
    return y  
end  
f() ; # => 0 :
```



Notice that the variable `y` in the while loop is local to it, and therefore, the value returned by the function is 0 not 5.

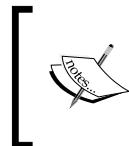


There is a further construct that Julia provides in passing anonymous function definitions as arguments, which is `do - end` and one that we will find convenient when working with the file I/O in the next chapter. Consider mapping an array to its squares when the value is 0.3 or more.

```
a = rand(5)  
map(x -> begin  
    if (x < 0.3)  
        return(0)  
    else  
        return(x*x)  
    end  
end, a)  
5-element Array{Real,1}:  
# => [0.503944 , 0.711046, 0 , 0.214098 , 0]  
map(a) do x  
    if (x < 0.3)  
        return(0)  
    else  
        return(x*x)  
    end  
end  
5-element Array{Real,1}:  
# => [0.503944 , 0.711046, 0 , 0.214098 , 0]
```

Both produce the same result, but the latter is cleaner and more compact. The use of the `do x` syntax creates an anonymous function with argument `x` and passes it as the first argument to `map`.

Similarly, `do a, b` would create a two-argument anonymous function and a plain `do` would declare that what follows is an anonymous function of the form `() -> ...`.



We should note that Julia does not (as yet) have a switch statement (as in C), which would be equivalent to successive `if else` statements. There is a package called `Match.jl`, which introduces a macro that will set up to generate multiple `if else` statements.

To illustrate this, let us consider the mathematicians' proof that all odd numbers are prime! We can code this concisely by using pattern matching as follows:

```
using Match

allodds(x) = @match x begin
    !isinteger(x) || iseven(x) || (x < 3) => "Not a valid choice"
    3 || 5 || 7 => "$x is prime"
    _ => "By induction all numbers are prime"
end
```

Running the preceding code on a select few gives the following:

```
for i in [1:2:9]
    @printf "%d : %s\n" i allodds(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
```

The Queen's problem

Finally, I will introduce a function that we will use in the next chapter for timing macros. This is to solve the Queen's problem, which was first introduced by Max Bezzel in 1848, and the first solutions were published by Franz Nauck in 1850.

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 8 queens on a chessboard so that no queen could take any other, although this was later generalized to N queens on an $N \times N$ 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 chessboard, 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 question.

The Julia implementation of the solution uses quite a few of the constructs that we have discussed:

```
qadd(board::Array{Vector{Int}}),
    qrow::Vector{Int}) = push!(copy(board), qrow)

qhits(a::Array{Int}, b::Array{Int}) =
    any(a .== b) || abs(a-b)[1] == abs(a-b)[2]

qhit(qrow::Vector{Int}, board::Array{Vector{Int}}) =
    any(map(x->qhits(qrow, x), board))

function qsolve(m, n=m, board=Array(Vector{Int}, 0))
    if n < 4
        return board
    end
    for px = 1:m
        for py = 1:m
            if !qhit([px, py], board)
                s = solve(m, n-1, qadd(board, [px, py]))
                s != nothing && return s
            end
        end
    end
    return nothing
end

qsolve(8) # => 8-element Array{Array{Int64,1},1}:
    [1,1], [2,5], [3,8], [4,6], [5,3], [6,7], [7,2], [8,4]
```

The code has an array of vectors to represent the board but could be written using a matrix.

- `push!()`: This is used to recreate a copy of the board in `qadd()`
- `qhit()` uses `map()`: This is to apply an anonymous function to the board
- `qsolve()`: This is the main function that 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.

Julia's type system

Julia is not an object-oriented language, so when we speak of objects, they are a different sort of data structure from those in traditional O-O languages.

Julia does not allow types to have methods, so it is not possible to create subtypes that inherit methods. While this might seem restrictive, it does permit methods to use a multiple dispatch call structure rather than the single dispatch system employed in 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, and 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.

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 and the source is in `base/rational.jl` and is available to inspection. Because `Rational` is a base type, it does not need to be included explicitly and we can explore it immediately:

```
julia> names(Rational)
2-element Array{Symbol,1}:
 :num
 :den
```

The `names()` 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, and 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.

To see how we can construct `Rational`, we can use the `methods()` function as follows:

```
julia> methods(Rational)
# 3 methods for generic function "Rational":
Rational{T<:Integer}(n::T<:Integer,d::T<:Integer) at rational.jl:11
Rational(n::Integer,d::Integer) at rational.jl:12
Rational(n::Integer) at rational.jl:13
```

Obviously, we wish to construct rationals from integers not floating-point numbers, but in Julia, there are integers of different sizes ranging from `Int8` to `Int32`, and we have signed and unsigned types of integers.

Rather than providing a recipe for making a rational for every combination of these, Julia provides three ways that are generic and the `methods()` function helpfully lists the line in the source file in which they occur.

- `Rational(n::Integer)`: This is when there is a denominator of 1 (assumed not passed)
- `Rational(n::Integer, d::Integer)`: This is when both the numerator and the denominator are provided
- `Rational{T<:Integer}(n::T, d::T) = Rational{T}(n,d)`: This is called a parametric definition for all types, which are of the `Integer` (`Int8`, `UInt8`, ...) types

Parametric definitions are very useful for establishing the rules for manipulating types as we will see later.

The entire source is quite long, but the first few lines are informative and are reproduced here:

```
immutable Rational{T<:Integer} <: Real
    num::T
    den::T
    function Rational(num::T, den::T)
```

```

num == den == 0 && error("invalid rational: 0//0")
g = den < 0 ? -gcd(den, num) : gcd(den, num)
new(div(num, g), div(den, g))
end
end

Rational{T<:Integer}(n::T, d::T) = Rational{T}(n,d)
Rational(n::Integer, d::Integer) = Rational(promote(n,d)...)

Rational(n::Integer) = Rational(n,one(n))

//(n::Integer, d::Integer ) = Rational(n,d)
//(x::Rational, y::Integer ) = x.num//(x.den*y)
//(x::Integer, y::Rational) = (x*y.den)//y.num
//(x::Rational, y::Rational) = (x.num*y.den)//(x.den*y.num)
//(x::Complex, y::Real      ) = complex(real(x)//y, imag(x)//y)
//(x::Real,      y::Complex ) = x*y'//real(y*y')

function //(x::Complex, y::Complex)
    xy = x*y'
    yy = real(y*y')
    complex(real(xy)//yy, imag(xy)//yy)
end

```

A rational number is defined using the `immutable` statement rather than `type`. This does not mean that a rational number once defined can't be changed, just that Julia makes it difficult.

```

Julia> r = 5//7; r.num = 3; println(r)
ERROR: type Rational is immutable

```

Although we said that type definitions can't contain methods: one that has the same name as the type is a special case and is called the (inner) constructor.

This is a recipe for how to build a datatype from its constituent parts. In the case of the rational, the constructor checks whether the denominator is zero, note the use of the short circuit form of evaluation terminating in an error statement (if true).

Otherwise, the constructor finds the **greatest common divisor (GCD)** of the denominator by the numerator in order to reduce the two numbers to their most primitive form. The special function called `new()` is called to return a value for the datatype.

Then, there are a number of outer constructors indicating the way to build rationals for different datatypes starting with the integer.

One other thing is that rational numbers can be defined for complex numbers as long as all of the real and imaginary parts of both complex numbers are integers. In fact, it is easy to achieve by multiplying the rational number with the complex conjugate of the denominator, which splits the number into a real (rational) and an imaginary (rational) part. Note that 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
```

The rest of the code in `rational.jl` deals with defining arithmetic operations on rationals. We will look at how this is done when discussing parametric types and multiple dispatch.

A vehicle datatype

Let us build a datatype from scratch, and as an example, let us look at data structures that describe vehicles and their ownership. This will be in a file called `vehicles.jl`, which needs to be added to the Julia environment by using the `include()` statement or encapsulated in a module.

```
type Contact
    name::String
    email::String
    phone::String
end
julia> methods(Contact)
# 1 methods for generic function "Contact":
Contact(name::String, email::String, phone::String)
```

All vehicles will have a person (or company) responsible for them. This provides the minimum information in terms of a name, e-mail, and phone number. All are strings and can be empty.

The definition creates default constructors that just define a new contact by supplying its arguments. To make some fields required or impose other restrictions on them, we need to provide an internal constructor.

```
function Contact(name::String, email::String, phone::String)
    length(name) == 0 && error("Need to provide a contact name")
    length(email) == 0 && length(phone) == 0 &&
        error("Need to provide either an email or a phone number")
    new(name, email, phone)
end

julia> methods(Contact)
# 1 method for generic function "Contact":
Contact(name::String,email::String,phone::String) at none:7

julia> me = Contact("", "", "")
ERROR: Need to provide a contact name
in Contact at none:7

julia> me = Contact("malcolm", "", "")
ERROR: Need to provide either an email or a phone number
in Contact at none:8
```

Now, we only have a single constructor and impose a compound rule that either the e-mail or the phone number must exist, as it is of little use to have a contact that we can't contact. Here, we could also impose maximum string lengths, which might be required if the data structure is eventually going to be saved as a database record.

```
abstract Vehicle
abstract Car <: Vehicle
abstract Bike <: Vehicle
abstract Boat <: Vehicle
abstract Powerboat <: Boat
```

This set of statements introduces a hierarchy of abstract types. These are types that do not have any attendant fields associated with them. At the top of the tree is Vehicle, and Car, Bike, and Boat are subtypes of Vehicle. Likewise, Powerboat is a subtype of Boat and implicitly of Vehicle.

The parent of any type can be shown by using the super() function and child types by using subtypes():

```
super(Car); # => Vehicle  
super(Powerboat) ; # => Boat  
  
julia> subtypes(Vehicle)  
3-element Array{Any,1}: ;# => [Bike, Boat, Car]
```

Subtypes in an Any array are only returned to one depth lower, so we do not know explicitly that a powerboat is a type of vehicle. We can use a different routine subtypetree(), which will provide the full breakdown of all subtypes not just the immediate children.

Now, we can start defining a few subtypes, which all define the actual makes of cars, bikes, and boats:

```
type Ford <: Car  
    owner::Contact  
    model::String  
    fuel::String  
    color::String  
    engine_cc::Int64  
    speed_mph::Float64  
    function Ford(owner, model, engine_cc,speed_mph)  
        new (owner,model,"Petrol","Black",engine_cc,speed_mph)  
    end  
end  
  
type BMW <: Car  
    owner::Contact  
    model::String  
    fuel::String  
    color::String  
    engine_cc::Int64  
    speed_mph::Float64
```

```
function BMW(owner,model,engine_cc,speed_mph)
    new (owner,model,"Petrol","Blue",engine_cc,speed_mph)
end
end

type VW <: Car
    owner::Contact
    model::String
    fuel::String
    color::String
    engine_cc::Int64
    speed_mph::Float64
end

type MotorBike <: Bike
    owner::Contact
    model::String
    engine_cc::Int64
    speed_mph::Float64
end

type Scooter <: Bike
    owner::Contact
    model::String
    engine_cc::Int64
    speed_mph::Float64
end

type Yacht <: Boat
    owner::Contact
    model::String
    length_m::Float64
end

type Speedboat <: Powerboat
    owner::Contact
    model::String
    fuel::String
    engine_cc::Int64
    speed_knots::Float64
    length_m::Float64
end
```

These are called concrete types and are subclasses of the abstract type preceding them. However, further concrete types can't be obtained as subclasses from them.

All vehicles have an owner and a model, but these are defined in the concrete type rather than in the `Vehicle` type, as would be the case in an object-orientated language. Likewise, all cars have top speed, fuel type, engine capacity, etc., but these can't be inherited from `Car`.

The constructor for the Ford car imposes a restriction that the color (according to Henry Ford) has to be black and that the fuel is petrol. Further, we set the color of all BMWs as blue, which is not strictly true.

These are not immutable types, so they could be changed after they were constructed.

With these definitions, we can start to instantiate some variables corresponding to the 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 using the following code:

```
jmCar.color = "Black"
```

Given the type structure, we can already do something with these:

```
cs = [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)
```

We can define the `cs` array since `Ford`, `BMW`, and `VW` are all subtypes of `Car`, and Julia creates the appropriate array and we can iterate over it as follows:

```
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 can define the vehicles that James owns as follows:

```
vs = [jmCar, jmBike, jmBoat]
```

This will create an `Any` array of the `Vehicle` type since it is the nearest common supertype, and we can use this array to list the vehicles as follows:

```
println("James owns the following:")
for v in vs
    model = v.model
    make = typeof(v)
    mtype = super(make)
    print("$mtype\t$make\t$model\n")
end
```

We can also define some functions using some of the type parameters. For example, we can find out if VW is quicker than BMW by defining the following function:

```
function isquicker(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
```

Further, we can compare different types of vehicles, such as a boat and a bike, taking account of the fact that a boat's speed is expressed in knots and a bike's or a car's speed in miles per hour or kilometers per hour.

```
function isquicker(a::Speedboat, b::Scooter)
    const 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

@printf "%s %s\n" isquicker(dvCar,jmCar) "has the faster car"
James has the faster car

@printf "The faster vehicle is the %s\n" isquicker(msBike,dsBoat)
The faster vehicle is the Sealine 28.
```

It is clear that there are a number of problems with this as it stands.

First although we can compare the speed of VW with BMW, we can't do this the other way round unless we define the following:

```
isquicker(a::BMW, b::VW) = isquicker(b,a)
```

Further, 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 bikes and boats. Julia solves this by using parametric types that we will meet in the next section.

Secondly, we require all vehicles to have a speed field, and if defining rules by using parametric types, the speed field will need to be the same symbol (that is, 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. The problem largely goes away if concrete types can inherit fields from their supertype(s).

The vehicle type and the accompanying constructors and function definitions are defined in the `vehicles.jl` file.

There is some merit in treating this as a module, and for this, we would add the following lines to the beginning of the file and terminate it with an end statement:

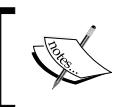
```
module Vehicles
export Contact, Vehicle, Car, MotorBike, Yacht, Powerboat, Boat
export Ford, BMW, VW, Scooter, Speedboat, isquicker
```

This is now accessed by means of the `using Vehicles` statement and will be picked up from the current folder or in a special array called `LOAD_PATH`.

Further, we can add our own folder(s) to this by using a `push!()` call as follows:

```
push!(LOAD_PATH, "/user/malcolm/jlmods"); println(LOAD_PATH)
```

Putting this in the `.juliarc` file will ensure that it happens each time that Julia is started.



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 and has to be explicitly referenced as follows: `Vehicles.islonger(jmBoat, dvBoat)`.

Modules can also have `using` statements, so the use of `myModule` implies that `myModule` will be available for resolving names as needed. It is possible to restrict this to a set of names such as `using myModule, myModule.fn2, myModule.fn3`, and this can be shortened to `using myModule: fn1, fn2, fn3`.

Modules can also use `import` statements to reference functions from other modules and `importall`, which will import all functions exported from a module rather than individual ones. The `import myModule: fn1, fn2, fn3` syntax is a shorthand for importing three functions from `myModule`.

Modules can also have `import` statements. All these support the same syntax as `using` but only operate on a single name at a time. Importing does not add modules to be searched the way `using` does and needs to have the functions to 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 a convenience than a necessity. When developing using the console REPL, any types defined are fixed, and once defined, we cannot change them without restarting Julia and redefining them. However, modules can be reused and this will effectively redefine all that they contain including any defined types.

Typealias and unions

It is often convenient to introduce a new name for an already expressible type, and for this, Julia provides the `typealias` keyword. In terms of 1D and 2D arrays, we have seen the use of the `Vector` and `Matrix` terms; these are as follows:

```
typealias Vector{T} Array{T,1}
typealias Matrix{T} Array{T,2}
```

The `typealias` keywords can be defined for parametric types `T`; these are discussed in the next section.

`Matrix{Float64}` is equivalent to writing `Array{Float64, 2}`, and `Matrix` has as instances all `Array` objects where the second parameter (the number of dimensions) is 2, for all element types. In Julia, this allows one to write just `Matrix` for the abstract type including all 2D dense arrays of any element type.

Typealias is useful when defining the `umbrella` type as a union of simpler ones. Union types are extensively used in `Base`, and there are many examples in the code listing there. An example from `int.jl` is as follows:

```
typealias Signed64 Union(Int8, Int16, Int32, Int64)
typealias Unsigned64 Union(UInt8, UInt16, UInt32, UInt64)
typealias Integer64 Union(Signed64, Unsigned64)
```

Recall that in our vehicle type, we provided contact details 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:

```
type Address
    name::String
    street::String
    city::String
    country::String
    postcode::String
end

postal = Address("Malcolm Sherrington", "1 Main Street",
    "London", "UK", "WC2N 9ZZ");

typealias Owner Union(Contact, Address)
```

The alias allows us to supply the owner field either as contact or postal details:

```
type Yacht <: Boat
    owner::Owner
    make::String
    length_m::Float64
end

y1 = Yacht(me, "Moody 36", 11.02)
Yacht(Contact("malcolm", "malcolm@abc.com", "07777555999"),
    "Moody 36", 11.02)

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; # => "malcolm"
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
```

Enumerations (revisited)

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 set of values. Previously, we discussed a macro that provides one approach to enumerations. In the absence of a preferred definition in Julia, various developers have adopted different strategies and I'll provide our own here.

First, we will use a vector of type {Any} to hold the enumerated values. This could be consts using integers or strings, but I'll restrict it to a list of symbols and create the vnum.jl file to hold the following:

```
typealias VecAny Array{Any,1}
function vnum(sym::Symbol...)
    A = []
    for v in sym
        push!(A, v)
    end
    A
end

function vidx(A::VecAny, a::Symbol)
    for (i, v) in enumerate(A)
        if v == a then
            return (i - 1)
        end
    nothing
    end
end
```

```
    end
end
vin(A::VecAny, a::Symbol) = (vidx(A,a) >= 0 ? true : false)
```

vnum is created by pushing a variable list of symbols onto an empty Any vector. Additionally, there is a function called vidx(), which returns the position in the enumeration; holding with convention, this is zero-based, and a one-line vin() function checks whether a symbol is that of v-numeration.

We can use this to define our fuels types as follows:

```
fs = vnum(:NONE,:PETROL,:DIESEL,:LPG);
vidx(fs,:DIESEL); # => 2
vin(fs,:NONE);    # => true
vin(fs,:GASOIL); # => false
```

Further, we can now define a Fuel type to be used in Vehicles as follows:

```
type Fuel
    fuel
    function Fuel(fuel)
        fs = vnum(:NONE,:PETROL,:DIESEL,:LPG)
        vin(fs,fuel) ? new(fuel) : error(TypeError)
    end
end
```

Multiple dispatch

A function is an object that maps a tuple of arguments to a return value. In a case where the arguments are not valid, the function should handle the situation cleanly by catching the error and handling it or throw an exception.

When a function is applied to its argument tuple, it selects the appropriate method and this process is called dispatch. In traditional object-oriented languages, a method is chosen based only on the object type and this paradigm is termed single dispatch. With Julia, the combination of all function arguments determines which method is chosen; this is the basis of multiple dispatch.

To the scientific programmer, all this seems very natural. It makes little sense in most circumstances for one argument to be more important than the others. In Julia, all functions and operators (which are also functions) use multiple dispatch. The methods are chosen for any combination of operators.

For example, look at the methods of the power operator (^):

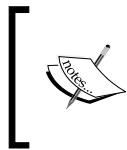
```
julia> methods(^)
# 43 methods for generic function "^":
^(::Bool,y::Bool) at bool.jl:41
^(::BigInt,y::Bool) at gmp.jl:314
^(::Integer,y::Bool) at bool.jl:42
^{T<:FloatingPoint}(z::Complex{T<:FloatingPoint},
    p::Complex{T<:FloatingPoint}) at complex.jl:322
^{T<:Complex{T<:Real}}(z::T<:Complex{T<:Real},
    p::T<:Complex{T<:Real}) at complex.jl:369
^{T<:FloatingPoint}(z::Complex{T<:FloatingPoint},n::Bool)
    at complex.jl:431
^{T<:Integer}(z::Complex{T<:Integer},n::Bool)
    at complex.jl:432
^(A::Array{T,2},p::Number) at linalg/dense.jl:180
^{::MathConst{:e}},x::AbstractArray{T,2}) at constants.jl:87
```

We can see that there are 43 methods for ^ and the file and line where the methods are defined are given too.

Because any untyped argument is designed to be of the Any type, it is possible to define a set of function methods such that there is no unique specific method applicable to some combinations of arguments.

```
julia> pow(a,b::Int64) = a^b;
julia> pow(a::Float64,b) = a^b;
Warning: New definition
pow(Float64,Any) at /Applications/JuliaStudio.app/Contents/Resources/
Console/Console.jl:1
is ambiguous with:
pow(Any,Int64) at /Applications/JuliaStudio.app/Contents/Resources/
Console/Console.jl:1.
To fix, define pow(Float64,Int64) before the new definition.
```

A call of pow(3.5, 2) can be handled by either function. In this case, they will give the same result because of the function bodies and Julia can't know that.



Although Julia does not disallow this, it does issue a warning and suggest a remedy. Sometimes, as a programmer, you will see such warnings when using a cocktail of packages. In many cases, they can be ignored, but you need to be aware of them.

Parametric types

Consider the following function definition: `f{T}(x::T, y::T) = x + y*y.`

We have seen this construct a few times already.

The `f` function will return its first argument added to the square of the second but only if the two arguments are of the same type.

```
f(1,2); # => 5
```

```
f(1.0,2.0); # => 5.0
```

```
f(1, 2.0)
```

```
ERROR: 'f' has no method matching f(::Int64, ::Float64)
```

This is not just true for integers and floats but also for all types provided that the function is valid. So, it would give an error for strings because the `+` operator is not defined, although the `*` operator is; overload `+`, and it will now work!

```
f(1+2im, 3+4im) ; # => -6 + 26im
f(1//2, 3//4); # => 17//16
```

This function definition is in terms of the parametric type `T`. We can use parameter types for defining new types, and this is the situation in which it proves to be the most useful. Consider the definition of an ordered pair of numbers (x, y) :

```
immutable OrdPair{T <: Integer}
    x::T;
    y::T;
end
```

Next, create a few ordered pairs:

```
julia> a = OrdPair(1,2); # => OrdPair{Int64}(1,2)1
julia> a = OrdPair(1.0,2.0);
ERROR: 'OrdPair{T<:Integer}' has no method matching ordpair{T<:Integer}
(::Float64, ::Float64)
```

The last case is clearly not what we want to happen. When creating an ordered pair of an integer and a float, we would like the result to be of two floats. Similarly, if we created a pair of a float and a complex, we would expect this to be a pair of two complex numbers.

Going up the type chain to the next common type is called promotion, and we will see how to do this later.

First, it would be a good idea to be able to do something with our ordered pairs, and the standard definitions of arithmetic operations seem to be a good place to start.

```
- (p::OrdPair) = OrdPair(-p.x, -p.y)
+ (p::OrdPair, q::OrdPair) = OrdPair(p.x + q.x, p.y + q.y)
- (p::OrdPair, q::OrdPair) = OrdPair(p.x - q.x, p.y - q.y)
*(p::OrdPair, q::OrdPair) =
    OrdPair (p.x * q.x - p.y * q.y, p.x * q.y + p.y * q.x)
```

You will probably recognize from the multiplication rule that this is a redefinition of complex numbers in terms of order pairs. We could equally have used rationals in this example, but then, we would need to restrict it to only the integer types.

We would need to define division (/), which is trickier and the power operation (^). For this (continuing with the complex number parallel), we need to define some more elemental operations such as the `abs()` function and create a conjugate pair `conj()`.

```
abs2(p::OrdPair) = p.x*p.x + p.y*p.y
abs(p::OrdPair) = sqrt(abs2(p))
conj(p::OrdPair) = OrdPair(p.x, -p.y)
```

We also need some logical function definitions, such as a way to assert that two ordered pairs are equal:

```
==(p::OrdPair, q::OrdPair) = (p.x == q.x) & (p.y == q.y)
```

Although we are well on the way to defining methods for working with ordered pairs, we can still not multiply these by a scalar real (integer or float) or by the more usual definition of a complex number. For this, we will need to be able to convert other numeric data types to ordered pairs and have (promotion) rules on how to apply such conversions.

Conversion and promotion

Conversion and promotion depend heavily on parameter typing and enable us to define a short set of rules rather than having to elucidate every possible combination of types.

The number hierarchy is defined in the `boot.jl` file (in base) as follows:

```
abstract Number
abstract Real <: Number
abstract FloatingPoint <: Real
abstract Integer <: Real
abstract Signed <: Integer
abstract Unsigned <: Integer
```

So, with the exception of complex numbers, which we will treat as a special case, all types will be `Real`.

Conversion

To define a new conversion, simply provide a new method for the `convert()` function. Since our new type is tautologous with an ordered pair, the first rule is particularly easy.

```
convert(::Type{OrdPair}, z::Complex) =
    OrdPair(real(z), imag(z))
```

All we need to do is extract the real and imaginary parts of our complex number `z` and use them as the component parts in the `OrdPair()` constructor.

The `convert` function takes two arguments, namely `::Type{OrdPair}`, and the type of argument, which will be a real number as we have dealt with complex numbers previously.

There are three parametric rules:

```
convert{T<:Real}(::Type{OrdPair{T}}, x::Real)
    = OrdPair{T}(x, 0)
convert{T<:Real}(::Type{OrdPair{T}}, p::OrdPair)
    = OrdPair{T}(p.x, p.y)
convert{T<:Real}(::Type{T}, p::OrdPair) =
    isreal(p.x) ? convert(T, p.x) : throw(InexactError())
```

An ordered pair just maps to itself and a real number x to the pair $(x, 0)$.

It is also necessary to cover the cases where a variable is not a real number (a string perhaps) by throwing an error.

Further, we supply a couple of generic (non-parametric) methods:

```
convert(::Type{OrdPair}, p::OrdPair) = p
convert(::Type{OrdPair}, x::Real) = OrdPair(x, 0)
```

Promotion

Promotion refers to converting values of mixed types to a single common type. Although it is not strictly necessary, it is generally implied that the common type to which the values are converted can faithfully represent all of the original values.

Promotion to a common supertype is performed in Julia by the `promote` function, which takes any number of arguments and returns a tuple of the same number of values converted to a common type. If promotion is not possible (that is, not defined), the routine must throw an error.

```
promote(1, 2//3); # => (1//1, 2//3)
promote(1, 2.0 + 3.0im); #=> (1.0 + 0.0im, 2.0 + 3.0im)
```

In the second case, the first argument is promoted from an integer to a real number by an existing rule and then the whole pair is promoted to a complex number.

For this, we provide the code for the `promote_rule()` function:

```
promote_rule{T<:Real,U<:Real}(:Type{OrdPair{T}}),
  ::Type{U}) = OrdPair{promote_type(T,U)}

promote_rule{T<:Real,U<:Real}(:Type{OrdPair{T}}),
  ::Type{OrdPair{U}}) = OrdPair{promote_type(T,U)}
```

Here, we have two real numbers but have possibly different types, and we need to promote one of the real numbers to match the other.

A fixed vector module

Julia does not have a fixed array type, but in the source examples, there is an example of a 4-vector quaternion, which was defined by the great Irish mathematician William Rowan Hamilton in the nineteenth century. These have some odd properties, one of which is that multiplication is not commutative. Sadly for Hamilton, at the time, 4 vectors in the form of tensors proved much more useful, but recently, quarterions have been utilized in problems of computer vision.

We are going to finish this section by looking at a module to represent a more conventional 3-vector and provide a module to define it.

```
module Vectors3D
    import Base.show, Base.convert, Base.abs, Base.dot
    export Vector3D, getindex, phi
    immutable Vector3D{T <: Real}
        x::T
        y::T
        z::T
    end
    +(u::Vector3D, v::Vector3D) = Vector3D(u.x + v.x, u.x + v.y, u.z + v.z)
    -(u::Vector3D, v::Vector3D) = Vector3D(u.x - v.x, u.x - v.y, u.z - v.z)
    *(a::Number, u::Vector3D) = Vector3D(a*u.x, a*u.y, a*u.z)
    *(u::Vector3D, a::Number) = Vector3D(a*u.x, a*u.y, a*u.z)
    /(a::Number, u::Vector3D) = Vector3D(u.x/a, u.y/a, u.z/a)
    /(u::Vector3D, a::Number) = Vector3D(u.x/a, u.y/a, u.z/a)
    show{T}(io::IO, u::Vector3D{T}) = print(io, "[$(u.x), $(u.y), $(u.z)]")
    abs{T}(u::Vector3D{T}) = (u.x*u.x + u.y*u.y + u.z*u.z)^0.5
    function phi(u::Vector3D, v::Vector3D)
        w0 = u.x*v.x + u.y*v.y + u.z*v.z
        w1 = abs(u) * abs(v)
        try
            return atan(w0 / w1)
        catch
    end
```

```
    error("Division by zero")
end
end

dot(u::Vector3D, v::Vector3D) = u.x*v.x + u.y*v.y + u.z*v.z
getindex(u::Vector3D, i) = (i == 1 ? u.x : (i==2 ? u.y : u.z))
convert(::Type{Vector3D}, u::Array{Float64,1}) =
    Vector3D(u[1],u[2],u[3])
end
```

The 3D vector is written as a module `vectors3d`, so using or `require()` will find it on `LOAD_PATH` and any types defined in it can be redefined in the REPL by reloading it.

We need to provide rules for the standard arithmetic operations (+, -, *, and /). These are written in a functional form, for example, `+(vec1, vec2) =`

In order to overload some routines from the base to be able to print the vector (`show`), convert a normal vector and take the modulus (`abs`) and the `dot` product of the two vectors. We would also wish to compute other metrics such as the `cross` product (which Julia denotes as the `times()` function).

In addition to the modulus, we will compute the angle between the vectors as the function `phi()`. Since this is not an extension to the base, the routine needs to be exported along with the `Vector3D` type. Recall that any routine not exported can still be called by fully qualifying it as `Vectors3D.phi()`.

Further, it is necessary to provide a routine for returning a component of the 3D vector `getindex()` and we need to also be able to set components by using `setindex()` and iterate over a collection of vectors by providing three routines `start()`, `next()`, and `end()`, whose meanings are clear.



For a more complete definition of the 3D vector type, see the code samples accompanying this book.



Summary

In this chapter, we looked at how the Julia type system defines common numeric and string types and the role that multiple dispatch plays in creating an efficient mechanism for calling functions.

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 discussed the topic of parametric types and developed a numeric example of an ordered pair as an alternate formulation of complex numbers.

In the next chapter, we will complete our survey of coding in Julia, looking at how it encompasses interoperability with other programming languages and how it is simple to utilize functions and methods in these languages, such as C, Python, and R, from Julia.

We will also introduce concepts of homoiconicity and metaprogramming and how this leads to the definition and the use of runtime macros in Julia and will see how this simplifies the execution of code asynchronously.

Finally, we will see how it is possible to interface with the underlying operating system, to spawn and run tasks, chaining these by pipelining and to process the results within the Julia environment.

4

Interoperability

In this chapter, we focus on the cooperation between Julia and other languages and with the underlying operating system.

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, and a brief overview of this is given at the end of this chapter.

Interfacing with other programming environments

Julia has a rich and varied syntax, but it was always known that a vast wealth of code existed in object libraries covering a wide range of specialities. So, a simple mechanism to call functions from these libraries would prove to be useful, and a one-line native instruction call `ccall()` was added.

This has led to the development of packages that effectively wrap code around the existing application programming interfaces (APIs). An example would be to access a database that exposes an API to perform functions such as those to establish a database connection, to create and update records, and to execute queries.

Another major area of interoperability has been between Julia and Python, and we have already shown some evidence of that in using Python for displaying graphs and using the IPython system as a proxy IDE.

There is also some further work on interfacing with R, Java, and MATLAB, and I'll present a brief overview of this too later.

Calling C and Fortran

One of Julia's strengths is that it makes calling code written in C, and by implication Fortran, very easy. The code to be called must be available as a shared library rather than just a standalone object file.

Most C and Fortran libraries are compiled into shared libraries and are distributed as such. On Unix/Linux systems, these (usually) have the extension `.so`; on Mac OS X, `.dylib`; and on Windows, `.dll`. To assemble your own C code, it is necessary to make it position independently by compiling with the `-fPIC` switch.

Sometimes, the differences in operating systems make it necessary to segment the code and Julia provides a base set of functions in `osutils.jl`, which incorporate macros such as `@unix`, `@osx`, and `@windows` to facilitate such differentiation.

The C-interface comprises of three functions in Julia (`Base.ccall`), namely `ccall`, `cglobal`, and `cfunction`. The first is the most common and we will focus on its use here.

The most common syntax for `ccall` is as follows:

```
ccall((symbol, library), RetType, (ArgType1, ...), Arg1, ...)
```

The first argument is a tuple consisting of the function to be called (passed as a symbol by prefixing it with a colon) and the second component is the shared library in which the function is to be found.

The second is the return type of the function call. The return type, which may be any bits type, including `Int32`, `Int64`, `Float64`, or `Ptr{T}`, the last being a pointer to the values of type `{T}`. For (void) functions, that is, ones that do not return a value, Julia uses the `Ptr{Void}` construct.

The next argument is another tuple of the types of arguments to pass to the function. This is written as a literal tuple, not a tuple-valued variable or expression.

Julia's type system is very rich and is exposed to you, so it is possible to create any type of data structure here. If passing by reference (rather than by value), a pointer to the data structure will be passed.

Finally, there is a list of variables, written individually, not a tuple, matching the data types provided in the third argument.

As the first example, we will look at the generation of random numbers using the standard library on Linux. It should be noted that Julia does not use this method for its own generation of random numbers (in `random.jl`). This is written in Julia by using the Mersenne Twister method, but the randomization starting seed does use calls to system libraries and because of the difference in operating systems, it makes use of the underlying operating system and the macros mentioned previously. These are defined in the base file called `osutils.jl`.

Here, we are also going to call the `rand()` function from the `libc` library. This is a very simple function to call as it takes no input argument and returns a 32-bit integer, which corresponds to an integer in the range of $[0 : 2^{31}]$. This will not work in Windows where we will have to use the Windows Advanced Services API (`advapi32.dll`).

```
x = ccall( (:rand, "libc"), Int32, () ) ;# => 16807
x = ccall( (:rand, "libc"), Int32, () ) ;# => 282475249
```

To look at a more exacting usage, the following pair of calls retrieves a user's home folder given by the environment variable `$HOME` and pushes a folder of my Julia modules on to `LOAD_PATH`, which is an array of folders. Using `push!()` will put the folder at the end of the folder chain; `unshift!()` will put it at the beginning:

```
home = ccall( (:getenv, "libc"), Ptr{UInt8},
              (Ptr{UInt8},), "HOME")
Ptr{UInt8} @0x00007fa5dbe73fe5
jmods = string(bytestring(home), "/julia/mymods");
push!(LOAD_PATH, jmods);
println(LOAD_PATH)
3-element Array{Union{UTF8String, ASCIIString}}
"/usr/local/Julia/local/share/julia/site/v0.3"
"/usr/local/Julia/share/julia/site/v0.3"
"/home/malcolm/julia/mymods"
```

The second of the places listed above is a convenient place to add system-wide folders, which will be available to all. Notice that Julia defines `LOAD_PATH` in terms of the system version number, which may well be a good practice to adopt with our modules, while the language is still under rapid development.

The `getenv` routine returns a pointer to a null-terminated C-style string, and the call to `bytestring()` converts this into a Julia string.

Mapping C types

Julia has corresponding types for all C types; refer to the Julia online documentation (<http://docs.julialang.org/>) for a full list. One caveat is that the `Char` type in Julia is 32-bit, so C characters must be passed as `UInt8`.

Since `ccall` requires a tuple of argument types, Julia is able to convert any passed variables implicitly rather than the programmer needing to do so explicitly.

Julia automatically inserts calls to the `convert` function to convert each argument to the specified type, so it is not necessary for the programmer to explicitly make a call to a library function.

Calling a replacement power function such as `pow(3, 3)` in `libmymath` will function correctly:

```
ccall((:pow,"libmymath"),Float64,(Float64,Int32),x,n)

pow(x,n) = ccall((:pow,"libmymath"),Float64,(Float64,Int32),
(convert(Float64,x),convert(Int32,n))
```

Array conversions

When an array is passed to C as a `Ptr{T}` argument, it is never converted; Julia simply checks that the element type of the array matches `T` and the address of the first element is passed. This is done in order to avoid copying arrays unnecessarily.

Therefore, if an array contains data in the wrong format, it will have to be explicitly converted using a call such as `int32(a)`.

To pass array `A` as a pointer of a different type without converting the data beforehand (for example, to pass a `Float64` array to a function that operates on uninterpreted bytes), you can either declare the argument as `Ptr{Void}` or you can explicitly call `convert(Ptr{T}, pointer(A))`.

Type correspondences

On all currently supported operating systems, basic C/C++ value types may be translated into Julia types. Every C type also has a corresponding Julia type with the same name, prefixed by C.

This can help in writing portable code and remembering that `int` in C is not necessarily the same as `Int` in Julia.

In particular, characters in C are 8-bit and in Julia 32-bit, so Julia defines typealiases:

```
(unsigned) char # => typealias uchar UInt8
(signed)  short # => typealias cshort Int16
```

We met pointer types in `ccall` to the `libc` function `getenv()`, and these follow the same scheme, but they are not C typealiases:

```
char*          => Ptr{UInt8}
char** or *char[] => Ptr{Ptr{UInt8}}
```

In addition, the Julia API, discussed in the following sections, defines C-type `j_type_t*` as `Ptr{Any}`.

For a comprehensive list of C-types versus Julia-types, the reader is referred to the Julia documentation.

Calling a Fortran routine

When calling a Fortran function, all inputs must be passed by reference.

The prefix `&` is used to indicate that a pointer to a scalar argument should be passed instead of the scalar itself.

For example, to compute a dot product using the **basic linear algebra system (BLAS)** by calling a function from the LAPACK library, use the following code:

```
function compute_dot(DX::Vector{Float64}, DY::Vector{Float64})
    assert(length(DX) == length(DY))
    n = length(DX)
    incx = incy = 1
    product = ccall((:ddot, "libLAPACK"),
                    Float64,
                    (Ptr{Int32}, Ptr{Float64}, Ptr{Int32},
                     Ptr{Float64}, Ptr{Int32}),
                    &n, DX, &incx, DY, &incy)
    return product
end
```

The function requires, and sets, three scalars, namely `n`, `incx`, and `incy`, and two 1D arrays `DX` and `DY`. The arrays are passed by reference (by default), and this is also required for the scalars.

Notice the use of the `assert(condition)` statement to check that both arrays are of the same size. This has no action if the condition is true but produces an error otherwise. The function calls the `ddot()` routine in the LAPACK library, which returns a 64-bit float.

Calling curl to retrieve a web page

For a slightly longer example, I have included a set of calls to the `libcurl` library to grab a webpage, in this case, the Julia home page. We need to define a couple of constants to be sure that the curl request follows the relocation status. The full list of constants can be found in the `curl.h` file.

Then, it is a four-step process:

1. Initialize curl and return a pointer to a data structure to use in the remaining calls.
2. Set up the options, the most important of them being passed the URL of the page to be retrieved.
3. Perform the HTTP get operation.
4. Tidy up by release of the memory grabbed in step 1.

Notice that the final call passes a single argument, the address of the memory to be freed, but this must be given as a tuple. So, this is written as `(Ptr{Unit8},)` with the trailing comma necessary to distinguish it as a tuple rather than a simple bracketed expression.

```
const CURLOPT_URL = 10002
const CURLOPT_FOLLOWLOCATION = 52;
const CURLE_OK = 0
jlo = "http://julialang.org";
curl = ccall( (:curl_easy_init, "libcurl"), Ptr{Uint8}, ())
ccall((:curl_easy_setopt, "libcurl"), Ptr{Uint8},
      (Ptr{Uint8}, Int, Ptr{Uint8}), curl, CURLOPT_URL, jlo.data)
ccall((:curl_easy_perform, "libcurl"),
      Ptr{Uint8}, (Ptr{Uint8},), curl)
ccall((:curl_easy_cleanup, "libcurl"),
      Ptr{Uint8}, (Ptr{Uint8},), curl);
```

Here are the first eight lines of the result:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-us" lang="en-us">
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<title>The Julia Language</title>
<meta name="author" content="Jeff Bezanson, Stefan Karpinski, Viral Shah,
Alan Edelman, et al." />
<link rel="stylesheet" href="/css/syntax.css" type="text/css" />
<link rel="stylesheet" href="/css/screen.css" type="text/css"
media="screen, projection" />
```

Python

PyCall is a package authored by Steve Johnson at MIT, which provides a `@pyimport` macro that mimics a Python `import` statement.

It imports a Python module and provides Julia wrappers for all the functions and constants including automatic conversion of types between Julia and Python. Type conversions are automatically performed for numeric, boolean, string, and I/O streams plus all tuples, arrays, and dictionaries of these types.

Python submodules must be imported by a separate `@pyimport` call, and in this case, you must supply an identifier to use in them.

After adding it via `Pkg.add("PyCall")`, an example of its use is as follows:

```
julia> using PyCall
julia> @pyimport scipy.optimize as so
julia> @pyimport scipy.integrate as si
julia> so.ridder(x -> x*cos(x), 1, pi); # => 1.570796326795
julia> si.quad(x -> x*sin(x), 1, pi)[1]; # => 2.840423974650
```

In the preceding commands, the Python `optimize` and `integrate` modules are imported, and functions in these modules are called from the Julia REPL.

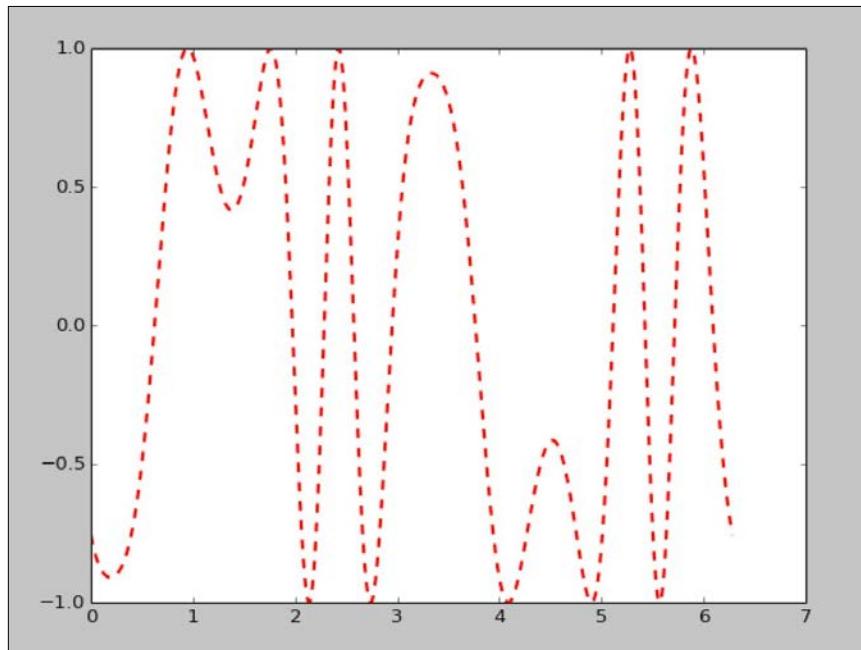
One difference imposed on the package is that calls using the Python object notation are not possible from Julia, so these are referenced using an array-style notation `po[:atr]` rather than `po.atr`, where `po` denotes a `PyObject` and `atr` represents an attribute.

It is also easy to use the Python `matplotlib` module to display simple (and complex) graphs:

```
@pyimport matplotlib.pyplot as plt
x = linspace(0,2*pi,1000); y = sin(3*x + 4*cos(2*x));
plt.plot(x, y, color="red", linewidth=2.0, linestyle="--")
1-element Array{Any,1}:
PyObject<matplotlib.lines.Line2D object at 0x0000000027652358>
plt.show()
```

Notice that keywords such as `color`, `linewidth`, and `linestyle` can also be passed in the preceding code.

The resulting plot is shown in the following figure:



Plotting using `matplotlib` is encapsulated in another package from Steve Johnson called PyPlot, which was introduced in *Chapter 1, The Julia Environment*, and will be considered in more detail when discussing visualization in *Chapter 7, Graphics*.

Further, when looking at IDEs for Julia, we meet IJulia, now incorporated as the part of the Jupyter project.

The latter is a package that provides a backend for IPython. Recall that the IJulia package needs to be installed, and Python with `matplotlib` has to be executable from the system path. A convenient way is to use an Anaconda bundle as this not only adds Python and IPython but also a large set of modules, including NumPy, SciPy, and Matplotlib. Other modules can be added using `easy_install`.

Once set up, IJulia is accessed by adding a profile option to an IPython notebook as follows:

```
ipython notebook --profile julia
```

Some others to watch

Python is by no means an exceptional case in providing inter-language cooperation hooks and since Julia is exposing such a wealth of riches to the developer, it is not surprising that packages are being written to link Julia to languages too. All packages are available on GitHub, and an overview can be found on pkg.julialang.org.

Four such packages are listed here:

- **R** (`Rif.jl`): This provides a link to the R interpreter and hence, the wealth of libraries available within R. There are a couple of caveats: in order to use this, R must be compiled as a shareable image using `-enable-R-shlib` (most distros are not built as shareable) and R must be available on the execution path. In addition, there is a recent package called `Rcall.jl` that permits an instance of R to be embedded in Julia provided that both R and RScript are on the execution path.
- **Java** (`JavaCall.jl`): This permits access to Java classes from within Julia by means of the **Java Native Interface (JNI)**, which in turn calls the **Java Virtual Machine (JVM)**. The package uses a `jcall()` function similar to `ccall()` and has been used to access an Oracle database via JDBC.
- **MATLAB** (`Matlab.jl`): This permits communication with MATLAB™ engine sessions. It is necessary to have a licensed copy of MATLAB installed, so it would be interesting to extend this to open source clones such as GNU Octave.
- **Mathematica** (`Mathematica.jl`): This likewise provides an interface for using Wolfram Mathematica™, and as with `Matlab.jl`, a licensed copy of Mathematica needs to be installed and executable.

All approaches will impose restrictions on Julia, and it is clearly better to work with the native language where possible. However, they do allow us to work with legacy code and the links to JVM will not have the limitations on speed that the others may have.

The Julia API

Julia is written in Julia, but not all of it. For speed and (sometimes) necessity, some of the implementation is coded in C and in LISP/Scheme, which requires Julia to have a runtime macro evaluation as we will see in the next section.

`boot.jl` (in base) comments out many of the types and functions that fall into this class and will use `ccall` to get them from the API. Note that `ccall()` has an alternate form when referencing these by using a symbol not a tuple.

Calling API from C

It is possible to access Julia from other languages by making calls to the Julia API, which is provided by the shared library called `libjulia`. This is created when Julia is built from source and will be located in a folder such as `$JULIA_HOME/usr/lib/julia` on Linux/OS X. On Windows, it is a DLL and is in the same folder (`bin`) as the Julia executable.

The library must be included in any linkage and locatable at runtime. Under Linux (say), this means using `ld.config` or adding the library to `LD_LIBRARY_PATH`:

Here is an example to print the Julia constant PI: `pi.c`

```
#include <julia.h>
#define LIBEXEC "/home/malcolm/julia/usr/lib"
int main() {
    jl_init(LIBEXEC);
    JL_SET_STACK_BASE;
    jl_eval_string("println(pi)");
    return 0;
}
```

It is necessary to include C-definitions in `julia.h`, and the program needs to know where the Julia system image (`sys.ji`) is located, which I've defined as `LIBEXEC`.

We build a version of Julia from source on CentOS 6.5 Linux by using the following commands:

```
export JH=/home/malcolm/julia
export LD_LIBRARY_PATH=$JH/usr/lib
gcc -o pi -I$JH/src -I$JH/src/support
-I$JH/usr/include -L$JH/usr/lib -ljulia pi.c

./pi ; # => π = 3.1425926535897 ...
```

If we wish to call Julia functions and pass C-variables, we need convert these to the Julia type. This is termed as **boxing**, and the generic `jl_value_t` is used with the appropriate box function for the C-type to be passed.

Here is a program that uses the Julia power operator (^) to compute the values of the square of PI (recall that ^ is a function in Julia).

```
#include <stdio.h>
#include <math.h>
#include <julia.h>
#define LIBEXEC "/home/malcolm/julia/usr/lib"
int main() {
    jl_init(LIBEXEC);
    JL_SET_STACK_BASE;
    jl_function_t *powf = jl_get_function(jl_base_module, "^");
    jl_value_t *arg1 = jl_box_float64(M_PI);
    jl_value_t *arg2 = jl_box_float64(2.0);
    jl_value_t *rtv = jl_call2(powf, arg1, arg2);
    JL_GC_PUSH1(&rtv);
    if (jl_is_float64(rtv)) {
        double e_pi = jl_unbox_float64(rtv);
        printf("pi (squared) is %f\n", e_pi);
    }
    else {
        printf("Oops, an error occurred!\n");
    }
    JL_GC_POP();
    return 0;
}
```

Building and executing gives the following: pi (squared) is 9.869604.

Notice the following points when calling the Julia API:

- We needed `stdio.h` for doing some C-based I/O (which we did not do previously)
- Further, `math.h` was included, just to pick a value for PI
- `jl_box_float64` was used to switch the C value to a Julia one float
- `jl_value_t` is a C-structure that contains the Julia type and a pointer to the data value; see `julia_internals.h` for its definition
- `^` is a function in base and has two arguments, so call `jl_call2` to apply the function to the arguments

- We needed to check whether the return value is `float64` (might be an error)
- If all was OK, the return value was an *unboxed* (to a C-value) value and printed

Julia is responsible for its own garbage collection, which is cleaning up any allocated memory when variables are defined or go out of scope. This can be triggered by the programmer by using the `gc()` function, but normally, this is not advisable.

When calling from C, it is not a good idea to assume that any memory allocated will be freed, so the purpose of `JL_GC_PUSH1` and `JL_GC_POP` C-macros is to indicate that the return value from the Julia call needs to be destroyed before the program exits.

Metaprogramming

Julia has been designed to tackle problems arising in science and statistics. It follows the route of traditional imperative languages. It is not a functional language, such as Haskell, OCaml, and Clojure, but since it incorporates a Scheme interpreter, it is possible to create runtime macros rather than the preprocessor style ones found in C/C++.

Macros are useful in efficiently generating boilerplate code, and we have already made use of the `@printf` macro, which emulates the C-style `printf()` function.

If a true functional approach is required, the reader is directed to the `Lazy.jl` package, which implements lazily evaluated lists plus a library of functions to use them and many well-designed macros. So, to get a grasp of Julia metaprogramming, you could do no better than looking at the source code in the `Lazy` package.

To generate macros, a language needs to have the property of homoiconicity. This means that the primary representation of program code is also a data structure in a primitive type of the language itself.

The obvious example is LISP, which only has the single data structure of a list, and a LISP program is also expressed in lists. This means that the interpreter is also able to parse and execute code as well as working with the underlying data.

In Julia, it is not clear how this is achieved, but before we look at this, we need to explore the Julia symbols in more detail.

Symbols

Consider the simple operation of multiplying two variables and adding the result to a third:

```
julia> ex = :(a+b*c)
:(a + b * c)
```

The use of the colon (:) operator in front of $(a + b * c)$ results in the expression being saved in `ex` rather than being evaluated.

In fact, the expression would clearly fail unless all the variables had values.

The colon (:) operator creates a symbolic representation, and this is stored in a data structure `Expr` and is saved in `ex` rather than being evaluated.

```
julia> typeof(ex); # => Expr
```

If we now initialize the variables `a`, `b`, and `c`, the expression can be evaluated as follows:

```
julia> a = 1.0;  b = 2.5;  c= 23 + 4im;
julia> eval(ex)
58.5 + 10.0im
```

So, this process effectively splits the calculation into two parts, storing the expression and then evaluating it.

`Expr` consists of a vector of three symbols:

```
julia> names(Expr)
3-element Array{Symbol,1}:
:head
:args
:typ
```

Further, if we look at the arguments of the expression `ex`, this is an array of symbols:

```
julia> ex.args
3-element Array{Any,1}:
:+
:a
:(b * c)
```

The third element is interesting since it is an n arithmetic expression and can be further reduced as follows:

```
julia> ex.args[3].args  
3-element Array{Any,1}:  
  :*  
  :b  
  :c
```

The other components of the `Expr` type are more straightforward:

```
julia> ex.head ; # => :call  
julia> ex.typ;    ; # => Any
```

More complex expressions would be difficult to store by means of the `:()` mechanism, so Julia provides a `quote ... end` block mechanism for storing code expressions. This two-stage evaluation forms the basis of Julia's macro capabilities.

Macros

The basis of Julia macros is in the incorporation of a LISP-style interpreter in its runtime. This allows code to be constructed, stored, and then initialized at a later stage and *spliced* inline.

Quite complex snippets of Julia, which would require considerable boilerplating, can be reduced to the use of a macro called from a single line of code.

Macros are constructed by using the `macro()` function, which has a name and takes a variable number of arguments: macro name (`ex1, ex2, ex3 ...`).

It is called with `@name`, and there are no commas between arguments as follows:
`@name ex1 ex2 ex3.`

One of the simplest macros calculates the execution time of a Julia function. This merely requires recording the time before the function is started and noting the time at the end of the execution of the function.

To write our own version of this, we can use the system library to return the Unix() `systime` which is the number of seconds since 1/1/1970. This is 32-bit value, so we are OK until 19/01/2038 when the time will overflow, a little after 3 am. For this, we will use a native call to the `time` function in `libc.so`:

```
systime() = ccall( (:time, "libc"), Int32, () );  
macro uxtime(ex)
```

```
quote
    t0 = systime()
    $(esc(ex))
    t1 = systime()
    println("Time taken was $(t1 - t0) sec.")
end
end
```

The code is too large for a single symbolization, so it is defined in a quote block. The expression/function to be timed is executed using the `$()` construct, and the use of `esc()` ensures that there is no name clash with the existing namespace.

The granularity of this macro is only to the nearest second, so in Julia, it is difficult to get routines that are computationally complex to "trouble to the scorers." However, the Queen's routine developed earlier for an 8×8 board on my i5-based computer takes a little time:

```
include('queens.jl')
@uxtime qsolve(8)
Time taken was 53 sec.
```

Of course, Julia has its own timing macro called `@elapsed`, similar to ours to the nanosecond rather than the second. This is defined in base in the `util.jl` file and to achieve this, makes a call to a routine in the Julia API: `ccall(:jl_hrtime, UInt64, ())`, which returns an unsigned 64-bit value.

The `macroexpand()` function can be used to show the code produced from a macro call. Consider a timing function `hangabout()` defined as follows:

```
function hangabout(n)
    s = 0.0
    for i = 1:n
        s += i/(i+1)^2
    end
    s
end

julia> hangabout(1_000_000_000) ; # => 17.35296236700634
julia> @elapsed hangabout(1_000_000_000); # => 5.786654631
```

This function is written to loop over an integer range and evaluate the (floating point) sum of $1/4$, $2/9$, $3/16$, ...; this series does not converge but increases very slowly.

Running this a billion times takes around 5.8 s on my laptop:

```
macroexpand(quote @elapsed hangabout(1_000_000_000) end)
quote # none, line 1:
begin # util.jl, line 68:
    local #173#t0 = Base.time_ns() # line 69:
    local #174#val = hangabout(1_000_000_000) # line 70:
    Base./ (Base.- (Base.time_ns(),#173#t0),1.0e9)
end
end
```

Expanding the macro gives a reference to the file where it is defined and you will see that it is very similar to our own. The first two lines of code create a copy of local variables to store the start time and the return value of `hangabout()`.

The third line looks a little odd, but note that `Base./` is a fully qualified call to the division operator, and likewise, `Base.-` is a call to subtraction. As noted, `time_ns()` returns the time in nanoseconds to the line corresponding to $(t1 - t0) / 1.0e9$.

To time a command, we will write a macro called `@benchmark`, which calls `@elapsed` 10 times and returns the mean.

```
macro benchmark(f)
quote
    $(esc(f))
    mean(@elapsed $(esc(f)) for i = 1:10)
end
end
```

This macro calls `@elapsed`, which will be expanded inline in the normal fashion. Further, the `f()` function is called once before the timing loop so that it will be compiled, if not already done so; therefore, the compilation time is not taken into account in the timing. The use of `esc()` is to avoid any possible name clashes while expanding the macro in the user's program context.

```
macroexpand(quote @benchmark hangabout(1_000_000_000) end)
quote # none, line 1:
begin # none, line 3:
```

```
hangabout(1000000000) # line 4:  
mean($Expr(:comprehension, quote # util.jl, line 68:  
    local #176#t0 = Base.time_ns() # line 69:  
    local #177#val = hangabout(1000000000) # line 70:  
    Base./(Base.-(Base.time_ns(),#176#t0),1.0e9)  
end, :(#175#i = 1:10)))  
end  
end
```

Lines 6–8 correspond to the `@elapsed` expansion; this is wrapped up as a list comprehension, and the whole is passed to the `mean()` function.

Not all macros are as simple as timing ones, and while in base, you may well look at an old favorite `@printf` in `printf.jl`. For an example of a cleverly constructed macro in terms of a number of helper functions, we advise you to look at the macro source code and the expansion of (say): `macroexpand(quote @printf "%s %\n" "The answer is " x end)`.

Package authors too make good use of macros and provide a good reference source. Often, the principal aim may be to define the main macro, as is the case in point with the `PyCall` package and `@pyimport`, which we met earlier.

Testing

Julia is a rapidly developing language, so it is very important to ensure that all code works and continues to work. All packages, in addition to an `src` folder, contain a test and in this are placed unit tests that can be run to assert that the package is not broken.

We have seen that Julia defines the `assert()` function (`error.jl`), which raises an "assert exception" if a condition is not met but does nothing otherwise. Also provided is an assert macro (`@assert`), which will evaluate a condition and output a custom message if given; otherwise, a default one.

Assert macros are often used to raise problems within a package where exceptions can be dealt with at runtime. For unit testing, it is more usual to utilize the functionality of a different base file: `test.jl`.

The simplest usage is the `@test` macro, as follows:

```
julia> using Base.Test  
julia> x = 1;
```

```
julia> @test x == 1
julia> @test x == 2
ErrorException("test failed: :((x==2))")
```

This will throw an error if the test fails. However, sometimes, we may wish to check for a particular type of error while reporting many others; this is done using `@test_throws`.

```
@test_throws
julia> a = rand(10);
julia> @test_throws BoundsError a[11] = 0.1
julia> @test_throws DomainError a[11] = 0.1
ERROR: test failed: a[11] = 0.1
in error at error.jl:21
in default_handler at test.jl:19
in do_test_throws at test.jl:55
```

As floating-point comparisons can be imprecise conditions, such as `sin(pi) == 0.0`, they may fail. In these cases, additional macros exist, which will take care of numerical errors.

Consider the following cases:

```
julia> @test_approx_eq 0.0 sin(pi)
ERROR: assertion failed: |0.0 - sin(pi)| <= 2.4651903288e-28
0.0 = 0.0
sin(pi) = 1.2246467991473532e-16
```

This still fails as the value retuned from `sin(pi)` is above the (default) permitted value of the macro call. However, it is possible to specify the allowable level.

```
julia> @test_approx_eq_eps 0.0 sin(pi) 1.0e-10
julia> @test_approx_eq_eps 0.0 sin(pi) 1.0e-20
ERROR: assertion failed: |0.0 - sin(pi)| <= 1.0e-20
0.0 = 0.0
sin(pi) = 1.2246467991473532e-16
difference = 1.2246467991473532e-16 > 1.0e-20
```

It is worth noting that a de facto standard testing framework is now provided by the use of the `FactCheck.jl` package. The reader is referred to the final chapter of this book when we will return to the question of testing in a more general discussion of package development.

Error handling

A handler is a function defined for three types of arguments: `Success`, `Failure`, and `Error`.

It is possible to specify different handlers for each separate case by using the `with_handler()` function.

```
julia> using Base.Test  
julia> my_handler(r::Test.Success) =  
    println("I'm OK with $(r.expr)");  
julia> my_handler(r::Test.Failure) =  
    error("Error from my handler: $(r.expr)")  
julia> my(r::Test.Error) = rethrow(r)
```

`Test.Success` and `Test.Failure` have a single parameter called `expr`, which is the expression, whereas `Test.Error` is a little more complex as it returns an error status and backtrace reference:

```
julia> names(Test.Error)  
3-element Array{Symbol,1}:  
  :expr  
  :err  
  :backtrace  
  
julia> my_handler(r::Test.Failure) =  
    println("$(r.expr): Yep, As expected!");  
julia> my_handler(r::Test.Success) =  
    println("$(r.expr): I'm OK with this");  
julia> my_handler(r::Test.Error) = rethrow(r);
```

It is possible to overwrite the `Error` status as well as `Success` and `Failure`, but normally, it is a good idea to rethrow the expression and let Julia report it. So, we use our handlers as follows:

```
Test.with_handler(my_handler) do
    x = 1;
    @test x == 1
    @test x == 2
    @test x / 0
end
x == 1: I'm OK with this
x == 2: Yep, As expected!
ERROR: test error during x / 0
```

The enum macro

When looking at our vehicle types, we constructed a simple representation from an enumeration, and I commented that the Julia language does not yet provide an enum type although various approaches have been suggested (and adopted) in dealing with problems that incorporate enumerations such as interfacing with database systems.

The examples section of Julia provides one such solution in terms of an `@enum` macro, and I'm going to close this section by having a look at it.

First, let's use it to define four possible status returns from a system call from informational to fatal, which perhaps might be inserted into a system log.

```
@enum STATUS INFO WARNING ERROR FATAL
::Type{enum STATUS (INFO, WARNING, ERROR, FATAL) }

INFO; # => INFO
STATUS(0); # => INFO
STATUS(1); # => WARNING
typeof(INFO); # => enum STATUS (INFO, WARNING, ERROR, FATAL)
```

`typeof(INFO)` looks a bit strange, so to see how this occurs, let us look at the definition of the enum macro:

```
macro enum(T,syms...)
    blk = quote
```

```
immutable $(esc(T))
    n::Int32
    $(esc(T))(n::Integer) = new(n)
end
Base.show(io::IO, x::$(esc(T))) = print(io, $syms[x.n+1])
Base.show(io::IO, x::Type{$(esc(T))}) = print(
    io,
    $(string("enum ", T, ' ', '(' , join(syms, ", "), ')'))
)
end
for (i,sym) in enumerate(syms)
    push!(blk.args, :(const $(esc(sym)) = $(esc(T))($(i-1))))
end
push!(blk.args, :nothing)
blk.head = :toplevel
return blk
end
```

Looking at this macro definition, we note the following:

- Using @enum is going to generate and splice the preceding code
- The symbolic representation in blk = quote ... end creates an immutable type based on the first argument T and defines the way to display it (Base.show) as the word enum followed by T and a join of the other arguments enclosed by brackets
- Looping through the arguments (syms...), which are passed as a tuple, creates a constant for each one, which is zero-based: \$(esc(T))(\$(i-1))
- These are pushed onto the macro's argument stack, and the process is terminated with the symbol (:nothing)
- The macro head is assigned to :toplevel
- This is similar in construct to our simple version in the previous section in that the symbols (in blk.args) are enumerated and pushed onto blk itself as the macro return via the push! () call
- Notice the overloading of Base.show(), which allows the enumeration to be displayed

The following code snippet illustrates the use of the `enum` macro:

```
require("Enum")
@enum MSGSTAT INFO WARN ERROR FATAL
typealias MsgStatus typeof(INFO)
enum MSGSTAT (INFO, WARN, ERROR, FATAL) (constructor with 1 method)

using Calendar
type LogMessage
    stamped::CalendarTime
    msgstat::MsgStatus
    message::String
end

import Base.show
show(m::LogMessage) =
print("$m.stamped) : $(m.msgstat) >> $(m.message)")

msg = LogMessage(Calendar.now(), WARN, "Be very afraid")
show(msg)
28 Aug 2014 09:55:33 BST: WARN >> Be very afraid!
```

Therefore, we can see the following:

- This snippet uses `require()` to find `enum.jl`. It's the routine used by using a package and will find the file in the current folder or on `LOAD_PATH`
- The type generated by the `@enum` macro is rather unwieldy, so `typeassert` is defined to the `typeof()`: anyone of the enum's values
- `LogMessage` has a timestamp of the `CalendarTime` type provided by the `Calendar` package, which also has the `now()` function to get the current (local) time
- Adding a `show()` routine for displaying log messages needs `Base.show` to be imported

Tasks

Tasks (aka co-routines) form the basis for Julia's provision of parallel processing. They are sometimes referred to as lightweight or green threads. When some code is executed as a task, it is possible to suspend it and switch to another task. The original task can be resumed and will continue from where it was suspended.

Tasks cooperate by using a producer-consumer mechanism. A producer task will halt at a point where it has some values, which need to be consumed, and a separate task will be able to access these values. Producer and consumer tasks can both continue to run by exchanging values as necessary.

```
function fibs(n = 10)
    fib = int64(zeros(n))
    fib[1] = 1
    produce fib[1]
    fib[2]
    produce fib[2]
    for i = 3:n
        fib[i] = fib[i-1] + fib[i-2]
        produce(fib[i])
    end
    produce(-1)
end
```

This function computes the first 10 numbers in the Fibonacci series. When this function is used to create a task, it will halt at each `produce()` statement until the value being signaled is consumed. The function sends `(-1)` as its last value to indicate that it is ending.

```
p = Task(fibs); #=> Task
consume(p); # => 1
.....
.....
consume(p); # => 55
consume(p); # => -1
```

A task is an iterable object, so the values produced can be used in a loop as follows:

```
for x in p
    if x < 0
        break
    else
        @printf "%d " x
    end
end
1 1 2 3 5 8 13 21 34 55
```

Parallel operations

Julia provides a multiprocessing environment based on message passing. The programmer only needs to explicitly manage one processor in a two-processor operation. The operations involve high-level function calls rather than traditional message send and message receive operations.

To do this, Julia introduces a remote reference; this is an object that can be used to refer to any process on a particular processor.

Associated with remote references are remote calls. These are requests to call a function with a given set of arguments on a specific processor (possibly its own) and return a remote reference to the call. In the case where the remote call fails, an exception is raised; otherwise, it runs asynchronously with the remote task. It is possible to wait for the remote task to finish by using its remote reference, and the value of the result can be obtained by using `fetch()`.

There are a few ways to set up a multiprocessor system. First, Julia can be started using the following: `julia -p n`, which will create `n` separate processors on the same computer. If the computer is multi-core, then each processor will run a separate core, so it is sensible to choose `n` to reflect this.

Separate files can also be preloaded on multiple processes at startup, and the main script can be used to control the overall process:

```
julia -p 2 -L file1.jl -L file2.jl main.jl
```

Each process has an associated identifier. The process providing the interactive Julia prompt always has an ID equal to 1, as would the Julia process running the main script in the preceding example. The processes used by default for parallel operations are referred to as workers. When there is only one process, process 1 is considered a worker. Otherwise, workers are considered to be all processes other than process 1.

Alternatively, it is possible to create extra processors from a running system by using `addproc(n)`. Since there is already a process running, the number `n` should now be chosen as one less than the number of cores:

```
julia> addprocs(3); # => equivalent to startup as: julia -p 4
julia> nprocs(); #=> 4
```

`nprocs()` confirms that four processors are now available.

An `nworkers()` call returns the number of workers available, which is one less than `nprocs()` in a multiprocessor environment; otherwise, it is 1.

After adding the workers, we can select one of them to execute some code, as follows:

```
r = remotecall(2, randn, 5)
fetch(r) ;# => [0.094615, 0.339905, 0.422061, 0.790972, 0.552309 ]
```

In a distributed setup, an alternative form of `addprocs(machines)` is available where `machines` is a vector with items taking the following form: `user@host:port`.

The user defaults to the current user and the port to SSH, so only the host is required. In the case of multihomed hosts, `bind_addr` may be used to explicitly specify an interface.

This form of `addprocs()` has three additional keyword arguments:

- `tunnel = false`: If true then SSH tunneling will be used to connect to the worker
- `dir = JULIA_HOME`: Specifies the location of the Julia binaries on the worker nodes
- `sshflags`: Specifies additional SSH options, for example, `sshflags=' -i /home/malcolm/key.pem'`

In the era of big data, this aspect of Julia's parallelism makes it very attractive. Worker processes can also be spawned on arbitrary machines quite transparently in a cluster environment, and it is possible to use the `ClusterManager` interface to provide our own way to specify, launch, and manage worker processes.

Parallel execution is added via a set of macros such as `@async`, `@spawn`, and `@spawnat`.

Let us run the Fibonacci series again but now using a recursive definition. This can be computationally expensive due to the double recursion unless defined by a tail-recursive scheme.

```
addprocs(3)

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

The `@everywhere` macro ensures that the function definition is available on all processors:

```
julia> @elapsed fib(40); # => 1.311472619
julia> r = @spawn @elapsed fib(40); # => RemoteRef(4,1,17)
julia> fetch(r); # => 1.272693849
```

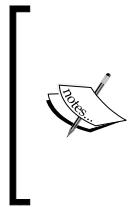
The `@spawn` macro ran the command on one of the workers chosen randomly, in this case, worker 4. The macro returned `RemoteRef`, and this was used to get the elapsed time.

It is possible to use `@spawnat`, which specifies which worker to select, and using the syntax `@spawnat 1 <args...>` will run the command on active processor (1) rather than a worker.

I'll look at spawning processes after we consider distributing our data.

Distributed arrays

When using a process on multiple computers, it is also desirable to be able to make use of the remote machine's memory. For this, Julia provides a DArray (distributed array). In order to create a DArray, there are functions such as `dzeros()`, `dones()`, `dfill()`, `drand()`, and `drandn()`, all of which function in a fashion similar to that of their non-distributed counterparts.



DArrays have been removed from Julia Base library as of v0.4, so it is now necessary to import the `DistributedArrays` package on all spawned processes.

If you are working with v0.4 or greater, please refer to the details on GitHub for information.

```
julia> addprocs(3);
julia> d = drand(100000);
julia> typeof(d) ; # => DArray{Float64,1,Array{Float64,1}}
```

This processes a distributed array of 100,000 random numbers distributed over the three workers but not the active machine.

The metadata associated with `d` is provided in a 5-element array of the special type `Any`.

As the name suggests, this is an array whose elements can be a mixture of types, integers, real numbers, strings, other arrays, and user-defined objects:

```
julia> names(d)
5-element Array{Symbol,1}:
 :dims
```

```
:chunks  
:pmap  
:indexes  
:cuts
```

The chunks parameter provides the remote references, and the indexes parameter provides the ranges on a per worker basis. Therefore, the length of either chunks or indexes gives the number of workers holding the data:

```
julia> d.cuts  
1-element Array{Int64,1},1:  
[1,33334,66668,100001]  
julia> d.chunks  
3-element Array{RemoteRef,1}:  
RemoteRef(2,1,5)  
RemoteRef(3,1,6)  
RemoteRef(4,1,7)  
julia> length(d.chunks); # => 3  
  
julia> d.indexes  
3-element Array{UnitRange{Int64},1},1:  
(1:33333,)  
(33334:66667,)  
(66668:100000,)
```

Notice that the dims parameter gives a vector of the dimensions of the array rather than how it has been sliced and diced:

```
d0 = dzeros(100,200,300) ; d.dims ; # => (100,200,300)
```

The data can be treated similar to any "normal" array.

For example, we can compute the frequency histogram and estimate the bias in the random number generator from the standard error of the mean:

```
julia> (r, fq) = hist(d,10)  
(0.0:0.1:1.0,  
[10047,10008,10004,9982,9892,9850,10023,10007,10142,10045])  
  
julia> stdm(fq, mean(fq)); # => 81.3388
```

A simple MapReduce

As an example of processing, let's generate a matrix of 300×300 normally distributed random numbers with a zero mean and unit standard deviation using three workers.

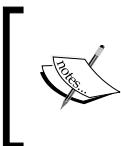
```
addprocs(3);  
d = drand(300,300);  
nd = length(d.chunks)
```

To work with data on an individual worker, it is possible to use the `localpart()` function. We need to apply the function to all the workers, which hold data, and for our purposes, this is a calculation of the values of the mean and standard deviation.

`procs()` provides an iterable array of workers, which can be used in the `@spawnat` macro. Finally, we wait for each process to yield results by using `fetch()` to map this over all workers and reduce the result by simple summation.

The total computation results in two lines of code:

```
μ = reduce(+, map(fetch,  
Any[@spawnat p mean(localpart(d)) for p in procs(d) ]))/nd;  
0.001171701  
σ = reduce(+, map(fetch,  
Any[@spawnat p std(localpart(d)) for p in procs(d) ]))/nd;  
1.001011265
```



Hint: Julia supports unicode characters and REPL can use these as normal variables. Therefore, to display (and use) the variables μ and σ within Julia, you would type `\mu<TAB>` and `\sigma<TAB>`, respectively.



Executing commands

We saw in the first chapter that it is possible to shell out of the console REPL to the operating system by using the `;` command. In addition, Julia provides a number of built-in functions so that scripts can interact with the OS and its filesystem. Most of these commands mirror the standard Unix commands. For Windows users, they will be familiar to those who have used the MINGW or similar shells.

Two familiar ones are `pwd()` and `cd()`:

```
julia> pwd();
"/Users/malcolm/Packt/Chapter-4"
julia> cd("../../Work");
julia> pwd();
"/Users/malcolm/Work"
```

The `pwd()` command indicates where we are now (print working folder), and of course, `cd()` is used to change the folder. This can be relative to the current location or an absolute if the file specification starts with `/`.

It is also possible to create new folders with `mkdir()` subject to your access permissions. Files may be copied `cp()`, moved `mv()`, or deleted `rm()`.

Folders are also deleted using the `rm()` command, which has a recursive argument (`default: false`) to remove a folder tree, with the caveat that the folder needs to be empty before it can be removed.

There is also the useful `download(url, [localfile])` command, which can get a remote file via HTTP. If the second argument is supplied, the download is saved to a disk file.

Commands not built into the language are easily added using the `ccall()` mechanism. For example, we saw that it is possible to find the value of an environmental variable by using a function, as follows:

```
function getenv(evar::String)
    s = ccall( (:getenv,"libc"), Ptr{UInt8}, (Ptr{UInt8},), evar)
    bytestring(s)
end
```

This can be used to locate the home folder and navigate from there. In the following examples, we have a (top-level) folder containing some of the poems of Lewis Carroll in the *Alice* books, and to get there, we can use the function as follows:

```
home_dir = getenv("HOME");
alice_dir = string(home_dir, "/Alice");
cd(alice_dir)
```

Running commands

For executing operating system commands, Julia uses a back tick convention, similar to Perl and Ruby. However, Julia does not run the command immediately, rather a `Cmd` object is returned, which can then be run later.

```
osdate = `date`;  typeof(osdate);  # => Cmd
run(osdate)
Sun 31 Aug 2014 09:26:02 BST
```

Having the ability to store commands in this manner gives great flexibility. Instead of using `run()` as mentioned previously, we can use `readall()` to capture the output:

```
julia> pwd(); # => "/Users/malcolm/Alice"

julia> ls = readall('ls -lnoSr')
"total 112\n-rw-r--r-- 501 1 813 28 Aug 15:37 voice-of-the-lobster.txt\n-rw-r--r-- 501 1 968 28 Aug 15:37 jabberwocky.txt\n-rw-r--r-- 501 1 1182 28 Aug 15:37 lobster-quadrille.txt\n-rw-r--r-- 501 1 1455 28 Aug 15:37 father-william.txt\n-rw-r--r-- 501 1 1887 28 Aug 15:37 mad-gardeners-song.txt\n-rw-r--r-- 501 1 2593 28 Aug 15:37 aged-aged-man.txt\n-rw-r--r-- 501 1 3415 28 Aug 15:37 walrus-and-carpenter.txt\n-rw-r--r-- 501 1 25225 28 Aug 15:37 hunting-the-snark.txt\n"
```

This is a listing of the `Alice` folder obtained from the `ls` command, which is returned as a string. It looks a little odd until we realize that the lines are delimited by line feed characters (`\n`), and so, we can split the output on these to produce a separate array of strings:

```
julia> split(ls, "\n")
8-element Array{SubString{ASCIIString},1}:
"total 112"
"-rw-r--r-- 501 1 813 28 Aug 15:37 voice-of-the-lobster.txt"
"-rw-r--r-- 501 1 968 28 Aug 15:37 jabberwocky.txt"
"-rw-r--r-- 501 1 1182 28 Aug 15:37 lobster-quadrille.txt"
"-rw-r--r-- 501 1 1455 28 Aug 15:37 father-william.txt"
"-rw-r--r-- 501 1 1887 28 Aug 15:37 mad-gardeners-song.txt"
"-rw-r--r-- 501 1 2593 28 Aug 15:37 aged-aged-man.txt"
"-rw-r--r-- 501 1 3415 28 Aug 15:37 walrus-and-carpenter.txt"
"-rw-r--r-- 501 1 25225 28 Aug 15:37 hunting-the-snark.txt"
```

Often, we wish to do more than just capture the output of the command as a string. It is possible to do this by opening the standard output (STDOUT) and working within a `do - end` block.

For example, in the spirit of "Through the Looking Glass," here are the first four lines of Jabberwocky as Alice first saw them:

```
open('cat jabberwocky.txt',"r",STDOUT) do io
    for i = 1:4
        s = readline(io)
        println(reverse(chomp(s)))
    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
```

Revisiting an earlier example where we interfaced with `libcurl` to get a copy of the Julia homepage, this can be done easily by using an OS command:

```
Julia> jlo = readall(`curl -L "http://www.julialang.org"`);
julia> [println(jlo[i]) for i = 1:8];

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en-us" lang="en-us">
<head>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
<title>The Julia Language</title>
<meta name="author" content="Jeff Bezanson, Stefan Karpinski, Viral Shah,
Alan Edelman, et al." />
<link rel="stylesheet" href="/css/syntax.css" type="text/css" />
<link rel="stylesheet" href="/css/screen.css" type="text/css"
media="screen, projection" />
```

Working with the filesystem

We should note that when using commands in Julia, the shell expansions do not work. In particular, it is not possible to run expressions containing wild cards such as `ls *.txt` since `*` will be interpreted literally.

Julia has a `readdir()` function, which will return a folder listing as an array of strings, so using the `wc` command to get a line/word/character count for the files in Alice:

```
julia> run('wc $(readdir())')  
      4      43      255 README.1st  
     92     502    2593 aged-aged-man.txt  
     38     280    1455 father-william.txt  
    734    4423   25225 hunting-the-snark.txt  
     33     168     968 jabberwocky.txt  
     19     221    1182 lobster-quadrille.txt  
     68     331    1887 mad-gardeners-song.txt  
     16     157     813 voice-of-the-lobster.txt  
    124     626    3415 walrus-and-carpenter.txt  
  1128    6751   37793 total
```

Most of the files have the `.txt` extension, but there is a `README` file, and later, we will be producing other files as a result of some text processing for which I will deliberately use a different extension.

In order to work on the contents of the poems alone, we will need to filter the list by applying a regular expression-based pattern match.

```
function filter(pat::Regex, dir=".")  
    a = []  
    for f1 in readdir(dir)  
        ismatch(pat, f1) && push!(a, f1)  
    end  
    a  
end
```

This function is a very simple wrapper around `readdir()`, which grabs the listing and tests it for a match against a regex (regular) expression. Regex was introduced in *Chapter 2, Developing in Julia*, and you might recall that these are strings delimited by `r"..."`.

Using our filter function, we can pick out all the titles with the word `the` in them. Note that we will also get `father-william` as it has `the` in `father`:

```
julia> run('wc $(filter(r"the"))')
 38      280    1455 father-william.txt
 734     4423   25225 hunting-the-snark.txt
   16      157     813 voice-of-the-lobster.txt
 788     4860   27493 total
```

However, we still have a problem using this (and `readdir()`) as the `wc` command will stall if the filter function does not produce an output, so we need to check for this before running the command.

Looking for files beginning with `h`, we find one `.txt` file but no log file, so we use the following code:

```
aa = filter(r"^\h\.\log"); length(aa) > 0 && run('wc -l $(aa)')

aa = filter(r"^\h\.\txt"); length(aa) > 0 && run('wc -l $(aa)')
 734     4423   25225  hunting-the-snark.txt
 734     4423   25225  total
```

It is worthy to notice that there is a Unix command that will handle wildcard `find`. This is because `find` mimics the shell rather than utilizing it. The wildcard `*` needs to be preceded by a backslash; otherwise, it is interpreted literally:

```
julia> run('find "/Users/Malcolm" -name \*.1st';
/Users/Malcolm/Alice/README.1st
```

This can be very useful as it produces a recursive list of fully qualified files. Further, `find` has a great number of other options to enhance the file search; see the Unix manual for a description of the full syntax.

Redirection and pipes

A consequence of Julia not using the Unix shell for its command processing is that the conventional chaining of commands using pipes will not work.

Instead, Julia provides its own pipe operator (`|>`) to chain *separate* Julia command objects:

```
julia> run('echo "Don\'t Panic!" |> 'rev')
!cinap t'noD
```



The pipe operator has been deprecated as of version v0.4, and while continuing to be valid, it will produce deprecation warnings. We are working with the current stable version v0.3.x and will use this notation, but the alternate accepted form for future versions will now be the following use of the pipe() function:

```
julia> run(pipe('echo "Don't Panic!"', 'rev'))
```

Assuming that we are in the `Alice` folder, the following searches (using `grep`) for all lines containing the word beaver in "The Hunting of the Snark":

```
julia> pwd(); # => "/Users/malcolm/Alice"
julia> wc(f) = isfile(f) && run('wc $f');
julia> txtfile = "hunting-the-snark.txt";
julia> logfile = "hunting-the-snark.log";
julia> run('grep -i beaver $txtfile' |> logfile);
julia> wc(logfile)
19          138          821  hunting-the-snark.log
```

- The `grep -i` command enforces a case-insensitive search
- The `wc()` function just encapsulates the Unix `wc` command for a given file, checking first whether the file exists
- If the command chain is terminated by a string, Julia will write the output to a file of that name; otherwise, to `STDOUT`

Note that using `|>` in this fashion will always create a new file. To append the output to an existing file, the `>>` operator is used as follows:

```
julia> run('grep -i bellman $txtfile' >> logfile);
julia> wc(logfile)
49          371          2022  hunting-the-snark.log
```

Therefore, to check for lines containing both `Bellman` and `Beaver`, we can double `grep` through a pipe as follows:

```
julia> run('grep -i bellman snark.log' |> 'grep -i beaver')
He could only kill Beavers. The Bellman looked scared,
He could only kill Beavers. The Bellman looked scared,
```

That the line occurs twice is confirmation that `>>` has appended to the file. It was added once when using `grep` for `beaver` and again when using `grep` for `bellman`.

Perl one-liners

As a devotee of Perl, I still find it one of the best languages for munging data. Perl is famous (among other things) for what can be achieved on a single line so much so that books are devoted just to Perl one-liners.

The following command reverses the lines in a file and applying it to the Jabberwocky:

```
julia> cp0 = 'perl -e "print reverse <>"';
julia> run('cat jabberwocky.txt' |> cp0
|> 'tee jabberwocky.rev' |> 'head -4')
And the mome raths outgrabe
All mimsy were the borogoves,
Did gyre and gimble in the wabe:
'Twas brillig, and the slithy toves
```

This creates a file on disk (via `tee`) and displays the first four lines of the output (that is, the last four lines of the poem reversed).

```
julia> run('find . -name \*.rev');
./jabberwocky.rev
```

Use of Perl's one-liners adds considerable text processing capabilities to Julia without exiting the language.

```
tfl = "hunting-the-snark.txt";
cp1 = 'perl -pne "tr/[A-Z]/[a-z]/' $(tfl)';
cp2 = 'perl -ne "print join("\n",
    split(/ /,$_)); print("\n")"';
run(cp1 |> cp2 |>'sort' |>'uniq -c' |>'sort -rn' |>'head -8')
889
299 the
153 and
123 a
105 to
91 it
82 with
76 of
```

This lists counts the words (case insensitive) listed in a descending numerical order. Not surprisingly <space> ranks the highest, followed by common words such as the and and.

As with all text, processing punctuation does present its problems; for example, looking for the number of times bellman appears in the Jabberwocky gives the following:

```
run(cp1|> cp2|>'sort'|>'uniq -c'|>'sort -rn'|>'grep bellman')  
25 bellman  
 4 bellman,  
 1 bellman's
```

In general, shell commands can just be dropped into the Julia syntax, and the back tick will take care of it with few problems. Cases that we need to be aware of are the contexts in which special characters such as \$ and @ are used as these are interpreted differently by Julia and Perl; normally, this requires not getting single and double quotes confused.

To finish, here are a couple of examples of using "The Walrus and the Carpenter":

```
julia> tf1 = "walrus-and-carpenter.txt";  
julia> of1 = "walrus-and-carpenter.out"  
julia> run('wc $tf1' |> 'tee $of1')  
124      626      3291 walrus-and-carpenter.txt
```

We have created a file (using tee) of the line / word/character counts in the poem and can now define a command that will find all integer numbers and increment them by 1.

```
julia> cp0 = 'perl -pe 's/(\d+)/ 1 + $1 /ge' $of1'  
julia> run(cp0)  
125      627      3292 walrus-and-carpenter.txt
```

Finally, here is another approach to reverse the lines in a file, which we implemented earlier on the Jabberwocky by using a line-by-line read in an open-do-end block. This is a one-liner applied this time to the first verse (6 lines) of "Walrus and the Carpenter":

```
julia> cp1 = 'perl -alne 'print "@{[reverse @F]}"'  
'perl -alne 'print "@{[reverse @F]}"'  
julia> run('head -6 $tf1' |> cp1)
```

```
sea, the on shining was sun The
might: his all with Shining
make to best very his did He
-- bright and smooth billows The
was it because odd, was this And
night. the of middle The
```

Summary

This chapter marks the end of our overview of some of the more unusual features of the Julia language. We have discussed the way Julia can interface directly with routines written in C and Fortran and how it is possible to use modules from other programming languages such as Python, R, and Java.

Further, we looked at the homoiconity nature of Julia and how this leads to the ability to define genuine runtime macros.

Finally, we introduced the running of separate tasks and pipelining and how the use of macros creates boilerplate code to greatly simplify the execution of the code on parallel processors.

In the subsequent chapters, we will be looking at individual topics that are of particular interest to analysts and data scientists, with a discussion of the I/O system and how to process data stored in disk files.

5

Working with Data

In many of the examples we have looked at so far, the data is little artificial since it has been generated using random numbers. It is now time to look at data held in files.

In this chapter, I'm going to concentrate on simple disk-based files, and reserve discussions of data stored in databases and on the Web for later chapters. Also, we will look at some of the statistical approaches in Julia that are incorporated both into the Julia base and into some of the growing number of packages covering statistics and related topics.

Basic I/O

Julia views its data in terms of a byte stream. If the stream comes from a network socket or a pipe, it is essentially asynchronous but the programmer need not be aware of this as the stream will be blocked until cleared by an I/O operation.

The primitives are the `read()` and `write()` functions which deal with binary I/O. When dealing with formatted data, such as text, a number of other functions are layered on top of these.

Terminal I/O

All streams in Julia have at least a read and a write routine. These take a Julia stream object as their first argument. In common with most systems, Julia defines three special terminal streams: standard-in (`STDIN`), standard-out (`STDOUT`) and standard-error (`STDERR`). The first is read-only and the latter two, write-only.

The special streams are provided by calls to the Julia API (`libjulia`) that links to the underlying operating system. Their definitions -- in terms of `ccalls()` -- along with the functions to operate on them, are part of the `Base::streams.jl` file.

There are a number of other files in `Base` for the curious to look at, notable among them are `io.jl` and `fs.jl`. However, to the analyst all the inner workings of the I/O system are relatively straightforward and follow a normal pattern of POSIX-based operating systems.

Consider some simple output to `STDOUT`:

```
julia> write(STDOUT, "Help ME!!!\n")
Help ME!!!
11
```

`11` is the number of bytes written, including the newline character. Due to the asynchronous nature of the I/O, the REPL outputs the string before the number of characters that were written.

We have seen earlier that using the `print` function, rather than `write`, absorbs the byte count. Also, if not provided, the `STDOUT` stream is taken as default. So `print(STDOUT, "Help ME!!!\n")` and `print("Help ME!!!\n")` will both produce identical output. Also, conveniently the `println()` function will append more.

To format output, we saw that Julia employs the macro system for generating the boilerplate. In fact, the whole system is so complex that it warrants its own module in `Base`, `printf.jl`. Also, included in the module is a `@sprintf` macro which will print formatted output to a string rather than to an output stream.



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.



Now, let's look at how to get input from `STDIN`. Julia uses the `read()` function that takes two arguments. The Julia stream object as the first argument and the second being the datatype to input:

```
julia> read(STDIN,Char)  # => Typing a return (®) gives 'a'
```

But all is not immediately obvious, try for instance:

```
julia> read(STDIN,Int32)  # => Typing 1234®  gives 875770417
```

Although this seems bizarre, the behavior is easily explained as:

```
julia> read(STDIN,UInt32) # => Typing 1234 ®  gives 0x34333231
```

So the result (`875770417`) is the ASCII representation of `1234` in reverse order!

To get the expected behavior, we need to use `readline()` instead of `read()`. This will consume the entire line and give the output as a string in the correct order:

```
julia> a = readline() # => Typing 1234 ⚡ gives a = "1234\n"
```

A couple of points to note are that `readline` includes the final return character, so to strip it off use the `chomp(readline())` construct.

Also, the routine returns a string where we probably expected an integer. The easiest way to deal with this is by just casting (converting) the string to the required datatype, in this case: `int(chomp(readline()))`.

Of course, the conversion will fail unless the input is parsed correctly as an integer.

This will raise an exception error, unless trapped with a `try/catch` block. So a function such as the following would clean up the input:

```
function getInt()
    a = chomp(readline())
    try
        return int(a)
    catch
        return Nothing
    end
end
```

Finally, it is possible to use the `readbytes(STDIN,N)` function to read `N` bytes into a `UInt8` array as:

```
julia> readbytes(STDIN,4)
abcd
#=> 4-element Array{UInt8,1}: [0x61, 0x62, 0x63, 0x64]
```

Disk files

Files on disk are accessed or created using the `open()` function. There are various flavors of `open` statements that can be listed using the `methods(open)` command. Here, I'll deal with the usual type where a file name is passed as an argument to `open()`, these are to be found in `iostream.jl`:

```
open(fname::String)
open(fname::String [,r::Bool,w::Bool,c::Bool,t::Bool,f::Bool])
open(fname::String, mode::String)
```

In all cases, `fname` is the name of the file to be opened. This may be relative to the current location or can be a fully qualified path name and successful opening of the file returns an `iostream` object.

The first variant is the most straightforward. This opens the file for reading and if the file does not exist, or the current user process can't access the file due to insufficient privileges, an error (trappable) is raised.

The second form consists of the file name followed by five optional boolean parameters. The first corresponds to opening the file for reading or writing, if both are true it is open for both reading and writing. The third parameter will specify that the file will be created, if it does *not* exist, and the fourth indicates that it will be truncated, if it *does* exist. The final parameter indicates that any writes will be appended to the end of the current file. When no flags are specified the default is `r`. If this is true and the rest are false, then that the file is opened read-only.

Not all of the combinations of the flags are logically consistent, so it is more common to employ the third form of open where the mode is passed as an ASCII string having the following possible values:

+-----+		+-----+
<code>r</code> <code>read</code>		
+-----+		+-----+
<code>r+</code> <code>read, write</code>		
+-----+		+-----+
<code>w</code> <code>write, create, truncate</code>		
+-----+		+-----+
<code>w+</code> <code>read, write, create, truncate</code>		
+-----+		+-----+
<code>a</code> <code>write, create, append</code>		
+-----+		+-----+
<code>a+</code> <code>read, write, create, append</code>		
+-----+		+-----+

These will be familiar to any Unix readers as they are the same as those used by the Unix standard library.

Since opening a file, doing something to its contents, and finally closing it again is a very common pattern, there is another option that takes a function as its first argument and filename as its second. This opens the file, calls the function with the file as an argument, applies the function to the file contents and then closes it again.

For example, suppose we take the file consisting of the first four lines of the Jabberwocky from the previous chapter in the /Users/malcolm/Alice/jabber4.txt file.

Then, we can capitalize the text using the following:

```
function capitalize(f::IOStream)
    return uppercase(readall(f))
end

open(capitalize, "/Users/malcolm/Alice/jabber4.txt")
"Twas brillig, and the slithy toves\nDid gyre and gimble in the wabe:\nAll mimsy were the borogoves,\nAnd the momeraths outgrabe.\n"
```

In this example, the entire file is input into a string using `readall()` and since `uppercase` will operate on the entire string, the function is pretty lightweight.

For more complex processing, we will open the file and work on individual parts of the file. So the example of reversing the lines of text, which we performed using Perl in the previous chapter, could be written natively in Julia as:

```
f = open("/Users/malcolm/Alice/jabber4.txt")
while (!eof(f))
    println(reverse(chomp(readline(f))))
end
close(f)
sevot yhtils eht dna ,gillirb sawT'
:ebaw eht elbmig dna eryg diD
,sevogorob eht erew ysmim lla
.ebargtuo shtar emom eht dnA
```

Notice that `readline()` again returns a string including a return character, but because we are reversing the line we need to strip it off and output the line with `println()`. Again, this `open/process/close` construct is so common that it can be written in an alternative form using a `do` block as:

```
open("/Users/malcolm/Alice/jabber4.txt") do f
    while (!eof(f))
        println(reverse(chomp(readline(f))))
    end
end
```

Text processing

Use of functions such as `read()`, `readline()`, and `readall()` provides us with the tools to implement text processing routines without having to resort to the sort of operating system tricks we employed in the previous chapter.

Consider the basis of the standard Hadoop example of counting the number of words in a file. We can start with a basic `word_count()` function that splits an arbitrary line of text in to individual words.

```
function wordcount(text)
    wds = split(lowercase(text),
    [ ' ', '\n', '\t', '-', '.', ',', ':', ';', '!', '?', '\'', '\"'];
    keep=false)
    d = Dict()
    for w = wds
        d[w] = get(d,w,0)+1
    end
    return d
end
```

This function converts the input text to lowercase so as not to differentiate between cases such as `The` and `the`. It splits the text on the basis of whitespace and various punctuation marks, and creates a `d` dictionary (hash) to hold the word count.

The `get()` function returns the value for the `w` word from the `d` dictionary or a 0 if the word is not yet present. After processing all the text, the function returns.

We can apply this to our `jabber4` file. Rather than using the full file specification, I've used the `getenv()` function we created earlier to get the `HOME` directory and changed it into the `Alice` subdirectory.

```
cd (getenv("HOME")**"/Alice");
open("jabber4.txt") do f
    c = wordcount(lowercase(readall(f)))
    print(c)
end
{ "'twas"=>1, "gyre"=>1, "and"=>3, "brillig"=>1, "raths"=>1, "in"=>1, "mome"=>1,
  "toves"=>1, "mimsy"=>1, "did"=>1, "the"=>4, "borogoves"=>1, "were"=>1, "all"=>1
, "wabe"=>1, "outgrabe"=>1, "slithy"=>1, "gimble"=>1}
```

Coupled with the `readdir()` function, it is possible to work on collections of files in a specific directory on the filesystem. Combine this with regex style pattern matching and all processing of text can be coded natively in Julia.

So if we wish to compute the word count for all the files with `.txt` extension in the Alice directory, we can refine the word count routine to take a dictionary and return the number of words processed as:

```
function wordcount(dcnts, text)
    total = 0
    words = split(lowercase(text),
                  [' ', '\n', '\t', '-', '.', ',', ':', ';', '!', '?', '\'', '\"', "'('', ')'"'],
                  keep=false)
    for w = words
        dcnts[w] = get(dcnts, w, 0) + 1
        total += 1
    end
    return total
end
```

Applying this to all the text (`.txt`) files in the directory is relatively easy:

```
d = Dict();
for fname in filter!(r"\.txt$", readdir())
    open(fname) do f
        n = wordcount(d, readall(f))
        @printf "%s: %d\n" fname n
    end
end
```

Running the preceding code gives the following output:

```
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  
voice-of-the-lobster.txt: 158  
walrus-and-carpenter.txt: 623
```

The highest word count is for the stop words such as: the, and, a.

More interestingly, we can see the number of times a particular word, such as "snark", occurs from the following:

```
wd = ["bellman", "boots", "gardener", "jabberwock", "snark"];  
for w in wd  
    @printf "%12s => %4d\n" w d[w]  
end  
    bellman # => 30  
    boots # => 3  
    gardener # => 2  
    jabberwock # => 3  
    snark # => 32
```

Binary files

Julia can handle binary files as easily as text files using `readbytes()` and `write()`.

In the first chapter, we created a simple grayscale image for a Julia set. It's possible to add some color using the following algorithm for each pixel value:

```
function pseudocolor(pix)  
    if pix < 64  
        pr = uint8(0)  
        pg = uint8(0)  
        pb = uint8(4*pix)  
    elseif pix < 128  
        pr = uint8(0)  
        pg = uint8(4*(pix-64))  
        pb = uint8(255)  
    elseif pix < 192  
        pr = uint8(0)  
        pg = uint8(255)
```

```
    pb = uint8(4*(192 - pix))
else
    pr = uint8(4*(pix - 192))
    pg = uint8(4*(256 - pix))
    pb = uint8(0)
end
return (pr, pg, pb)
end
```

Recall that the magic number of a binary colored NetPBM image is "P6":

```
img = open("juliaset.pgm");
magic = chomp(readline(img));      # => "P5"
if magic == "P5"
    out = open("jsetcolor.ppm", "w");
    println(out, "P6");
    params = chomp(readline(img));  # => "800 400 255"
    println(out, params);
    (wd, ht, pmax) = int(split(params));
    for i = 1:ht
        buf = readbytes(img, wd);
        for j = 1:wd
            (r,g,b) = pseudocolor(buf[j]);
            write(out,r); write(out,g); write(out,b);
        end
    end
    close(out);
else
    error("Not a NetPBM grayscale file")
end
close(img);
```

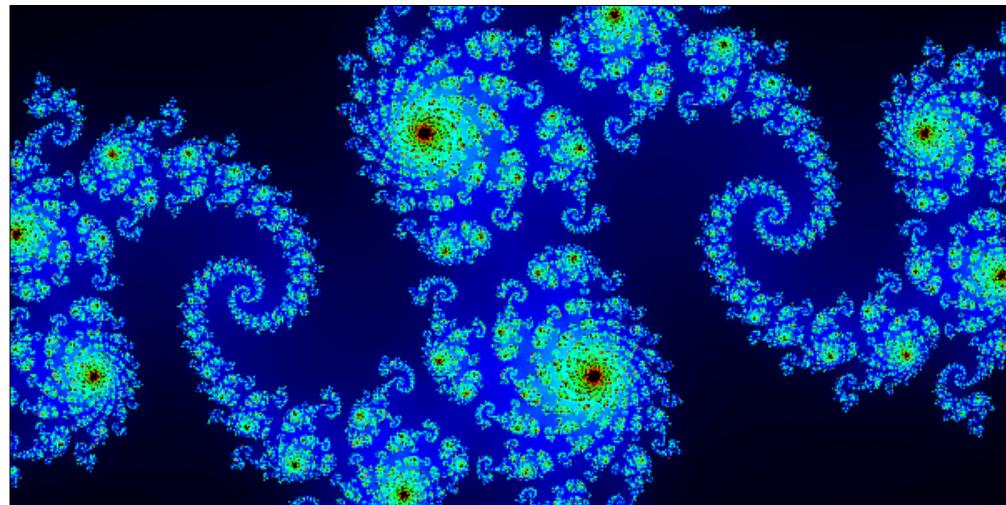
We use `readline()` to grab the magic number and check that it is a PGM file. In this case, we open a file to hold the colored image and write "P6", the PPM magic number.

The next line has the width, height and maximum pixel values. Because we are keeping byte values, we can write this back as it is. Also, we will need to decode the width and height in order to process all the pixels in the image.

I have applied the `pseudocolor()` function on a line-by-line basis.

This could have been easily done by reading individual bytes or grabbing all the bytes with a single `readbytes(wd*ht)` call. The latter has the merit of reducing file read/writes since the image will easily fit into available memory.

The result can be displayed using an image processing program that can read NetPBM images:



Structured datasets

In this section, we will look at files that contain metadata, the way that data is arranged as well as the values themselves. This includes simple delimited files and also files with additional metadata such as XML and HDF5.

Also, we will introduce the important topic of data arrays and frames in Julia, which is familiar to all R users and also implemented in Python via the `pandas` module.

CSV and 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. Columns in the table are consistent, in the sense that they may be all integers, floats, dates, and so on,, and are to be considered as the same class of data. This will be familiar to most as it maps similar to the way data is held in a spreadsheet.

One of the oldest forms of such representation of such data is the **Comma-Separated-Value (CSV)** file. This is essentially an ASCII file in which fields are separated by commas (,) and records (rows) by a newline.

There is an obvious problem if some of the fields are strings containing commas, so CSV files use quoted text (normally using a ") to overcome this. However, this gives rise to the new question of how to deal with text fields that contain a " character.

In fact, the CSV file was never defined as a standard and a variety of implementations exist. However, the principle is clear that we require a method to represented involving a field separator and a record separator together with a way to identify any cases where the field and record separators are to be interpreted as regular characters.

This generalization is termed a **delimited file (DLM)** and an alternative to the CSV file is a **tab-separated file (TSV)** where the comma is replaced by a tab.

Julia has a built-in function called `readdlm()` that can open and read an entire disk file. Its source can be found in `base/datafmt.jl`.

The general form of the call is:

```
readdlm(source, delim::Char, T::Type, eol::Char; opts... )
```

Each line; is separated by `eol` and the fields separated by `delim`. The source can be a text file, stream, or byte array. Memory-mapped files can be used by passing the byte array representation of the mapped segment as source.

If `T` is a numeric type, the result is an array of that type with any non-numeric elements as `NaN` for floating-point types, or zero; other values for `T` can be `ASCIIString`, `String`, and `Any`.

A simpler form of `readdlm(source; opt)` will assume a standard end-of-line separator ('`\n`') and that fields are delimited by whitespace. This can encompassed TSV files, if consisting of all numeric data, but if some of the fields are text containing spaces, it is necessary to specify `delim = '\t'` explicitly.

For comma-separated files, there is a separate `readcsv(source; opt...)` call where the delimiter is a comma (`delim = ', '`).

The optional arguments are as follows:

```
header=false, skipstart=0, skipblanks=true, use_mmap, ignore_invalid_
chars=false, quotes=true, dims, comments=true, comment_char='#'
```

- If `header` is true, the first row of data will be read as header and the `(data_cells, header_cells)` tuple is returned instead of only `data_cells`.
- Specifying `skipstart` will ignore the corresponding number of initial lines from the input; if `skipblanks` is true, blank lines in the input will be ignored.
- If `use_mmap` is true, the file specified by `source` is memory mapped for potential speedups. The default is true except on Windows, for which we may want to specify true if the file is large and has to be read only once and not written to.
- If `ignore_invalid_chars` is true, bytes in `source` with invalid character encoding will be ignored. Otherwise an error is thrown indicating the offending character position.
- If `quotes` is true, column enclosed within double quote ('') characters are allowed to contain new lines and column delimiters. Double quote characters within a quoted field must be escaped with another double quote.
- Specifying `dims` as a tuple of the expected rows and columns (including header, if any) may speed up reading of large files.
- If `comments` is true, lines beginning with `comment_char` and text following `comment_char` in any line are ignored.

In the `Work` directory, we have a CSV file of the Apple share prices from 2000 to 2002. This has a header line and six fields, and we can read it into an array as:

```
(aapl,cols) = readcsv("AAPL.csv"; header=true);
```

The `aapl` matrix contains the data and the `head` vector contains the column information:

```
julia> cols
1x13 Array{String,2}:
"Date" "Open" "High" "Low" "Close" "Volume"
"Ex-Dividend" "Split Ratio"
"Adj. Open" "Adj. High" "Adj. Low" "Adj. Close" "Adj. Volume"
```

We see that there are 13 fields per row comprising the date, opening, high, low, and closing prices, and the volume traded (both actual and adjusted) as well as the ex-dividend and split ratio values. Nominally these will be `0.0` and `1.0` respectively unless there is a day on which a dividend is paid or a rights issue in effect.

The data consists of a matrix of 752 rows and 13 columns:

```
julia> typeof(aapl)    # => Array{Any,2}
```

The data is in the form of an `{Any}` array; looking at the first 6 columns:

```
julia> aapl[:, 1:6]
"2002-12-31"  14.0   14.36  13.95  14.33  3584400
"2002-12-30"  14.08  14.15  13.84  14.07  2768600
"2002-12-27"  14.31  14.38  14.01  14.06  1429200
"2002-12-26"  14.42  14.81  14.28  14.4   1525400
"2002-12-24"  14.44  14.47  14.3   14.36  702500
"2002-12-23"  14.16  14.55  14.12  14.49  2246900
"2002-12-20"  14.29  14.56  13.78  14.14  5680300
"2002-12-19"  14.53  14.92  14.1   14.2   6250700
"2002-12-18"  14.8   14.86  14.5   14.57  2691100
```

Since the date is a non-numeric field, this is input as a string and results in the `{Any}` typing. Also, notice that the dates are in descending order.

So in order to get the closing prices, we will need to slice the matrix along the fifth column, convert to `float64`, and reverse the array order as:

```
aa_close = float64(reverse(aapl[:,5]));
```

Since the dates are not arranged linearly, due to weeks and bank holidays, we need to transform the dates as an offset from some reference date for which we will take the minimum date as `2000-01-01`.

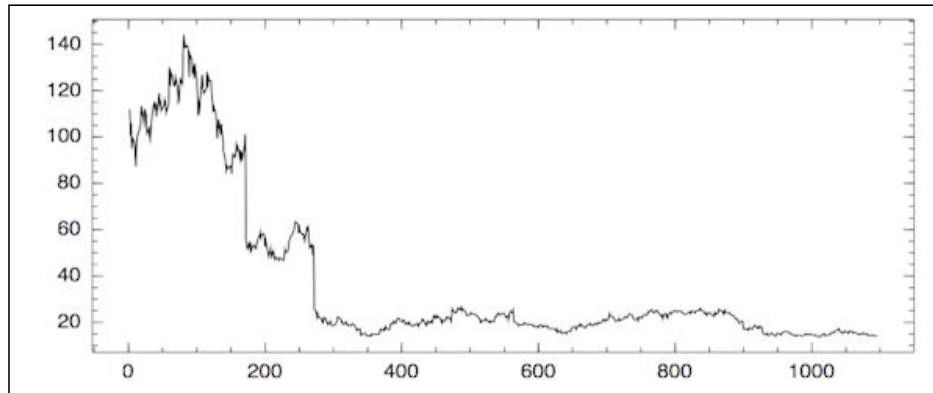
We can compute the days from this reference and use the indexing of the list comprehension to reverse the date order as:

```
using Date
d0 = Date("2000-01-01")
naapl = size(aa_date)[1]
aa_date = zeros(naapl)
```

```
[aa_date[1+n-i] = int(Date(aapl[i,1]) - d0) for i = 1:napp1]

using Winston
plot(aa_date, aa_close)
```

The resulting graph is shown in the following figure:



The first year (2000) was clearly a volatile one with a rapid decline in value, although the current value of Apple stocks is more than 500 percent of the peak price here.

On two occasions we can see a fall of over \$20 in a single day, so looking for these days:

```
for i in 2:(size(aapl)[1] - 1)
    if abs(aapl[i,5] - aapl[i-1,5]) > 20.0
        println(i+1, " : ", aapl[i+1,1:8])
        println(i, " : ", aapl[i,1:8])
        println(i-1, " : ", aapl[i-1,1:8])
    end
end
```

This code produces the following output:

```
635 : Any["2000-06-20" 98.5 103.94 98.38 101.25 4.4767e6 0.0 1.0]
634 : Any["2000-06-21" 50.5 56.94 50.31 55.62 8.75e6 0.0 2.0]
633 : Any["2000-06-22" 55.75 57.62 53.56 53.75 8.352e6 0.0 1.0]
```

```
566 : Any["2000-09-27" 51.75 52.75 48.25 48.94 7.1832e6 0.0 1.0]
565 : Any["2000-09-28" 49.31 53.81 48.12 53.5 1.74926e7 0.0 1.0]
564 : Any["2000-09-29" 28.19 29.0 25.38 25.75 1.325293e8 0.0 1.0]
```

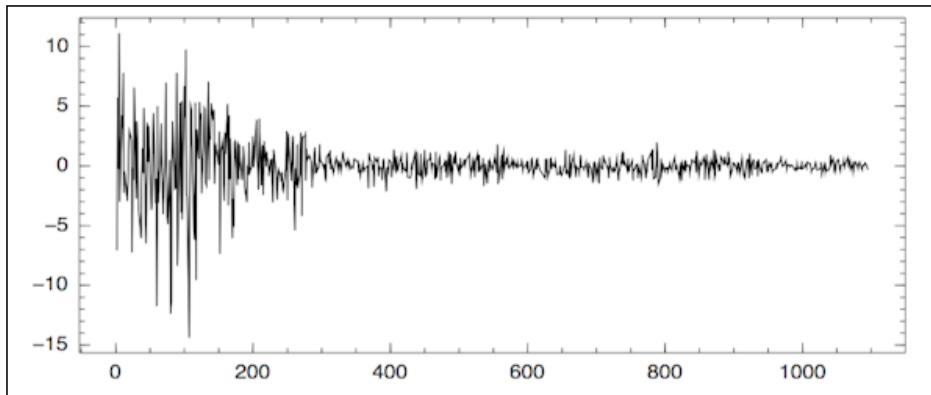
The stock price from close of 20/06/2000 to open of 21/06/2000 has roughly halved but this is explained by the Split Ratio = 2.0; so in fact there was a 2 for 1 script issue following the close of 20/06/2000.

For the second sharp fall on 28/09/2000, there is no obvious cause but there are very high volumes on this day and more so on the following day, indicating a rapid sell off of Apple stocks.

If we wish to look at the price spreads over individual days, we need to look at the differences between the opening and closing prices, which is easily achieved as:

```
aa_diff = float64(reverse(aapl[:,2] - aapl[:,5]));
plot(aa_date[1:250], aa_diff)
```

This plot is displayed in the following figure:



HDF5

HDF5 stands for **Hierarchical Data Format v5**. It was originally developed by the NCSA in the USA and is now supported by the HDF Group. It was developed to store large amount of scientific data that are exposed to a user as groups and datasets that are akin to directories and files in a conventional filesystem.

Version 5 was developed to overcome some of the limitations of the previous version (v4) and Julia, in common with languages such as Python, R, MATLAB/Octave, 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.

Language wrappers for HDF5 may be viewed as either low-level or high-level. The `hdf5.jl` Julia package 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 in order 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 to computing architectures.

Plain HDF5 files are created and/or opened in Julia with the `h5open` command:

```
fid = h5open(filename, mode)
```

In the preceding command, `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 `HDF5File` abstract type. *Plain* files have no elements (groups, datasets, or attributes) that are not explicitly created by the user.

The `HDF5.jl` package also provides a **Julia Data format (JLD)** to accurately store and retrieve Julia variables. It is possible to use plain HDF5 for this purpose, but the advantage of the JLD module is that it preserves the exact type information of each variable.

The Julia implementation of the HDF5 is very comprehensive and the reader is encouraged to look at the excellent online documentation on HDF and JLD in the `HDF5.jl` GitHub repository.

Here, I will concentrate on the JLD module. This is very convenient for interchanging the Julia data between sites.

As an example, we will store the `aa_date` APPL and the `aa_close` data that we created in the previous section as:

```
jldopen(/Users/malcolm/Work/aapl.jld, "w") do fid
    write(fid, "aa_date", aa_date)
    write fid "aa_close", aa_close)
end
close(fid)
```

This creates a `aapl.jld` file and adds datasets for `aa_date` and `aa_close`.

Alternatively, the same process can be achieved using:

```
save("/Users/malcolm/Work/aapl.jld",
    "aa_date", aa_date. "aa_close", aa_close)
```

There are also macro versions that assume the dataset name is the same as the variable such as:

```
@save "/Users/malcolm/Work/aapl.jld" aa_date aa_close
```

Reading back the data is relatively straightforward:

```
fid = jldopen("/Users/malcolm/Work/aapl.jld")
ac = read(fid, "aa_close")
ad = read(fid, "aa_date")
```

We can check that the re-read data matches the original:

```
typeof(ad); # => 752-element Array{Float64,1};

[assert(ad[i] == aa_date[i]) for i = 1:size(ad)[1]];
```

Because JLD is a special case of HDF5, we can use the routines from the latter module. For example, the structure of the file we can use `dump()`, the first few lines are the most enlightening:

```
dump(fid)
JldFile
plain: HDF5File len 3
    _require: HDF5Dataset () : UTF8String[]
    aa_close: HDF5Dataset (752,) : [111.94,102.5,104.0,95.0,
    aa_date: HDF5Dataset (752,) : [2.0,3.0,4.0,5.0,
```

In the previous example, we wrote the AAPL data in its own file. However, it is easy to create a file, and create an aapl group to hold the datasets:

```
fid = jldopen("/Users/malcolm/Work/mydata.jld", "w")
g = g_create(fid, "aapl")
g["aa_date"] = aa_date
g["aa_close"] = aa_close
```

We can check the structure has changed as:

```
dump(fid)
JldFile
    plain: HDF5File len 2
        _require: HDF5Dataset () : UTF8String[]
    aapl: HDF5Group len 2
        aa_close: HDF5Dataset (752,) : [111.94,102.5,104.0,95.0,
        aa_date: HDF5Dataset (752,) : [2.0,3.0,4.0,5.0,
```

The datasets can be retrieved by either of the following:

```
g = fid["aapl"]; ad = g["aa_date"]
ad = fid["aapl/aa_date"]
```

XML files

Alternative data representations are provided using XML and JSON. We will consider data handling again in *Chapter 8, Databases* and *Chapter 9, Networking*, when we look at JSON and discuss networking, web, and REST services.

In this section, we will look at XML file handling and the functionality available in the `LightXML` package.

To assist we will use the `books.xml` file, which contains a list of ten books. The first portion of the file is:

```
<?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 covering the most functionalities:

- Parse a XML file or string into a tree
- Access XML tree structure
- Create an XML tree
- Export an XML tree to a string or an XML file

Here, I will cover parsing an XML file as it is probably the most common procedure:

```
cd(); cd ("Work")
using LightXML
xdoc = parse_file("books.xml");
xtop = root(xdoc);
println(name(xtop)); # => catalog
```

To read the file into memory, we use the `parse_file()` function. To traverse the entire tree, we start by getting a reference to the top element: `catalog`. The following finds all the element nodes and the `title` child element. This element has an attribute corresponding to the `genre` element and the code prints out all ten books and their genre.

```
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 "%20s :%s\n" 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 Programming Bible : Computing
```

Finally, we look for all the computing books and print out the full details. The publication date is in the format YYYY-MM-DD, so I've used the `Dates` package to create a more readable string.

Notice that, as I have pointed out earlier, this book is using the current stable version of Julia (v0.3.x). There is an important change in v0.4, here the `Dates` package is incorporated into the `Base` and this is likely to be the stable version at the time of reading.

```
using Dates
for c in child_nodes(xtop)
    if is_elementnode(c)
        e = XML_ELEMENT(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: 44.95 (GDP)
An in-depth look at creating applications with XML.

Title: .NET Programming Bible
Author: O'Brien, Tim
Date: 9 December 2000
Price: 36.95 (GBP)

Microsoft's .NET initiative is explored in detail in this deep programmer's reference.

DataFrames and RDatasets

When dealing with tabulated datasets there are occasions when some of the values are missing. One of the features of statistical languages is that they can handle such situations.

In Julia, the `DataFrames` package has been developed in order to treat such cases and this is the subject of this chapter.

The `DataFrames` package

The package extends the Julia base by adding three new types:

- `NA` is introduced in order to represent a missing value. This type only has one particular value `NA`.
- `DataArray` is a type that emulates Julia's standard `Array` type, but is able to store missing values in the array.
- `DataFrame` is a type that is capable of representing tabular datasets such as those found in typical databases or spreadsheets. The concept of the data frame is most evident in R language and is one of the cornerstones of its popularity.

Except for its ability to store `NA` values, the `DataArray` type is meant to behave exactly as Julia's standard `Array` type. In particular, `DataArray` provides two type aliases called `DataVector` and `DataMatrix` that mimic the `Vector` and `Matrix` type aliases for 1D and 2D array types.

The `DataArray` is included as part of the `DataFrames` package, so in order to introduce it and the `NA` type we type: `using DataFrames`.

Creating a data array is via the `@data` macro:

```
using DataFrames  
da = @data([NA, 3.1, 2.3, 5.7, NA, 4.4])
```

This creates a 6-element `DataArray` of the `Float64` type with the first and fifth elements being `NA`.

If we take a simple statistical metric such as the `mean(da)` command, this gives a `NA` value.

In order to get a meaningful value for the mean, we need to ignore (`drop`) the `NA` values:

```
mean(dropna(da)) ; # => 3.875
```

This effectively sums up all non-`NA` values, and computes the mean of 4 values and not 6. The `dropna()` function can also be used to convert the data array to a regular array as:

```
aa = convert(Array, dropna(da))  
4-element Array{Float64, 1} #=> [3.1, 2.3, 5.7, 4.4]
```

It is also possible to substitute missing (`NA`) values for a default value:

```
aa = array(da, 0)  
4-element Array{Float64, 1} #=> [0.0, 3.1, 2.3, 5.7, 0.0, 4.4]  
  
mean(aa); # => 2.5833
```

The mean value is now different as we have 6 elements rather than 4.

DataFrames

`NA` and `DataArray` provide mechanisms for handling missing values for scalar types and arrays, but most real-world datasets have a tabular structure that do not correspond to a simple `DataArray` type.

Columns of tabular datasets may be of different types, normally scalars such as lengths integers or reals, strings or booleans (such as `Yes/No`). Also they may be values within enumerated set.

However, apart from the possibility of absence of a value in any particular position in a column all values will be of the same type and the columns will be of equal length.

This description will be familiar to anyone who has worked with R's `dataframe` type, Pandas' `DataFrame` type, an SQL-style database, or Excel spreadsheet.

We can create a data frame consisting of two columns: `x` being linear spaced between 0 and 10, and `y` being an equivalent number of normally distributed values. This can be done as:

```
df = DataFrame(X = linspace(0,10,101), Y = randn(101))  
101x2 DataFrame
```

Row	X	Y
1	0.0	-0.0929538
2	0.1	0.56163
3	0.2	-1.23955
4	0.3	-1.19571
5	0.4	-0.66432

The names of the columns are now part of the `DataArray` type structure and so we use the symbol `(:Y)` in the `df[(:,Y)]` notation as a shorthand for `df[:, 2]`.

Recalling that this is a set of randomly spread data values, we can look at the basic statistic metrics of mean, standard deviation, median and covariance:

```
mean(df[(:,Y)]) ; # => 0.0568417959105659 (expected 0.0)  
var(df[(:,Y)]) ; # => 0.9609662097467784 (expected 1.0)
```

The package has a `readtable()` function similar to `readcsv()` we saw earlier, but rather than returning any existing header row as part of the data or as a separate vector it is now part of the data frame structure.

There is also a `writetable()` function to export data to a file, typically:

```
writetable(fname, df; opts...)
```

`readtable()` and `writetable()` take a pragmatic view to opening the file; if the extension is `csv`, the file type is assumed to be comma-separated, if it's `tsv` then tab-separated, and if `wsv`, it is white-space separated.

Otherwise, it is possible to use the optional argument (`separator =`) to indicate the field separator character.

There are a number of other options, a useful one is `header =` that takes the `false` value if there is no header row and `true` (default) otherwise.

Using our APPLE share data: `aapl = readtable("AAPL.csv")`

752x13 DataFrame

(Looking at the first 6 columns):

Row	Date	Open	High	Low	Close	Volume
1	"2002-12-31"	14.0	14.36	13.95	14.33	3.5844e6
2	"2002-12-30"	14.08	14.15	13.84	14.07	2.7686e6
3	"2002-12-27"	14.31	14.38	14.01	14.06	1.4292e6
4	"2002-12-26"	14.42	14.81	14.28	14.4	1.5254e6
5	"2002-12-24"	14.44	14.47	14.3	14.36	702500.0

The base provides a few basic statistical functions, such as the `mean()` used previously. Tying these all together to define a `description()` routine, we have:

```
function describe(aa)
    println("Mean      : " , mean(aa))
    println("Std. Devn. : " , std(aa))
    println("Median     : " , median(aa))
    println("Mode       : " , mode(aa))
    println("Quantiles  : " , quantile(aa))
    println("# samples : " , length(aa))
end

julia> describe(aapl[:Close])
Mean      : 37.12550531914894
Std. Devn. : 34.118614874718254
Median     : 21.49
Mode       : 19.5
Quantiles  : [13.59,17.735,21.49,31.615,144.19]
```

We can calculate the correlation and covariance between opening and closing prices as:

```
cor(aapl[:Open], aapl[:Close]); # => 0.9982147046681757
cov(aapl[:Open], aapl[:Close]); # => 1162.2183466395873
```

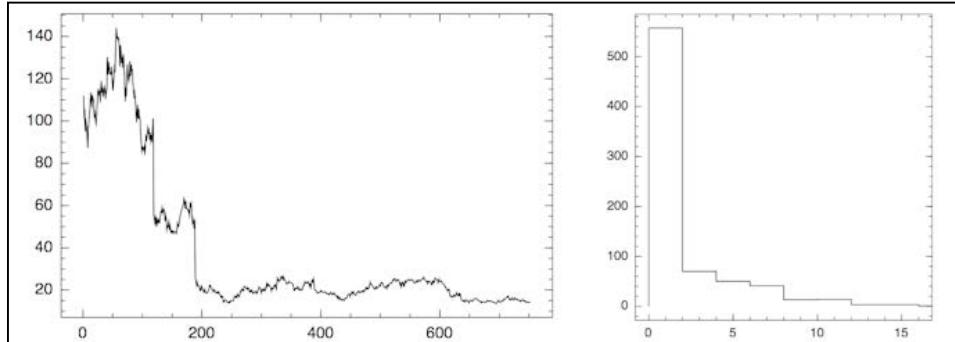
If we can create a vector of the daily spreads, using the `hist()` function computes an histogram with (default) 10 intervals:

```
spread = aapl[:High] - aapl[:Low];
hist(spread); # => (0.0:2.0:18.0, [557, 70, 50, 41, 13, 14, 3, 3, 1])
```

`hist()` returns a tuple of two elements, range, and frequency vector.

The following figure is a plot of the spreads and histogram. The histogram figure was created using Winston with the following code:

```
using Winston
p = FramedPlot()
add(p, Histogram(hist(spread) ...))
```



RDatasets

The Julia `RDatasets` package is a part of the `Rdatasets` *repo* authored by Vincent Arelbundock. These are taken from many standard R packages and are located on GitHub at <https://github.com/vincentarelbundock/Rdatasets>.

Installing `RDatasets.jl` places all the datasets in a number of subdirectories that are under a data directory, which we refer to as data group. The individual data is provided as either compressed CSV (`.csv.gz`) or R datafile (`.rda`) format.

There are over 700 packages in 34 groups. A full list of the groups can be found at <https://github.com/johnmyleswhite/RDatasets.jl>.

In order to load any of the datasets, the `DataFrames` package needs to be installed.

To load data we use the `dataset()` function, which takes two arguments, the name of the data group and the specific dataset, and returns a data frame of the underlying data:

```
using DataFrames, RDatasets  
mlmf = dataset("mlmRev", "Gcsemv")
```

`mlmRev` is a group of datasets from multilevel software review and `Gcsemv` refers to GSCE exam scores.

A complete list of all datasets can be obtained by `RDatasets.available_datasets()`.

For an individual group such as `mlmRev`, the related data is listed with `RDatasets.datasets("mlmRev")`.

Subsetting, sorting, and joining data

We will look at the `Gcsemv` dataset in the `mlmRev` group.

This covers the GCSE results from 73 schools both in examination and course work. The data is not split by subject (only school and pupil) but the gender of the student is provided:

```
julia> mlfm = dataset("mlmRev", "Gcsemv");  
julia> summary(mlfm); # => "1905x5 DataFrame"  
julia> head(mlfm)
```

Row	School	Student	Gender	Written	Course
1	"20920"	"16"	"M"	23.0	NA
2	"20920"	"25"	"F"	NA	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
6	"20920"	"62"	"F"	36.0	NA

There are 5 columns, the school code, student code, gender, and the written and course work percentages.

Notice that some of the data values are marked as NA (not available). We can select the data for a particular school as:

```
julia> mlmf[mlmf[:School] .== "68207", :]  
5x5 DataFrame  
| Row | School | Student | Gender | Written | Course |  
| ---- | ----- | ----- | ----- | ----- | ----- |  
| 1 | "68207" | "7" | "F" | 32.0 | 58.3 |  
| 2 | "68207" | "84" | "F" | 25.0 | 41.6 |  
| 3 | "68207" | "101" | "F" | 23.0 | 62.9 |  
| 4 | "68207" | "126" | "F" | 26.0 | 57.4 |  
| 5 | "68207" | "167" | "M" | NA | 50.0 |
```

We can group the data by school as:

```
julia> groupby(mlmf, :School);  
GroupedDataFrame 73 groups with keys: [:School]
```

So let's list all schools with results from more than 50 students and compute the mean examination and course work scores:

```
for subdf in groupby(mlmf, :School)  
    (size(subdf)[1] > 40) &&  
        @printf "%6s : %4d : %.2f : %.2f\n" subdf[:School][1] size(subdf)[1]  
        mean(dropna(subdf[:Written])) mean(dropna(subdf[:Course]))  
end  
  
22520 : 65 : 35.84 : 56.45  
60457 : 54 : 53.34 : 85.61  
68107 : 79 : 44.41 : 74.19  
68137 : 104 : 28.92 : 62.62  
68321 : 52 : 52.00 : 78.64  
68411 : 84 : 40.96 : 59.21  
68809 : 73 : 42.68 : 70.98
```

We can sort the dataset in ascending examination scores:

```
julia> sort!(mlmf, cols=[:Written])
1905x5 DataFrame
| Row | School | Student | Gender | Written | Course |
| ---- | ----- | ----- | ----- | ----- | ----- |
| 1 | "22710" | "77" | "F" | 0.6 | 41.6 |
| 2 | "68137" | "65" | "F" | 2.5 | 50.0 |
| 3 | "22520" | "115" | "M" | 3.1 | 9.25 |
| 4 | "68137" | "80" | "F" | 4.3 | 50.9 |
| 5 | "68137" | "79" | "F" | 7.5 | 27.7 |
| 6 | "22710" | "57" | "F" | 11.0 | 73.1 |
```

We can see that, in general, marks for course work seem higher than for examination.

So computing the differences and some basic metrics:

```
diff = dropna(mlmf[:Course] - mlfmf[:Written]);
mean(diff) is 26.9 with SD of 15.58.
minimum(diff) is -43.5 and maximum(diff) is 76.0.
```

A histogram tells us more:

```
hh = hist(diff)
(-50.0:10.0:80.0, [2,1,6,16,53,125,251,408,361,211,78,10,1])
```

hh is a tuple where hh[1] is the range and hh[2] is an array containing the frequency counts. There are 5 bins for negative and 8 for positive ones.

```
julia> sum(hh[2][1:5]) ; # => 78
julia> sum(hh[2][6:13]); # => 1445
```

So in only 78 cases (out of 1523) is the examination mark greater than that of the course work.

Finally, let's compute the correlation coefficient between the two sets of marks.

We need to select only rows with no NA values, one method would be to use a logical condition such as:

```
!(isequal(mlmf[:Written][i], NA) || isequal(mlmf[:Course][i], NA))
```

However, there is an easier way using the `complete_cases()` function to create a subsetted data frame.

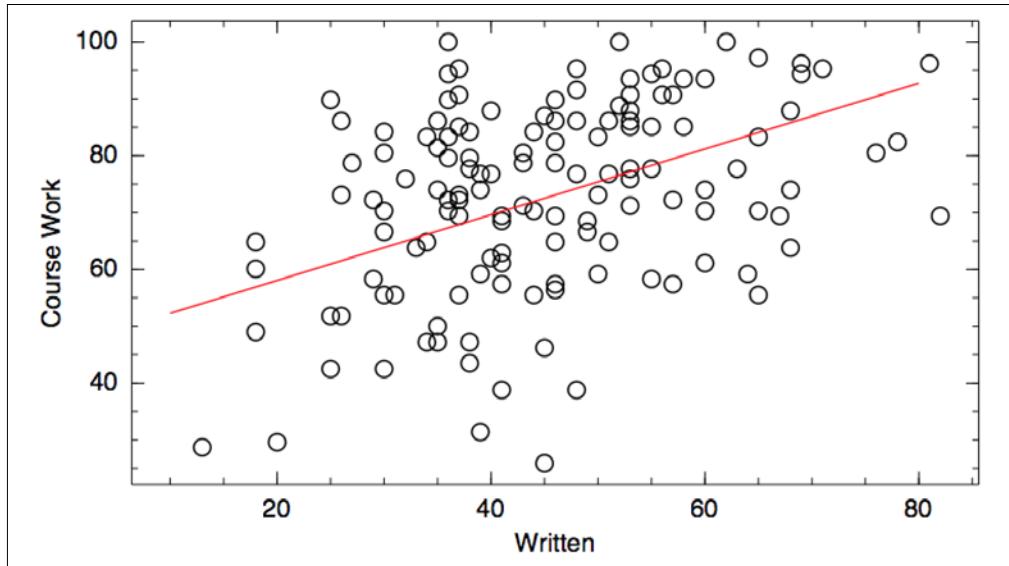
```
df = mlfmf[complete_cases(mlmf[[:Written, :Course]]), :]
cor(df[:Written], df[:Course]); => 0.4742
```

So the correlation between the written and course work is only 0.48.

The Julia base also has a `linreg()` linear regression function that will fit a least squares line through the data values:

```
linreg(array(df[:Written]), array(df[:Course]));
# => 2-element Array{Float64,1}: [ 17.9713, 0.388884 ]
```

The following figure is a scatterplot of the written and course work scores, together with the least squares linear regression line:



Statistics

The Julia community has begun to group packages and developers together where there is some significant common functionality. One of the most prolific is the JuliaStats group that, as the name suggests, is concerned with statistical matters.

Although Julia was initially conceived as a replacement for slow scientific languages such as MATLAB and its clones, the success of the language in a wide variety of disciplines, including statistics, makes it attractive for a much wider audience.

JuliaStats has its own web page on GitHub (<https://github.com/JuliaStats>) that list links to various packages. The group is concerned with machine learning as well as basic and advanced statistics. Many packages provide documentation at <http://readthedocs.org> in addition to the markdown README file in the package itself.

In this section, I will be looking at some of the features provided by the Julia base and in StatsBase.jl, the full documentation of the latter is at <http://statsbasejl.readthedocs.org>.

This enhances the base with a series of statistics-related mathematical and empirical functions and covers the following topics:

- *Mean and Counting Functions*
- *Scalar and Summary Statistics*
- *Ranking and Correlation*
- *Sampling and Empirical Estimation*

Most of the popular pandas.py Python module is now covered by the combination of DataFrames and StatsBase and the goal of the group is to extend this to (at least) all of pandas.

There is also a Pandas.jl wrapper package that interfaces with the pandas API, which uses PyCall and so needs a working version of Python with pandas installed. I will look at this package briefly later.

There are many modules provided by R and it is possible to use Rif.jl to access these. Although, this is out of the scope of this book, the reader is directed to the GitHub sources for more details.

Simple statistics

We will return again to the Apple share prices from 2000 to 2002. In addition to the full CSV file, I have provided a short version with just the first 6 columns, that is, up to Volume.

Until now we have used simple statistics in Julia base, now we will investigate this data further.

```
using DataFrames, StatsBase  
aapl = readtable("AAPL-Short.csv");  
  
naapl = size(aapl)[1]  
m1 = int(mean((aapl[:Volume]))); # => 6306547
```

The data is read into a DataFrame and we can estimate the mean (`m1`) as before. For the volume, I've cast this as an integer as this makes more sense.

As we have seen, there are 3 years of data and we wish to give more data to the recent years than earlier ones. We can do this by creating a weighting vector:

```
using Match  
wts = zeros(naapl);  
for i in 1:naapl  
    dt = aapl[:Date][i]  
    wts[i] = @match dt begin  
        r"^\d{4}" => 1.0  
        r"^\d{4}\.\d{2}\.\d{2}" => 2.0  
        r"^\d{4}\.\d{2}\.\d{2}\.\d{2}" => 4.0  
        _ => 0.0  
    end  
end;  
  
wv = WeightVec(wts);  
m2 = int(mean(aapl[:Volume], wv)); # => 6012863
```

In addition, by computing weighted statistical metrics it is possible to trim off the outliers from each end of the data.

Returning to the closing prices:

```
mean(aapl[:Close]);           # => 37.1255  
mean(aapl[:Close], wv);      # => 26.9944  
trimmean(aapl[:Close], 0.1); # => 34.3951
```

`trimmean()` is the trimmed mean where 5 percent is taken from each end.

```
std(aapl[:Close]);           # => 34.1186  
skewness(aapl[:Close])      # => 1.6911  
kurtosis(aapl[:Close])      # => 1.3820
```

As well as second moments such as standard deviation, `StatsBase` provides a generic `moments()` function and specific instances based on this such as for skewness (third) and kurtosis (fourth).

It is also possible to provide some summary statistics:

```
summarystats(aapl[:Close])
```

```
Summary Stats:  
Mean:          37.125505  
Minimum:        13.590000  
1st Quartile:  17.735000  
Median:         21.490000  
3rd Quartile:  31.615000  
Maximum:        144.190000
```

The 1st and 3rd quartiles are related to 25 percent and 75 percent percentiles. For a finer granularity we can use the `percentile()` function:

```
percentile(aapl[:Close], 5);  # => 14.4855  
percentile(aapl[:Close], 95); # => 118.934
```

Samples and estimations

Let's now return to look at the GCSE scores we loaded from the `m1mRev` collection from `RDatasets`:

```
julia> using RDatasets
julia> m1mf = dataset("m1mRev", "Gcsemv");
```

Recall that we can reduce the data to cases where there are no available values (`NA`) as:

```
df = m1mf[complete_cases(m1mf[:Written, :Course]), :];
mean(df[:Written]); # => 46.5023
std(df[:Written]); # => 13.4775
sem(df[:Written]); # => 0.3454
sem() standard error of the mean.
```

Let's compute 10 samples, each with 100 data values, from the written scores and estimate it ourselves:

```
n = 10;
mm = zeros(n);
[mm[i] = mean(sample(df[:Written], 100)) for I on 1:10];
# => [47.67, 45.15, 47.48, 45.94, 47.49, 44.89, 45.71, 46.48, 45.47,
45.79]
mean(mm); # => 46.2062
std(mm); # => 1.020677
```

The mean value of the samples approaches the population mean of 46.5 and our estimate of the standard error is found by dividing the sample standard deviation by the square root of the number of samples:

```
std(mm) / sqrt(n); # => 0.3328
```

Pandas

As we stated earlier, much of functionality in the `pandas` Python module is now in `DataFrames` and `StatsBase` packages. However, calling Python via `PyCall` makes the entire module accessible from Julia.

Let's look at our reduced dataset of Apple share prices using Pandas:

```
julia> aapl = read_csv("AAPL-short.csv");
julia> aapl
      Date   Open   High    Low  Close  Volume
0  2002-12-31  14.00  14.36  13.95  14.33  3584400
1  2002-12-30  14.08  14.15  13.84  14.07  2768600
2  2002-12-27  14.31  14.38  14.01  14.06  1429200
3  2002-12-26  14.42  14.81  14.28  14.40  1525400
```

Notice that the row count starts at 0 rather than 1, this is because the Pandas module automatically switches between the zero-based indexing in Python and the 1-based indexing in Julia.

The other main difference is that the method calls in Python are again transformed to function calls in Julia:

```
mean(aapl[:Close]); # => 37.1255
std(aapl[:Close]); # => 34.1186
```

We can find the row values for the minimum and maximum closing prices as:

```
idxmin(aapl[:Close]); # => 57
aapl[:Close][57]; # => 13.59

idxmax(aapl[:Close]); # => 696
aapl[:Close][696]; # => 144.19
```

Also, we can use the pandas describe the entire dataset:

```
julia> describe(aapl)
      Open      High      Low     Close      Volume
count  752.00000  752.00000  752.00000  752.00000  7.520000e+02
mean   37.142168  38.252460  36.069840  37.125505  6.306547e+06
std    34.124977  35.307115  33.020107  34.118615  6.154990e+06
min    13.540000  13.850000  13.360000  13.590000  7.025000e+05 25%
17.730000 18.280000 17.250000 17.735000 3.793100e+06
50%    21.450000  22.030000  20.810000  21.490000  5.103250e+06
75%    32.532500  33.672500  30.382500  31.615000  7.165525e+06
max    142.440000 150.380000 140.000000 144.190000 1.325293e+08
```

Selected topics

The JuliaStats group lists a variety of packages to tackle both statistics and machine learning.

Worthy of mention are:

- `MultivariateAnalysis.jl`: Data analysis, dimension reduction and factor analysis
- `Clustering.jl`: A set of algorithms for data clustering
- `MCMC.jl`: Monte Carlo Markov Chain algorithms

I am going to conclude this chapter by looking briefly at four other packages, but a reader who is new to Julia is encouraged to look at the JuliaStats' GitHub site for more information, and download the packages exploring the examples and tests provided.

Time series

Time series are often used in analysis of financial data where we wish to even out daily market fluctuations to view underlying trends.

For this section, I will return again to Apple share prices between 2000 and 2002 and will use the *short* CSV, which comprises the first six fields, that is, upto the nonadjusted trading volumes.

One of the problems we encountered previously while working with the dates was that they were returned as strings and the dataset was presented in descending date order. So we needed to compute a day difference and reverse the order of underlying data.

The `TimeSeries` data structure (defined in `TimeSeries.jl`) solves this problem by encapsulating the dataset in separate parts:

```
timestamp::Union{Vector{Date}, Vector{DateTime}}
values::Array{T,N}
colnames::Vector{UTF8String}

• timestamp can be either Date or DateTime
• values holds the actual data
• colnames is a string vector of column names
```

`DateTime` is depreciated in favor of `Date` in v0.4. The `Date` type does not include `hours:minutes:seconds`, whereas `DateTime` does and both the `Date` and `DateTime` types are defined in the new `Dates` package incorporated in the base. While it is possible to set up a `TimeArray` using the constructor defined in the package, `TimeSeries` also provides a `readtimearray()` function that makes a good attempt at 'finding' the time data.

Note that the `TimeSeries` package supports both `date` types and is likely to do so in the foreseeable future:

```
using TimeSeries
cd(); cd("Work") ;

ta = readtimearray("AAPL-Short.csv")
752x5 TimeArray{Float64,2} 2000-01-03 to 2002-12-31
      Open     High      Low     Close    Volume
2000-01-03 | 104.88  112.5  101.69  111.94  4.7839e6
2000-01-04 | 108.25  110.62  101.19  102.5   4.5748e6
2000-01-05 | 103.75  110.56  103.0   104.0   6.9493e6
2000-01-06 | 106.12  107.0   95.0    95.0   6.8569e6

2002-12-26 | 14.42   14.81  14.28   14.4    1.5254e6
2002-12-27 | 14.31   14.38  14.01   14.06   1.4292e6
2002-12-30 | 14.08   14.15  13.84   14.07   2.7686e6
2002-12-31 | 14.0    14.36  13.95   14.33   3.5844e6
```

If you wish to look at current stock prices, these can be obtained from the `Quandl.jl` package using `aapl = quandl ("GOOG/NASDAQ_AAPL")`.

The date order and associate values are now in ascending order and the data values are now `Float64` (not `Any`). Also, the `show()` function for the time array gives the date range.

The `ta.colnames` parameter can also be listed using a convenient `colnames(ta)` function and likewise for the `timestamp` and `values` parameters:

```
julia> colnames(ta)
5-element Array{UTF8String,1}: "Open" "High" "Low" "Close" "Volume"
```

Let's look at the year 2000. We have to extract the data and create a new Time Array. To do this, we can use the `from()` and `to()` functions:

```
ta0 = from(to(ta,2000,12,31), 2000,1,1)
252x5 TimeArray{Float64,2} 2000-01-03 to 2000-12-29
```

This is now a 252 array containing just the values for the year 2000.

We can look at the lag (previous values) and lead (future values):

```
lag(ta0["Close"],30) [1]; # => 2000-02-15 | 111.94
lead(ta0["Close"],30) [1] # => 2000-01-03 | 119.0
```

The lag starts at 2000-02-15 as earlier values are outside the dataset. Note that these are 30 values not thirty days. Also, we may wish to apply a function to a series of values, and we can do this using `moving()`.

So to compute the moving average:

```
moving(ta0["Close"],mean,30)
223x1 TimeArray{Float64,1} 2000-02-14 to 2000-12-29
    Close
2000-02-14 | 105.24
2000-02-15 | 105.47
2000-02-16 | 105.86
2000-02-17 | 106.22
```

Rather than working with a fixed number of values, we may wish to take a period such as a calendar month:

```
collapse(ta["Close"], mean, period=month)
12x1 TimeArray{Float64,1} 2000-01-31 to 2002-12-31
    Close
2000-01-31 | 103.36
2000-02-29 | 111.64
2000-03-31 | 128.5
```

Also, we can compute the change from the previous values by looking for the maximum and taking absolute values to account for falls as well as rises:

```
maximum(abs(values(percentchange(ta0["Close"])))) ; # => 0.5187
```

Recall that this corresponds to the script issue of 2 for 1.

Finally, we may wish to when the closing price was above \$125:

```
for i in findall(ta["Close"] .> 125)
    ts    = ta0["Close"].timestamp[i]
    vals = ta0["Close"].values[i]
    println(ts, ":", vals)
end
```

```
2000-03-01: 130.31
2000-03-03: 128.0
2000-03-06: 125.69
2000-03-10: 125.75
2000-03-21: 134.94
2000-03-22: 144.19
```

Distributions

`Distributions.jl` is a package which covers a wide variety of probability distributions and associated functions, and also provides facilities to create other distributions.

- Moments (mean, variance, skewness, and kurtosis), entropy, and other properties
- Probability density/mass functions (`pdf`), and their logarithm (`logpdf`)
- Moment generating functions, and characteristic functions
- Sampling from population, or from a distribution
- Maximum likelihood estimation
- Posterior w.r.t. conjugate prior, and **Maximum-a-Posteriori (MAP)** estimation

As an example, let's look at Poisson distribution.

This models the occurrence of discrete events, such as goals in a football match or number of children in a family. It is completely described by a single parameter l and the probability that a variate takes a particular value k is given by:

$$\text{Poisson}(k; l) = \Pr(X = k) = l^k * \exp(-l) / \text{factorial}(k)$$

Since it is *well-known* that the average family comprises 2.2 children, we can produce the appropriate distribution function as:

```
using Distributions;
Poisson(2.2); # => Distributions.Poisson(l = 2.2 )

The Poisson distribution has mean and variance equal to 1, so:

mean(Poisson(2.2)); # => 2.2
std(Poisson(2.2)); # => 1.4832
median(Poisson(2.2)); # => 2.0

for i = 0:5
    pf = pdf(Poisson(2.2), i)
    cf = cdf(Poisson(2.2), i)
    @printf "%d : %7.4f %7.4f\n" i pf cf
end

0 : 0.1108 0.1108
1 : 0.2438 0.1106
2 : 0.2681 0.6227
3 : 0.1966 0.8194
4 : 0.1082 0.9275
5 : 0.0476 0.9751

rand(Poisson(2.2),15)
15-element Array{Int64,1}: [ 4 0 2 3 2 2 3 3 4 2 0 2 1 5 1 ]

fit(Poisson, rand(Poisson(2.2),1000))
Distributions.Poisson( lambda=2.262 )
```

Kernel density

Kernel density estimation (KDE) is a non-parametric way to estimate the probability density function of a random variable. KDE is a fundamental data smoothing problem where inferences about the population are made based on a finite data sample.

Kernel density overcomes the problem inherent with frequency histograms, that of selection of bins size. KDE may show substructures such as two (or more) maxima, which would not be apparent in a histogram.

Consider the GCSE written and coursework marks available from RDataSets:

```
using RDataSets, KernelDensity

mlmf = dataset("mlmRev", "Gcsemv");

df = mlmf[complete_cases(mlmf[[:Written, :Course]]), :];
dc = array(df[:Course]); kdc = kde(dc);
dw = array(df[:Written]); kdw = kde(dw);
```

In order to generate summary statistics, we need to eliminate any items that have a NaN value by indexing against the logical arrays created by `!isnan(dw)` and `!isnan(dc)`:

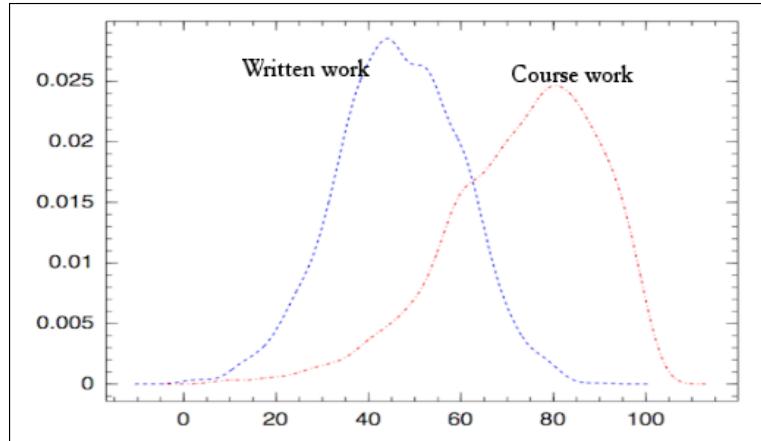
```
i.e ssw = summarystats(dw[!isnan(dw)])
and ssc = summarystats(dc[!isnan(dc)])
```

```
Summary Stats:
Written Course
Mean:        46.50    73.38
Minimum:      0.60    9.25
1st Quartile: 38.00   62.90
Median:       46.00   75.90
3rd Quartile: 56.00   86.10
Maximum:      90.00  100.00
```

We can display the kernel densities of the course work and written work as follows:

```
using Winston

kdc = kde(dc); kdw = kde(dw)
plot(kdc.x, kdc.density, "r--", kdw.x, kdw.density, "b;")
```



Hypothesis testing

A hypothesis test is a method of statistical inference on a dataset. A result is called significant if it has been predicted as unlikely to have occurred by chance, according to a pre-determined threshold probability termed the significance level.

Julia implements a wide range of hypothesis testing via the `HypothesisTests.jl` package that is well documented at <https://readthedocs.org/>.

We will look at GCSE scores that are subset by schools using the `groupby()` function, and we will look at schools that have at least 40 students:

```
for subdf in groupby(df, :School)
    (size(subdf) [1] > 40) &&
        @printf "%10s : %8.4f %8.4f\n" subdf[:School] [1]
        mean(subdf[:Written])  mean(subdf[:Course])
end
22520 : 35.4482 57.4580
60457 : 53.4773 85.9568
68107 : 44.9107 74.6750
68125 : 47.1556 77.5322
68137 : 28.2807 62.5373
68411 : 40.4615 59.4369
68809 : 42.7705 71.1115
```

We will create `DataFrames` for the schools 68107 and 68411 that have similar written scores but different ones for course work:

```
using HypothesisTests
df68107 = mlmf[mlmf[:School] .== "68107", :];
df68107 =
    df68107[complete_cases(df68107[[:Written, :Course]]), :];

df68411 = mlmf[mlmf[:School] .== "68411", :];
df68411 =
    df68411[complete_cases(df68411[[:Written, :Course]]), :];
```

We can test the hypothesis that the two distributions have same means with differing variances:

```
UnequalVarianceTTest(
    float(df68107[:Written]), float64(df68411[:Written]) )
```

```
Two sample t-test (unequal variance)
-----
Population details:
  parameter of interest: Mean difference
  value under h_0:          0
  point estimate:           4.4492
  95% confidence interval: (-0.1837, 9.0821)

Test summary:
  outcome with 95% confidence: fail to reject h_0
  two-sided p-value:          0.05963 (not significant)

Details:
  number of observations:   [56, 65]
  t-statistic:              1.9032531870995715
  degrees of freedom:       109.74148002018097
  empirical standard error: 2.337668920946911
```

Upon running the test we fail to reject the null hypothesis at the 95% significance level (note the double negative).

This means that the result could occur at least one in twenty times even if the values were taken at random from the same population dataset.

Now if we apply the same test to the coursework, we get the following:

```
UnequalVarianceTTest(  
    float(df68107[:Course]), float64(df68411[:Course]) )  
  
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: (10.6255, 19.8506)  
  
Test summary:  
    outcome with 95% confidence: reject h_0  
    two-sided p-value: 1.6304e-9 (extremely significant)  
  
Details:  
    number of observations: [56, 65]  
    t-statistic:            6.5420  
    degrees of freedom:     118.1318  
    empirical standard error: 2.3293
```

GLM

A **generalized linear model (GLM)** is a flexible extension of ordinary linear regression that allows response variables that have error distribution models other than a normal distribution.

GLM generalizes linear regression by allowing the linear model to be related to the response variable via a link function, and by allowing the magnitude of the variance of each measurement to be a function of its predicted value.

In Julia, the GLM package uses distributions to provide the link functionality.

Fitting may be by maximum likelihood or via a Bayesian method such as MCMC methods.

GLM has many methods for fitting; names are chosen to be similar to those in R:

- `coef`: Extract the estimates of the coefficients in the model
- `deviance`: Measure of the model fit, weighted residual sum of squares for `lm`'s
- `df_residual`: Degrees of freedom for residuals, when meaningful
- `glm`: Fit a generalized linear model (an alias for `fit(GeneralizedLinearModel, ...)`)
- `lm`: Fit a linear model (an alias for `fit(LinearModel, ...)`)
- `stderr`: Standard errors of the coefficients
- `vcov`: Estimated variance-covariance matrix of the coefficient estimates
- `predict`: Obtain predicted values of the dependent variable from the fitted model

To fit a GLM operates with `DataFrames` and uses:

```
glm(formula, data, family, link)
```

In the preceding command: `glm()` is a shortform for
`fit(GeneralizedLinearModel, ...)`.

- Formula is expressed in terms of the column symbols from the `DataFrame` data
- Data may contain `NA` values, any rows with `NA` values are ignored
- Family is chosen from `Binomial()`, `Gamma()`, `Normal()`, or `Poisson()`
- Link may be as `LogitLink()`, `InverseLink()`, `IdentityLink()`, `LogLink()`

Links are related to specific families, for example `LogitLink()` is a valid link for the `Binomial()` family.

We will just look at the `lm()` fit, a linear model, which is also a short form, this time for `fit(LinearModel, ...)` and apply this to the GSCE data (by school) from the previous section.

We will need to sample each dataset and sort the sample. Then, we will construct a `DataFrame` of corresponding low \leftrightarrow high pairs and see how the distribution of marks compares by looking at both written and course work marks.

The two schools we will look at are 68411 and 68107 that have similar written marks but different ones for coursework:

```
using GLM;

dw68411s = sort(sample(df68411[:Written], 50));
dw68107s = sort(sample(df68107[:Written], 50));
dc68411s = sort(sample(df68411[:Course], 50));
dc68107s = sort(sample(df68107[:Course], 50));

cor(dw68107s, dw68411s); # => 0.9894
cor(dc68107s, dc68411s); # => 0.8926
```

We can see that the correlation scores for both are good, for the given 50 samples, especially so for the written ones.

So constructing a couple of data frames:

```
dw = DataFrame(float64(dw68107s), float64(dw68411s));
names!(dw, [symbol("s68107"), symbol("s68411")]);
dc = DataFrame(float64(dc68107s), float64(dc68411s));
names!(dc, [symbol("s68107"), symbol("s68411")]);
```

I have renamed the column names to reflect the id of each school.

The model we are going to fit, for both the written (dw) and coursework (dc) dataframes, is a linear one as:

```
lm1 = fit(LinearModel, s68107 ~ s68411, dw)
DataFrameRegressionModel{LinearModel{DensePredQR{Float64}},Float64}:

Coefficients:
             Estimate Std.Error t value Pr(>|t|)
(Intercept) 0.12675  0.980441 0.129279   0.8977
s68411      1.15335  0.0244878 47.0988   <1e-41

i.e. s68107 ~ 0.12675 + 1.15335 * s68411
```

The fit has a low intercept and also a unity gradient reflecting the closeness of the two samples from the schools 68107 and 68411 for written scores.

Turning to the course work:

```
lm1 = fit(LinearModel, s68107 ~ s68411, dc)
DataFrameRegressionModel{LinearModel{DensePredQR{Float64}},Float64}:

Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)  28.0251   2.75815 10.1608 <1e-12
s68411       0.741007  0.0447073 16.5746 <1e-20

i.e.    s68107 ~ 28.0251 + 0.741007 * s68411
```

Here, the intercept is around 28 because of much higher marks for course work from s68107 to s68411. The gradient correspondingly lowers, implying that the low-end scores in s68107 are much higher than the high-end ones at s68411.

Summary

In this chapter, we looked at the terminal I/O in Julia, and also accessed text-based and binary disk files. We discussed how to access and work with data stored in a structured manner in files such as CSV, XML, and HDF5 and then introduced the important topic of data arrays and data frames.

We then continued with a review of some statistical methods applied to data frames available from the RDatasets package, including elementary hypothesis testing and general linear methods.

Statistics has been a particular success within the Julia community and the extremely productive group JuliaStats is an excellent reference for this work, with its own web pages at <https://github.com/JuliaStats> and a Google Groups forum **julia-stats**. Also, many of the projects are using <https://readthedocs.org/> to provide quite extensive documentation. For example, the guidance on the basic statistics can be found at <http://statsbasejl.readthedocs.org>.

In the next chapter, we will look at application of Julia to scientific problems. A dissatisfaction with existing programming languages, such as MATLAB and its clones, arguably formed the initial raison d'etre for the development of Julia and as we will see, it remains especially strong in this area too.

6

Scientific Programming

Julia was initially designed as a language directed to find solutions to the problems arising from science and mathematics. The current scripting languages then, and to some extent now, were slow especially when dealing with 'looping' code (devectorized) that is normally the algorithmic approach that the analyst will take in his/her blocked pseudo-coding.

This leads 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.

Julia compiles its sources to the appropriate machine code using just-in-time compilation from LLVM and so achieves execution times comparable with those of C and Fortran.

It is natural that 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 similar to **JuliaStats** in statistics, a number of community groupings have been formed and these will be noted as we go along.

The Julia base (standard library) contains most of 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, machine learning plus many others. The general purpose packages may be native implementations of low-level data structures or wrappers around open source libraries. Worthy of note in the latter category is the interface to the GNU Scientific Library (`GSL.jl`).

In this chapter, I am going to concentrate on a few areas between the specialist and general purpose. It is not an exhaustive list, so I encourage you to review the `pkg.julialang.org` web pages as the number and breadth of Julia packages is growing rapidly. Please be aware that because of the nature of the subject matter of this chapter, one or two of the topics covered may prove to be little more challenging than other areas of the book to the non-mathematical minded readers.

Linear algebra

Linear algebra is the 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.

In Julia, basic linear algebra is built into the standard library. Much of this is achieved using the **Basic Linear Algebra System (OpenBLAS)** and **Linear Algebra PACKAGE (LAPACK)** libraries that ship as shared libraries in binary distros or are built when installing the system from source.

We will begin our study by looking at sets of linear equations and the application of matrix methods to their solutions.

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 term 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 with which many of the readers will be familiar. Consider the case of the following:

$$\begin{array}{rcl} x - 2y + 2z & = & 5 \\ x - y + 2z & = & 7 \\ -x + y + z & = & 5 \end{array}$$

In matrix notation, we write this as: $\mathbf{Av} = \mathbf{b}$, 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. So setting up A and b as:

```
julia> A = [1 -2  2; 1 -1  2; -1  1  1];
3x3 Array{Int64,2}:

julia> b = [5, 7, 5];
3-element Array{Int64,1};
```

We first note that the matrix A has a non-zero determinate, which implies that the system is not degenerate and will have a solution:

```
det(A) ; # => 3
```

In fact, 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 I . This is such a common operation that it is also defined as the matrix division operation $A \backslash b$:

```
julia> v = A\b ; # or inv(A)*b
3-element Array{Float64,1}:
 1.0
 2.0
 4.0
```

Note that both v and b are (column) vectors present in order to satisfy matrix multiplication rules.

However, the transpose of v will be a matrix consisting of a single row:

```
julia> transpose(v); # usually written as: v'
1x3 Array{Float64,2}:
 1.0  2.0  4.0
```

Consider an over-determined system, that is, three equations in two unknowns:

```
A1 = A[:, 2:3] ; # A1 ==> [-2 2; -1 2; 1 1];
```

Effectively, this is the same system of equations but just in y and z . Matrix division produces a solution, but by minimizing the least squares residuals of y and z :

```
julia> (A1\b)' # output the transpose for brevity
1x2 Array{Float64,2}:
 1.27586  3.93103
```

The alternative is an under-determined system comprising just the first two equations:

```
A2 = A[1:2,:]; b2 = b[1:2];
```

Now, there is an infinite set of solutions obtained by specifying one variable and solving for the other two. In this case, the minimum norm of the solutions is returned as:

```
julia> (A2\b2)'
1x2 Array{Float64,2}:
 1.8  2.0  3.6
```

Decompositions

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 as matrix decomposition and proves useful in a number of classes of problems.

The \ function hides how the problem is actually solved. Depending on the dimensions of the matrix A, 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.

Julia defines an {Factorization} abstract type and several composite subtypes for actually storing factorizations. This can be thought of as a representation of the A matrix.

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:

```
julia> Alu = lufact(A);  
# => LU{Float64,Array{Float64,2}}(3x3 Array{Float64,2})
```

The components of the factorization can be accessed using the symbols :P, :L, and :U:

```
julia> Alu[:U]  
3x3 Array{Float64,2}:  
 1.0  -2.0  2.0  
 0.0   1.0  0.0  
 0.0   0.0  3.0
```

Moreover, we can compute the solution from the components as:

```
julia> Alu[:U]\(Alu[:L]\Alu[:P]'*b)  
3-element Array{Float64,1}:  
 1.0  
 2.0  
 4.0
```

This computation is encapsulated in the \ operator from LU factorizations, so can be simply written as: Alu\b.

In the case where the system is over specified, a LU factorization is not appropriate as the matrix is tall, that is, has more rows than columns; here we can use QR factorization:

```
julia> A1 = A[:,2:3]
julia> Aqr = qrfact(A1)
QRCompactWY{Float64}(3x2 Array{Float64,2}:
 2.44949  -2.04124
 0.224745  -2.19848
-0.224745  0.579973,2x2 Triangular{Float64,Array{Float64,2}},:U,false}:
 1.8165  -0.256629
 0.0      1.49659 )
```

The \ operator has a method for the QR and the least squares problem can be solved as:

```
julia> Aqr\b
2-element Array{Float64,1}:
 1.27586
 3.93103
```

It is worth noting that while it is possible to call specific factorization methods explicitly (such as lufact), Julia has a factorize(A) function that will compute a convenient factorization, including LU, Cholesky, Bunch-Kaufman, and Triangular, based upon the type of the input matrix.

Also, there are the ! versions of functions such as lufact!() and factorize!() that will compute in place to conserve memory requirements.

Eigenvalues and eigenvectors

An eigenvector or characteristic vector of a A square matrix is a non-zero v vector that, when the matrix multiplies v, gives the same result as when some scalar multiplies v. The scalar multiplier is usually denoted by λ .

That is: $Av = \lambda v$ and λ is called the eigenvalue or characteristic value of A corresponding to v.

Considering our set of three equations, this will yield three [eigvecs] eigenvectors and the corresponding [eigvals] eigenvalues:

```
A = [1 -2  2; 1 -1  2; -1  1  1];
eigvals(A)
3-element Array{Complex{Float64},1}:
 -0.287372+1.35im
```

```
-0.287372-1.35im
1.57474+0.0im

V = eigvecs(A)
3x3 Array{Complex{Float64},2}:
 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.

To confirm, let's compute the difference of $(Av - \lambda v)$ for the first vector:

```
julia> A*V[:,1] - λ[1]*V[:,1]
3-element Array{Complex{Float64},1}:
 -2.22045e-16+2.22045e-16im
 1.11022e-16+0.0im
 2.77556e-16-1.38778e-16im
```

That is, all the real and imaginary parts are of the $e-16$ order, so this is in effect a zero matrix of complex numbers.

Why do we wish to compute eigenvectors?

- They make understanding linear transformations easy, as they are the directions along which a linear transformation acts simply by "stretching/compressing" and/or "flipping". Eigenvalues give the factors by which this compression occurs.
- They provide another way to affect matrix factorization using singular value decomposition using `svdfact()`.
- They are useful in the study of chained matrices, such as the cat and mouse example we saw earlier.
- They arise in a number of studies of a variety of dynamic systems.

We will finish this section by considering a dynamic system given by:

$$\begin{aligned}x' &= ax + by \\y' &= cx + dy\end{aligned}$$

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 population of two species that affect one another and are termed the Lotka-Volterra equations.

We may consider that species x is a predator of species y .

1. The more of x , the lesser of y will be around to reproduce.
2. But if there is less of y then there is less food for x , so lesser of x will reproduce.
3. Then if lesser of x are around, this takes pressure off y , which increases in numbers.
4. But then there is more food for x , so x increases.

It also arises when you have certain physical phenomena, such 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 differential equations. 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 that depend linearly on x and y .

That is, $p = \alpha x + \beta y$ and $w = \gamma x + \delta y$, for some constants α, β, γ , and δ .

The system is transformed into something as follows:

$$p' = \kappa p \text{ and } q' = \lambda q$$

That is, you can "decouple" the system, so that now you are 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 x and y to find expressions for x and y . This results precisely to finding two linearly independent eigenvectors for the $[a \ c; b \ d]$ matrix.

p and q correspond to the eigenvectors and to the eigenvalues.

So 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 differential equations, so that the system can be represented as 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.

Special matrices

The structure of matrices is very important in linear algebra. In Julia, these structures are made explicit through composite types such as `Diagonal`, `Triangular`, `Symmetric`, `Hermitian`, `Tridiagonal`, and `SymTridiagonal`.

Specialized methods are written for the special matrix types to take advantage of their structure.

So, `diag(A)` is the diagonal vector of the `A` but `Diagonal(diag(A))` is a special matrix:

```
julia> Diagonal(diag(A))
3x3 Diagonal{Int64}:
 1  0  0
 0 -1  0
 0  0  1
```

A symmetric eigenproblem

Whether or not Julia is able to detect if a matrix is symmetric/Hermitian, it can have a big influence on how fast an eigenvalue problem is solved. Sometimes it is known that a matrix is symmetric or Hermitian, but due to floating point errors this is not detected by the `eigvals` function.

In following example, `B1` and `B2` are almost identical, if Julia is not told that `B2` is symmetric, the elapsed time for the computation is very different:

```
n = 2000;
B = randn(n,n);
B1 = B + B';
B2 = copy(B1);
B2[1,2] += 1eps();
B2[2,1] += 2eps();
issym(B1)' # => true
issym(B2)' # => false
```

The `B1` matrix is symmetric whereas `B2` is not because of the small error added to cells $(1, 2)$ and $(2, 1)$.

Calculating the eigenvectors of `B1` and `B2` and timing the operation gives:

```
@time eigvals(B1);
elapsed time: 1.543577865 seconds (37825492 bytes allocated)

@time eigvals(B2);
elapsed time: 8.015794532 seconds (33142128 bytes allocated)
```

However, if we symmetrize B_2 and rerun the calculation:

```
@time eigvals(Symmetric(B2));  
elapsed time: 1.503028261 seconds (33247616 bytes allocated)
```

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, ECG, 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, which takes a $x(t)$ function of time that is called the time-domain signal and computes the weights of its sinusoidal components. These weights are represented as a $x(f)$ function of frequency called the frequency-domain signal.

The Fourier transform 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 Fourier transform (DFT)**.

This takes a set of N samples in time and produces weights at N different frequencies. Julia's signal processing library, like most common signal processing software packages, computes DFTs by using a class of algorithms known as **fast Fourier transforms (FFT)**.

Filtering and smoothing

We will construct a signal of three sinusoids with frequencies of 500 Hz and multiples such as 1000 Hz, 1500 Hz, and so on, using the following code:

```
using Winston
fq = 500.0;
N = 512;
T = 6 / fq;
td = linspace(0, T, N);
x1 = sin(2pi * fq * td);
x2 = cos(8pi * fq * td);
x3 = cos(16pi * fq * td);
x = x1 + 0.4*x2 + 0.2*x3
plot(td, x)
```

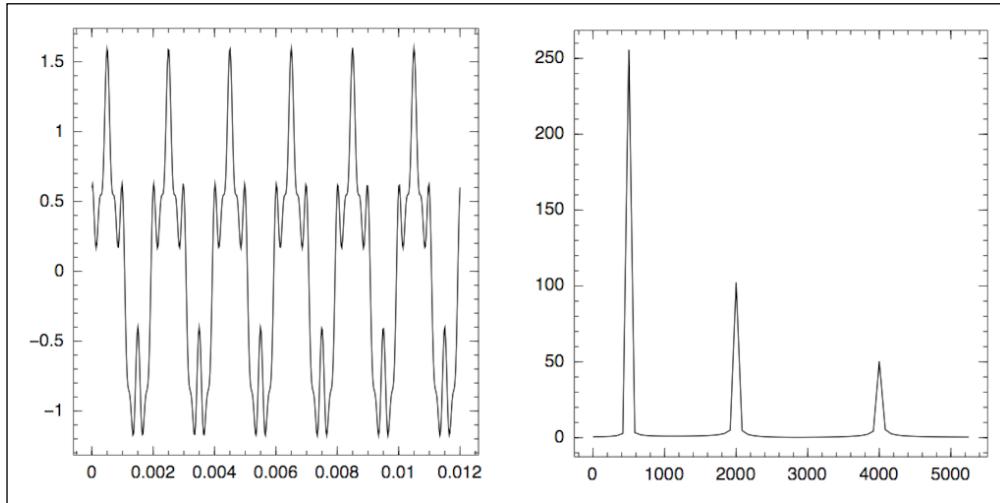
Now use the `rfft` function (the real FFT function), since our input signal is composed entirely of real numbers -- as opposed to complex numbers. This allows us to optimize by only computing the weights for frequencies from 1 to $N/2+1$.

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.

```
x = rfft(x);
sr = N / T;
fd = linspace(0, sr / 2, int(N / 2) + 1);
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:



Now we can add some high frequency noise to the signal using:

```
ns = 0.1*randn(length(x));
xn = x + ns;
```

Then use a convolution procedure in the time domain to attempt to remove it. In essence, this is a moving average 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.

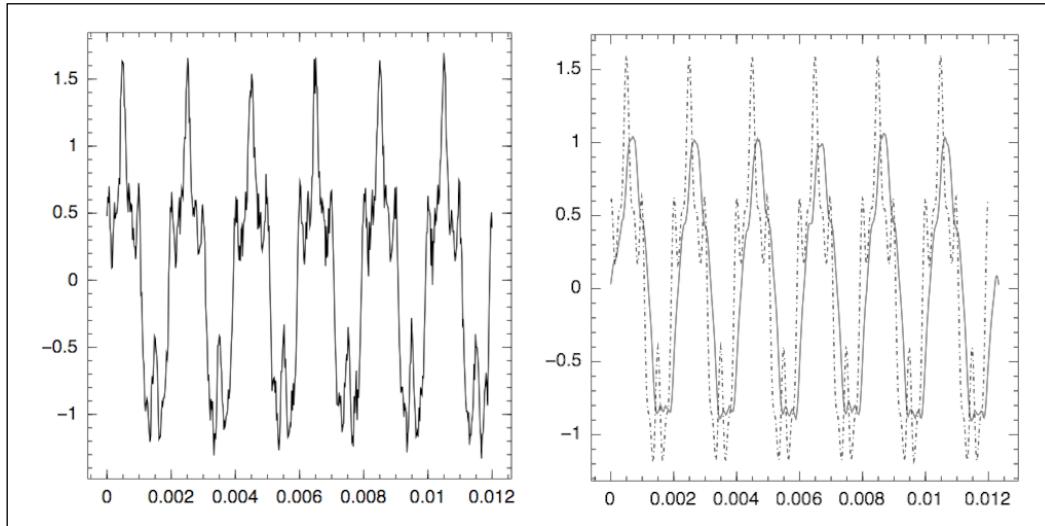
```
M = 16;
xm = ones(Float64, M) / M;
plot(xm)
```

It is important that the sum of the weights is 1.

The Julia standard library has a built-in convolution function and applying xm to xn:

```
xf = conv(xn, xm);
t = [0:length(xf) - 1] / sr
plot(t, xf, "-", t, x, ";" )
```

The following figure shows the noisy signal together with the filtered one:



The main carrier wave is recovered and the noise eliminated, but given the size of the window chosen the convolution has a drastic effect on the higher frequency components.

Digital signal filters

Moving average 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 the 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` group and we will apply some of the routines to filter out the noise on the signal we created in the previous section:

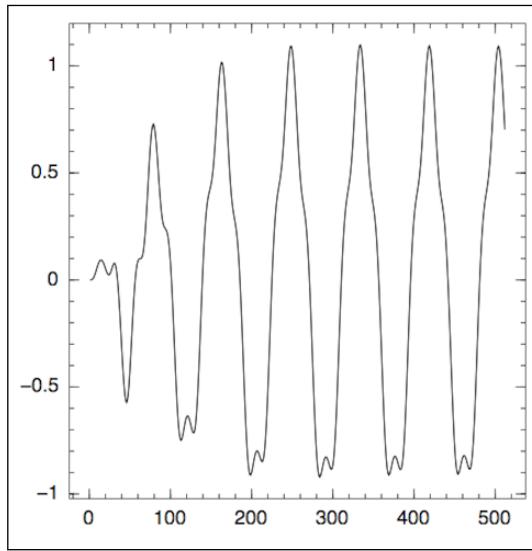
```
using DSP
responsetype = Lowpass(0.2)
```

```
prototype = Elliptic(4, 0.5, 30)
tf = convert(TFFilter, digitalfilter(responsetype, prototype))
numerator_coefs = coefb(tf)
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:

```
responsetype = Bandpass(10, 40; fs=1000)
prototype = Butterworth(4)
xb = filt(digitalfilter(responsetype, prototype), x);
plot(xb)
```

This code filters the data in the x signal, sampled at 1000 Hz, with a fourth order Butterworth bandpass filter between 10 and 40 Hz. The resultant signal is displayed as follows:



While being cleaner than convolution, this still affects the high frequencies. Also, while the band pass filter is infinite in extent, the one constructed is truncated and this means that the initial portion of the signal is modulated.

Image processing

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 section on *Frequency analysis* 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 histogram range of the image is altered. We will look at the lena image that is provided as a 512x512 PGM image:

```
img = open("lena.pgm");
magic = chomp(readline(img));
params = chomp(readline(img));
while ismatch(r"\s*#", params)
    params = chomp(readline(img));
end
pm = split(params);
if length(pm) < 3
    params *= " " * chomp(readline(img));
end
wd = int(pm[1]);
ht = int(pm[2]);

data = uint8(readbytes(img,wd*ht));
data = reshape(data,wd,ht);
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 as:

```
Gx = [1 2 1; 0 0 0; -1 -2 -1];
Gy = [1 0 -1; 2 0 -2; 1 0 -1];
```

The following loops over blocks of the original image applying Gx and Gy, 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:

```
dout = copy(data);
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 = int(sqrt(x*x + y*y))
    dout[i,j] = (p < 256) ? uint8(p) : 0xff
end
out = open("lenaX.pgm", "w");
println(out,magic);
println(out,params);
write(out,dout);
close(out);
```

The result is shown in the following figure:



Differential equations

Differential equations 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 differential equations. If more than a single dependent variable is involved, then they are termed partial differential equations.

Julia has a number of packages that aid the calculation of differentials of functions and to solve systems of differential equations. We will look at a few aspects of these in the next section.

The solution of ordinary differential equations

Julia supports the solution of ordinary differential equations through a couple of packages such as ODE and Sundials. ODE consists of routines written solely in Julia whereas Sundials is a wrapper package around a shared library.

We will look at the ODE package first.

ODE exports a set of adaptive solvers; adaptive meaning that the step size of the algorithm changes algorithmically to reduce the error estimate below a certain threshold.

The calls take the `odeXY` form, where `x` is the order of the solver and `y` is the error control:

- `ode23`: 2nd order adaptive solver with 3rd order error control
- `ode45`: 4th order adaptive solver with 5th order error control
- `ode78`: 7th order adaptive solver with 8th order error control

To solve the explicit ODE defined as a vectorize set of the $dy/dt = F(t, y)$ equations, all routines of which have the same basic form, we use: `(tout, yout) = odeXY(F, y0, tspan)`.

As an example, I will look at this as a linear three-species food chain model where the lowest-level `x` prey is preyed upon by a mid-level `y` species, which in turn is preyed upon by a top level `z` predator. This is an extension of the Lotka-Volterra system from two to three-species.

Some examples might be three-species ecosystems, such as mouse-snake-owl, vegetation-rabbits-foxes, and worm-sparrow-falcon.

```
x' = a*x - b*x*y  
y' = -c*y + d*x*y - e*y*z  
z' = -f*z + g*y*z #for a,b,c,d,e,f,g > 0
```

In the preceding equations, `a`, `b`, `c`, `d` are in the 2-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.

This translates to the following set of linear equations:

```
x[1] = p[1]*x[1] - p[2]*x[1]*x[2]  
x[2] = -p[3]*x[2] + p[4]*x[1]*x[2] - p[5]*x[2]*x[3]  
x[3] = -p[6]*x[3] + p[7]*x[2]*x[3]
```

It is slightly over-specified, since one of the parameters can be removed by rescaling the timescale. We define the `F` function as follows:

```
function F(t,x,p)
d1 = p[1]*x[1] - p[2]*x[1]*x[2]
d2 = -p[3]*x[2] + p[4]*x[1]*x[2] - p[5]*x[2]*x[3]
d3 = -p[6]*x[3] + p[7]*x[2]*x[3]
[d1, d2, d3]
end
```

This takes the time range, vectors of the independent variables and coefficients, and returns a vector of the derivative estimates:

```
p = ones(7);      # Choose all parameters as 1.0
x0 = [0.5, 1.0, 2.0]; # Setup the initial conditions
tspan = [0.0:0.1:10.0]; # and the time range
```

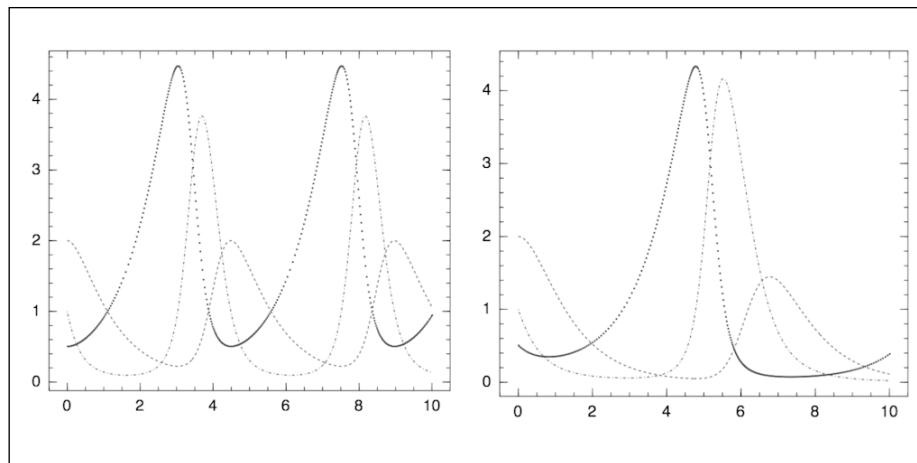
Solve the equations by calling the `ode23` routine. This returns a matrix of solutions in a columnar order that we can extract and display with the `using Winston` function:

```
(t,x) = ODE.ode23((t,x) -> F(t,x,pp), x0, tspan);

n = length(t);
y1 = zeros(n); [y1[i] = x[i][1] for i = 1:n];
y2 = zeros(n); [y2[i] = x[i][2] for i = 1:n];
y3 = zeros(n); [y3[i] = x[i][3] for i = 1:n];

using Winston
plot(t,y1,"b.",t,y2,"g-.",t,y3,"r--")
```

The output is shown as the LH graph of the figure as follows:



This model assumes the z species does not directly *eat* x . That might not be true, for example, for the mouse-snake-owl ecosystem, so in this case we would add an additional term:

$$x' = a*x - b*x*y - h*xz$$

Redoing the model as:

```
d1 = p[1]*x[1] - p[2]*x[1]*x[2] - p[8]*x[1]*x[3]
pp = ones(7);
pp[5] = pp[8] = 0.5; # Split rate so that p5 + p8 = 1
```

Apparently, the peak populations of the three species are little altered by the extra term but the periodicity is almost doubled. This is shown in the preceding RH graph.

Non-linear ordinary differential equations

Non-linear ODEs differ from their linear counterparts in a number of 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.

A classic example is the double pendulum that comprises of one pendulum with another pendulum attached to its end. It is a simple physical system that exhibits rich dynamic behavior with strong sensitivity to initial conditions that under some instances can result in producing chaotic behaviors.

The example we are going to look at is somewhat simpler, a non-linear system arising from chemistry.

We will consider the temporal development of a chemical reaction. 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.

Assuming the reaction is gaseous, then we can ignore heat transfer across the vessel. So the change in temperature will be given by:

$$\frac{dT}{dt} = Ae^{-E/RT} + B(T - T_0)$$

It is possible to write $e^{-E/RT} \sim e^{-E/RT_0} (T - T_0)$, which means at the low temperature, behavior is proportional to the exponential value of the temperature difference.

In addition, we will assume that the source of the heat (that is, the reactant 1) is limited. So the first term will now be a second differential equation in \ln , where n is the order of the reaction usually 1 or 2.

Although the ODE package is capable of handling non-linear systems, I will look at a solution that utilizes the alternative Sundials package.

Sundials is a wrapper around a C library and the package 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 solution of nonlinear algebraic systems. KINSOL treats nonlinear systems of the form $F(u) = 0$.

Tom Shorts' `Sim.jl` is a Julia package to support equation-based modeling for simulations, but this is outside the scope of this book. See the repository on GitHub for more details (<https://github.com/tshort/Sims.jl>).

By redefining the time scales, it is possible to simplify the equations and write the equations in terms of temperature difference x_1 and reactant concentration x_2 as:

```
x1 = x2^n * exp(x1) - a*x1
x2 = -b*x2^n * exp(x1)
```

Here, a and b are parameters and initial concentrations of $x_1 = 0$ and $x_2 = 1$.

This can be solved as:

```
using Sundials
function exoterm(t, x, dx; n=1, a=1, b=1)
    p = x[2]^n * exp(x[1])
    dx[1] = p - a*x[1]
    dx[2] = -b*p
    return(dx)
end;

t = linspace(0.0,5.0,1001);
fexo(t,x,dx) = exotherm(t, x, dx, a=0.6, b=0.1);
x1 = Sundials.cvode(fexo, [0.0, 1.0], t);
```

Executing this for a in $[0.9, 1.2, 1.5, 1.8]$ and keeping $b = 0.1$ gives four solutions, which are shown in the LH panel of the following figure:

```
using Winston
plot(t,x1[:,1],t,x2[:,1],t,x3[:,1],t,x4[:,1])
```

The solutions are stable for $a = 1.8$, but starts to increase for lower values of a and are only pulled back by depletion of the fuel.

If we start solving this for a simple case without any fuel depletion: $x_1 = \exp(x_1) - a*x_1$.

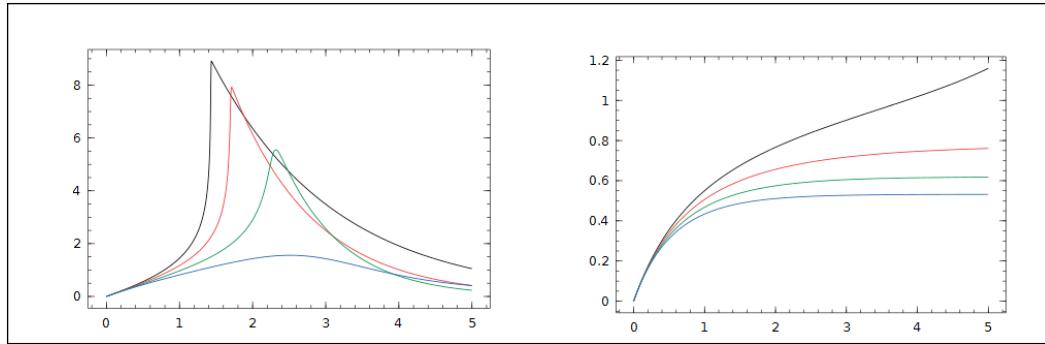
This is only stable for values of a greater than $\exp(1)$:

```
using Roots
f(x) = exp(x) - a*x
for a = 2.8:-0.02:2.72
    @printf "%f : %f\n" a fzero(f,1)
end
2.80 : 0.775942
2.78 : 0.802795
2.76 : 0.835471
2.74 : 0.879091
2.72 : 0.964871
```

So simplifying the `exotherm` function to that of a single variable with the heat loss a parameter and re-running for a in $[2.6, 2.8, 3.0, 3.2]$ produces the set of solutions shown in the RH panel in the following figure:

```
function exotherm2(t, x, dx; a=1)
    dx[1] = exp(x[1]) - a*x[1]
    return dx
end;

fexo2(t,x,dx) = exotherm2(t, x, dx, a=2.6);
x1 = Sundials.cvode(fexo2, [0.0, 1.0], t);
....
```



Partial differential equations

At the time of writing, solving differential equations in more than a single variable was not well supported in Julia as in the ODE case. Some work has been done on finite element analysis, but there is little in the way of any packages providing general purpose PDE solvers.

Consider the temporal development of temperature in the standard diffusion equation with a non-linear (heating) source.

The full equation in three spatial dimensions can be written as:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} + \Theta(u)$$

The internal heating term can once again be considered as exponential for low temperature differences, but we are now considering the case where there is conduction across the medium.

This can be solved by a recurrence relation and for simplicity we will consider time plus just a single spatial dimension, but the extension to three dimensions is straightforward.

To get the reaction started, we will assume a hotspot at the centre and segment the spatial dimension into $2n+1$ intervals of δx .

The $\Theta(u)$ heating term may be parameterized by two constants p_1 and p_2 and the equation written as:

$$u[i] = u[i-1] - (1-2r)*u[i] + u[i+1] + p1*exp(p2*u[i])$$

With $u[1] = u[2n+1] = 0$ and initial condition $u[n] = 10$, where $r = \delta t / \delta x^* \delta x$ and boundary conditions $u[0] - u[1] = u[2n] - u[2n+1] = p_3$.

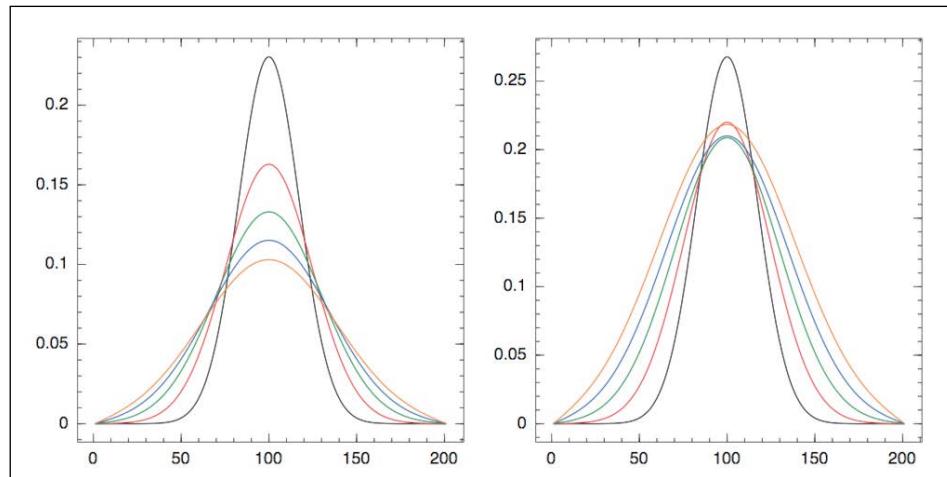
In Julia, this can be written as:

```
p = [0.001, 0.01, 0.1];
r = 0.01;
n = 100;
t = linspace(1,2n+1,2n+1);
u = zeros(2n+1);
u[n] = 10;           # Hot spot in centre
w = zeros(5,2n + 1);

for k1 = 1:5
    for j = 1:15000
        v = copy(u);
        v[1]      = p[3]*(v[1] - v[2]);
        v[2n+1]   = p[3]*(v[2n] - v[2n+1]);
        for i = 2:2n
            u[i] = v[i] + r*(v[i-1] - 2*v[i] + v[i+1]) +
                p[1]*(exp(p[2]*v[i]) - 1);
        end
    end
    [w[k1,k2] = u[k2] for k2 = 1:2n+1];
    (k1 == 1) ? plot(w[k1,:]) : plot(w[k1,:])
end
```

If $p[1] = p[2] = p[3]$, then there is no internal heating term and the solution is one for the regular diffusion equation.

This is shown in the LH panel of the following figure, whereas the RH panel is the parameterized solution as shown previously:



It is clear that temperature within the body has begun to rise since the heat loss at the boundaries is insufficient to dissipate the internal heating and in the absence of fuel depletion, the body will ignite.

Optimization problems

Mathematical optimization problems arise in the field of linear programming, machine learning, resource allocation, production planning, and so on.

One well-known allocation problem is that of the travelling 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 as 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:

- No solution exists.
- 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 .
- The function $f(x)$ having multiple extreminals.
- 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, the latter is quicker utilizing hill climbing type algorithms.

Optimization is supported in Julia by a community group JuliaOpt and we will briefly introduce three packages: JuMP, Optim, and NLOpt.

JuMP

Mathematical programming problems are supported in Python and MATLAB with packages such as PuLP, CVX, YALMIP; there are also commercial offerings such as AMPL, and GAMS. In general, the objective of the algorithms is not immediately apparent from the code.

The aim of the JuMP package is to create coding that provides a natural translation of the mathematics of the problem, preferably by loosely coupling the specific method of solution from the data. This is achieved by extensive generation of boilerplate code using macros and the application of specialized solvers for linear, conic, and semidefinite problems.

JuMP utilizes a MathProgBase secondary package whose function is to interface to solvers such as Gurobi, Clp, Mosek, and Ipopt, all of which are defined in Julia in packages such as `Gurobi.jl`, and so on.

The backend solvers are often commercial offerings, although in some cases academic licenses exist.

A simple example is to maximize the $5x + 3y$ function subject to the constraint that:

```
3x + 5y < 7
using JuMP
m = Model()
@defVar(m, 0 <= x <= 5 );
@defVar(m, 0 <= y <= 10 );
@setObjective(m, Max, 5x + 3y );
@addConstraint(m, 2x + 5y <= 7.0 );
```

The use of the `@defVar`, `@setObjective`, and `@addConstraint` define the starting domain form (x, y), function (objective), and constraints in a clear manner:

```
status = solve(m); # => :Optimal
printf "Function value of %5.2f at (%4.2f,%4.2f)\n"
getObjectiveValue(m) getValue(x) getValue(y)
Functional value of 15.60 at (3.00, 0.20)
```

Calling `solve()` produces the result (when possible). The solver used is default for the case of problem (in this case Clp) and this has to be available.

It is possible to call `Model()` with specific solvers such as `Model(solver=GurobiSolver())` that will execute as long as the solver specified is appropriate to the problem type.

The solution to the travelling salesman problem is given in the JuMP (GitHub) source. However, this requires the Gurobi solver, so as an alternative we will look at a solution to the **knapsack** problem.

This is a problem in combinatorial optimization; given a set of items, each with a mass and a value, 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 a fixed-size knapsack, 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, and nowadays 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.

```
using JuMP
N = 6;
m = Model(); # Use default solver
@defVar(m, x[1:N], Bin); # Define array variable to hold results
profit = [ 5, 3, 2, 7, 4, 4 ]; # Profit vector of size N
weight = [ 2, 8, 4, 2, 5, 6 ]; # Weights vector of size

capacity = 12;
@setObjective(m, Max, dot(profit, x));
@addConstraint(m, dot(weight, x) <= capacity);
```

The objective is to maximize profit 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 `maxcap`:

```
status = solve(m); # Solve problem using MIP solver
@printf "Value of the objective is %.1f" getObjectiveValue(m);

println("Solution is:")
for i = 1:N
    print("x[$i] = ", getValue(x[i]));
    println(", p[$i]/w[$i] = ", profit[i]/weight[i]);
end

Value of the objective is 16.0
x[1] = 1.0, p[1]/w[1] = 2.5
x[2] = 0.0, p[2]/w[2] = 0.375
x[3] = 0.0, p[3]/w[3] = 0.5
x[4] = 1.0, p[4]/w[4] = 3.5
x[5] = 1.0, p[5]/w[5] = 0.8
x[6] = 0.0, p[6]/w[6] = 0.667
```

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 that requires at least a function definition and vectors the starting values. Optionally, a value for the solution method can be supplied with one of the following:

- `:bfsgs`
- `:cg`
- `:gradient_descent`
- `:momentum_gradient_descent`
- `:l_bfgs`
- `:nelder_mead`
- `:newton`
- `:simulated_annealing`

It is also possible to aid optimization by additionally providing functions based on the first (gradient) and second (hessian) partial derivatives.

The value returned is a data structure with 15 elements, whose symbols may be displayed using the `names()` function

The values for the optimization process are set by default, but these can be overwritten:

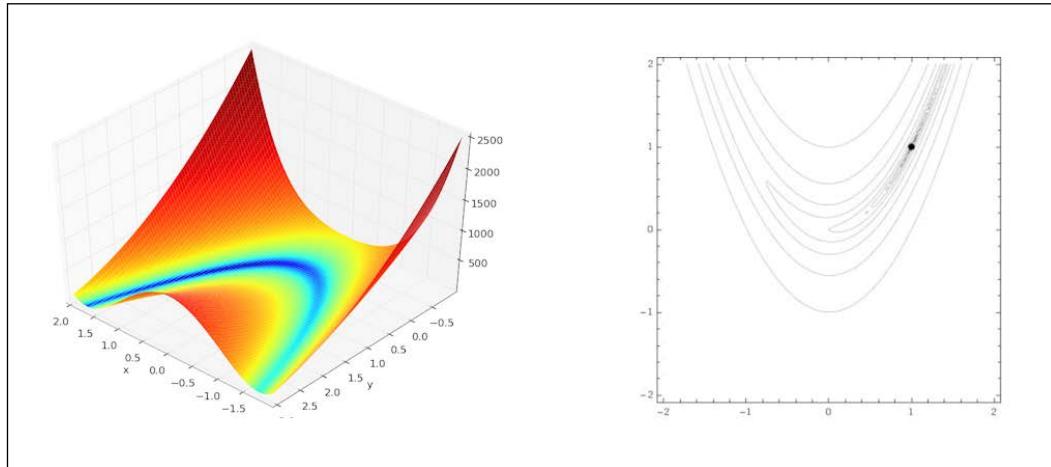
- `xtol`: Threshold tolerance in x ($1e-32$)
- `ftol`: Threshold tolerance in f ($1e-32$)
- `grtol`: Gradient tolerance ($1e-8$)
- `iterations`: Maximum number of iterations (1000)
- `store_trace`: Stores algorithm's state (false)
- `show_trace`: Outputs algorithm's state (false)

The Rosenbrock function is a non-convex function 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, to converge to the global minimum is difficult.

The function is defined by: $f(x, y) = (a-x)^2 + b(y-x^2)^2$.

It has a global minimum at $(x, y) = (a, a^2)$, where $f(x, y) = 0$.

Usually the (a, b) parameters are chosen as $(1, 100)$.



We can define the Rosenbrock function and solve the problem using Optim starting at $(0.0, 0.0)$ as:

```
function rb(x::Vector)
    return (1.0 - x[1])^2 + 100.0 * (x[2] - x[1]^2)^2;
end

using Optim
opt1 = optimize(rb, [0.0, 0.0]);

Results of Optimization Algorithm
* Algorithm: Nelder-Mead
* Starting Point: [0.0, 0.0]
* Minimum: [1.000005438687492, 1.0000079372595394]
* Value of Function at Minimum: 0.000000
* Iterations: 60
* Convergence: true
  * |x - x'| < NaN: false
  * |f(x) - f(x')| / |f(x)| < 1.0e-08: true
  * |g(x)| < NaN: false
  * Exceeded Maximum Number of Iterations: false
* Objective Function Calls: 115
* Gradient Call: 0
```

The minimum value took 115 iterations. This can be changed markedly by supplying gradient and Hessian functions:

```
function rb_grad(x::Vector, sv::Vector)
    sv[1] = -2.0*(1.0 - x[1]) - 400.0*(x[2] - x[1]^2) * x[1];
    sv[2] = 200.0*(x[2] - x[1]^2);
end

function rb_hess(x::Vector, sm::Matrix)
    sm[1, 1] = 2.0 - 400.0*x[2] + 1200.0*x[1]^2;
    sm[1, 2] = -400.0*x[1];
    sm[2, 1] = -400.0*x[1];
    sm[2, 2] = 200.0;
end
```

In the preceding code:

$$\text{gradient} = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \end{bmatrix} \text{hessian} = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix}$$

Re-running the optimization produces:

```
optimize(rb, rg_grad, rb_hess, [0.0,0.0], method=:newton, show_trace=true);

      Inter      Function value      Gradient norm
        0      1.000000e+00      2.000000e+00
        1      8.431140e-01      1.588830e+00
        2      6.586412e-01      4.959487e+00
        .      .....
        .      .....
       12      3.784204e-13      2.087334e-05
       13      5.639268e-24      5.214740e-11

Results of Optimization Algorithm
* Algorithm: Newton's Method
* Starting Point: [0.0,0.0]
* Minimum: [0.9999999999979515,0.9999999999960232]
* Value of Function at Minimum: 0.000000
* Iterations: 13
* Convergence: true
* |x - x'| < 1.0e-32: false
```

```
* |f(x) - f(x')| / |f(x)| < 1.0e-08: false
* |g(x)| < 1.0e-08: true
* Exceeded Maximum Number of Iterations: false
* Objective Function Calls: 54
* Gradient Call: 54
```

This converges to the same ~ (1.0, 1.0) solution, but now in just 13 iterations.

NLopt

NLopt is a free/open source library for nonlinear optimization problems with or without gradient information, and can be built from source from <http://ab-initio.mit.edu/nlopt>, although binaries exist for Windows. The manual on NLopt is also available from the MIT site.

This provides a common interface for many different optimization algorithms, including:

- Both global and local optimization
- Algorithms using function values only (derivative-free) and also algorithms exploiting user-supplied gradients
- Algorithms for unconstrained optimization, bound-constrained optimization, and general nonlinear inequality/equality constraints

The interface in Julia (`NLopt.jl`) can utilize the NLopt library directly with calls via its API, or alternatively by using the generic `MathProgBase` interface.

The NLopt API defines a structure of the `nlopt_opt` type, and this is encapsulated in the Julia package by the `Opt` type, which is normally created by the `Opt(algor, ndims)` constructor for a given type of the `algor` algorithm, and number of the `ndims` optimization parameters.

Algorithms are specified by symbols such as `:LD_MMA`, `:NL_COBYLA`, `:PRAXIS`, and so on, the full list is given at <https://github.com/JuliaOpt/NLopt.jl>. In this case:

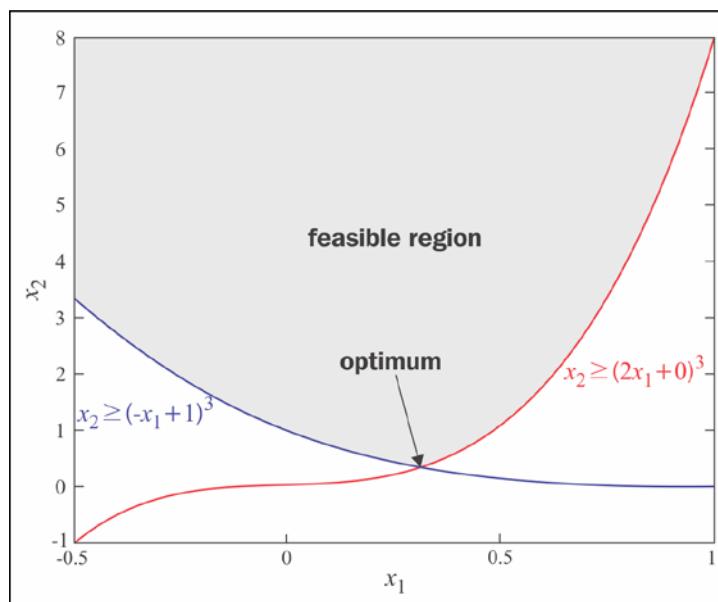
- The objective function is either `min_objective!(opt, n)` or `max_objective!(opt)`.
- The form of the function is `f(x::Vector, grad)::Vector`. A zero length vector can be passed for `grad`, otherwise it should be equivalent to the partial derivative of `f` with respect to `xi`.
- Constraints are specified by `lower_bounds!(opt, lb)` or `upper_bounds!(opt, rb)`, where `lb` and `rb`. An unbounded dimension may be specified as `+/- Inf`.

- It is also possible to specify non-linear constraints using `inequality_constraint()` or `equality_constraint()`.

Consider a non-linearly constrained minimization problem:

Minimise `sqrt(x2)`, subject to $x_2 > 0$, $x_2 > (-x_1 + 1)^3$, and $x_2 > (2x_1 + b_2)^3$ for parameters $a_1=2$, $b_1=0$, $a_2=-1$, $b_2=1$.

The feasible region defined by these constraints is shown as follows:



x_2 is constrained to lie above the maximum of the two cubic functions and the optimum point is located at their intersection $(1/3, 8/27)$.

The value of the objective at this point is $\sqrt{x_2} \Rightarrow \sqrt{8/27} \sim 0.5443$.

The solution in Julia is:

```
using NLopt
count = 0;
function myfunc(x::Vector, grad::Vector)
    if length(grad) > 0
        grad[1] = 0
        grad[2] = 0.5/sqrt(x[2])
    end
    global count;
```

```
count::Int += 1;
sqrt(x[2]);
end
function mycons(x::Vector, grad::Vector, a, b)
    if length(grad) > 0
        grad[1] = 3*a * (a*x[1] + b)^2
        grad[2] = -1
    end
    (a*x[1] + b)^3 - x[2]
end
```

The routine checks the `grad` vector for non-zero length, in which case the derivatives with respect to `x1` and `x2` are returned:

```
opt = Opt(:LD_MMA, 2);
lower_bounds!(opt, [-Inf, 0.]);
xtol_rel!(opt, 1e-4);
min_objective!(opt, myfunc);
```

`xtol_rel()` specifies a tolerance condition for the optimization and the constraints are defined as follows:

```
inequality_constraint!(opt, (x,g) -> mycons(x,g,2,0), 1e-8);
inequality_constraint!(opt, (x,g) -> mycons(x,g,-1,1), 1e-8);
```

Finally, the `optimize()` function is called with a starting estimate and returns the function value, a vector of the optimal point, and a return code.

Possible successful values of the return code are:

```
:SUCCESS, :XTOL_REACHED, :STOP_VALUE_REACHED, :FTOL_REACHED
```

This does not necessarily indicate that the algorithm has produced definitive results.

Failure-style return values indicate that the algorithm did not run, and include:

```
:FAILURE, :INVALID_ARGS, :OUT_OF_MEMORY
```

So running the minimization gives:

```
(minf,minx,ret) = optimize(opt, [1.2, 5.6]);
@printf "Ans => %6.3f at (%6.3f,%6.3f)" minf minx[1] minx[2];
Ans # => 0.5433 at (0.333, 0.296)
```

Using with the MathProgBase interface

As noted, NLOpt implements the MathProgBase interface for non-linear optimization, so it can be used interchangeably with JuMP.

The NLOpt solver is named `NLOptSolver` and the solution for the preceding problem is:

```
using JuMP
using NLOpt
m = Model(solver=NLOptSolver(algorithm=:LD_MMA));

a1 = 2; b1 = 0;
a2 = -1; b2 = 1;

@defVar(m, x1);
@defVar(m, x2 >= 0);
@setNLObjective(m, Min, sqrt(x2));
@addNLConstraint(m, x2 >= (a1*x1+b1)^3);
@addNLConstraint(m, x2 >= (a2*x1+b2)^3);

setValue(x1, 1.2);
setValue(x2, 5.6);
status = solve(m);
@printf "Ans => %6.4f at (%5.3f,%5.3f)\n"
getObjectiveValue(m) getValue(x1) getValue(x2);

Ans # => 0.5433 at (0.333,0.296)
```

This indicates the power of the macro definitions in decoupling the problem definition from the actual details of the coding.

The call requires an algorithm parameter to be specified. There are a number of other parameters that are can be default, unless provided explicitly.

Stochastic problems

Problems encountered so far are completely determined by the models, and will produce the same solutions repeatedly. Some models have terms that occur randomly, and these are called stochastics.

We have already seen one example earlier, that of the price a volatile stock. While the price increases roughly which 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 the underlying trends, but there are certain systems where the stochastics are paramount. Typical examples are models of queueing systems that might occur in service at banks, or checkouts in supermarkets. I will discuss a particular case of the bank teller later in this section.

Stochastic simulations

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 for simulation problems by IBM with GPSS in 1961; modern day inheritors are `jDisco` and `SimPy`.

SimJulia

The `SimJulia` package is similar to the Python-based `SimPy` module, although as it is a native implementation, neither a wrapper nor requires the use of `PyCall`.

`SimJulia` is both a discrete event and a continuous time process simulation framework. A process is implemented as a type containing a `Task` (co-routine).and the `produce()` function is used to schedule future events providing a pointer to the `Task` object.

In discrete event models, the 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.

`SimJulia` implements the following process intrinsics:

- `sleep`: The current process is deactivated and can be reactivated by another process
- `hold`: The process is busy for a certain amount of time, during this period an interruption can force the control back to the process
- `wait` and `queue`: A process can wait or queue for some signals to be fired by another process
- `waituntil`: A process can wait for a general system to be satisfied

- `request and release`: A resource, a discrete congestion point to which processes may have to queue up, can be requested and when it is finished, be released
- `put and get`: A level (continuous) and a store (discrete) model congestion point that can produce or consume continuous/discrete "material"
- `observe`: A monitor enables the observation of a single variable of interest and can return a data summary during or at the end of a simulation run

Many of the examples provided in the Julia package are direct analogues of these in the Python module. 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 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 (bank 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.

Distributions will clearly not be Gaussian, since we cannot have negative waiting and service times. Arrivals may differ in a city situation where there is a peak around lunchtimes and a 'rush' towards 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 balance against the need to minimise 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 the 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:

```
using SimJulia
using Distributions

function visit(customer::Process,
    time_in_bank::Float64,
    counter::Resource,
    max_in_queue::Int)
    arrive = now(customer);
    @printf("%8.4f %s: Here I am\n", arrive, customer);
    if length(counter.wait_queue) < max_in_queue
        request(customer, counter);
        wait = now(customer) - arrive;
        push!(wait_times, wait);
        @printf("%8.4f %s: Waited %6.3f\n",
            now(customer), customer, wait);
        hold(customer, time_in_bank)

        release(customer, counter);
        @printf("%8.4f %s: Finished\n", now(customer), customer)
    else
        @printf("%8.4f %s: Waiting\n", now(customer), customer)
    end
end
```

Next, we need to be able to create a number of customer processes and define the following function to do this:

```
function generate(source::Process, number::Int,
    mean_time_between_arrivals::Float64,
    mean_time_in_bank::Float64,
    counter::Resource,
    max_in_queue::Int)
    d_tba = Exponential(mean_time_of_arrivals);
    d_tib = Exponential(mean_time_in_bank);
    for i = 1:number
```

```
c = Process(simulation(source),
@sprintf("Customer%02d", i));
tib = rand(d_tib);
activate(c, now(source), visit, tib,
counter, max_in_queue);
tba = rand(d_tba);
hold(source, tba);
end
end

function number_in_system(counter::Resource)
length(counter.active_set) + length(counter.wait_queue);
end
```

As mentioned earlier, both arrivals and service (time in bank) times are Poisson processes, and so are solely parameterized by the distribution means. Processes are activated or held depending on whether resource is available and the number in the bank at any onetime is simply the number of customers being served and the number waiting.

We need to set some limits for the time the bank is open, the maximal queue length, and also values for the means that parameterize the exponential distributions.

Finally, we need to set the random number seed, activate the initial processes, and run the simulation:

```
# Set simulation data
max_number = 20;
max_time = 300.0;
max_queue = 1;
mean_arrival_time = 5.0;
mean_service_time = 8.0;
number_of_tellers = 2;
seed = randomize(); # Use function defined early or pick # a value to
make run reproducible

global wait_times = Array(Float64,1);

# Model/Experiment
strand(seed);
sim = Simulation(uint(16));
k = Resource(sim, "Counter", uint(1), false);
s = Process(sim, "Source");
activate(s, 0.0, generate,
        max_number, mean_arrival_time,
        mean_service_time, k, max_queue);
```

The bank is open for 5 hours, or until 20 customers have been served. Customers arrive on an average every 5 minutes and it takes around 8 minutes to be served:

```
run(sim, max_time);
@printf "Average waiting time is %5.3f\n" mean(wait_times);

0.0000 Customer01: Arrives
0.0000 Customer01: Waited 0.000
3.1863 Customer01: Finished
9.9273 Customer02: Arrives
9.9273 Customer02: Waited 0.000
10.9117 Customer03: Arrives
19.0337 Customer02: Finished
19.0337 Customer03: Waited 8.122
21.2591 Customer04: Arrives
26.3771 Customer05: Arrives
26.6146 Customer03: Finished
26.6146 Customer04: Waited 5.355
29.2442 Customer04: Finished
29.2442 Customer05: Waited 2.867
32.1817 Customer05: Finished
35.8994 Customer06: Arrives
35.8994 Customer06: Waited 0.000
.....
.....
Average waiting time is 8.762
```

Bayesian methods and Markov processes

To end this sortie to some of the aspects of scientific methods in Julia, we will look at another framework that is used in investigation problems arising in Bayesian analysis.

Some of these problems are difficult to vectorize, so Julia has a distinct advantage over alternative data science languages since de-vectorize programs run as quick or even more quickly than the vectorized ones.

Bayesian analysis is a form of statistic inference and at the heart of the analysis is the Bayes' Rule, which tells us how to do inference about hypotheses from data.

A model describes data that we can observe from a system. If we use probability theory to express all forms of uncertainty and noise associated with our model, then inverse probability (that is, Bayes rule) allows us to infer unknown quantities, adapt our models, make predictions, and learn from data.

Bayesian inference is about the quantification and propagation of uncertainty defined via a probability, in light of observations of the system. This is fundamentally different from classical inference that tends to be concerned with parameter estimation.

Monte Carlo Markov Chains

The aspect of Bayesian methods with which we will be concerned is **Monte Carlo Markov Chain (MCMC)**.

The MCMC procedure enables you to carry out analysis on a wide range of complex Bayesian statistical models. It uses algorithms to draw samples from an arbitrary posterior distribution, which is defined by the prior distributions for the parameters and the likelihood function for the specified data.

The algorithms are also called samplers and many exist, notable among these are the Gibbs sampler and the Metropolis-Hastings algorithm. Gibbs sampling has given rise to frameworks specifically utilizing the methods such as **Bayesian inference Using Gibbs Sampling (BUGS)** and **Just Another Gibbs Sampler (JAGS)**.

MCMC algorithms generate a Markov chain of samples, each of which is correlated with nearby samples. Care must be taken that if independent samples are desired, they can be obtained typically by thinning the resulting chain of samples by only taking every n th value. In addition, samples from the beginning of the chain may not accurately represent the desired distribution, so may need to be discarded (burn-in period).

MCMC frameworks

One popular approach for the frameworks is currently based on the **Stan**. Stan is available on Linux, OS X, and Windows and has a command line interface. It is also ported to Python, R, MATLAB, and Julia.

It compiles its models using its own compiler (`stan`) via C++ to an executable in order to produce sensible execution times. The interface from Julia (`Stan.jl`) builds the model seamlessly, and then runs the executable to dump the results of the simulation. Then, it uses a second package, the `Mamba (.jl)` package, to display and diagnose the results.

`Mamba` is a native Julia package that is a comprehensive toolset for the implementation and inference using MCMC sampling.

It provides a framework for:

- The specification of hierarchical models through stated relationships between data, parameters, and statistical distributions
- The block-updating of parameters with samplers provided, defined by the user, or available from other packages
- The execution of sampling schemes
- Posterior inference

The Mamba package is designed for general Bayesian model fitting via MCMC. Like OpenBUGS and JAGS, it supports a wide range of model and distributional specifications, and provides a syntax for model specification but differing from those. However like PyMC, Mamba provides a unified environment in which all interactions with the software are made through a single, interpreted language.

Any operator, function, type, or package can be used for model specification and custom distributions and samplers can be written in Julia to extend the package. This is of great benefit when compared to OpenBUGS and JAGS, where extensions can involve `tr` wrappers used to call the programs, their DSLs, and the underlying implementations in C++.

A comprehensive treatment of Mamba is given at mambajl.readthedocs.org. To illustrate the use of the package, let's look at an example drawn from OpenBUGS concerning a log-linear model for binary data via the Solomon-Wynne experiment on dogs.

A dog is put in a compartment, the lights are turned out, and a barrier is raised. 10 seconds later an electric stimulus is applied. The results are recorded as a success ($y = 1$) if the dog jumps the barrier before the stimulus occurs, or failure ($y = 0$) otherwise.

Thirty dogs were each subjected to 25 such trials. A plausible model is to suppose that a dog learns from previous trials, with the probability of success depending on the number of previous shocks and the number of previous avoidances. Lindley thus uses the following model:

$$pj = (A^x j) * (B^{j-x})$$

The preceding model is for the probability of a shock (failure) at the j , trial, where x_j = number of success (avoidances) before trial j and $j - x_j$ is the number of previous failures (shocks).

This is equivalent to a log linear model:

$$\log p_j = \alpha \alpha x_j + \beta (j - x_j)$$

We have included the data from the trails in the `dogs.wsv` file and we wish to test the feasibility of the model by obtaining estimates for α and β , and determining how close this fits to the actual results.

So we can read in the experimental results and setup into a `dogs` array as follows:

```
using Mamba
dogs = (Symbol => Any) [:Y => int(readdlm("dogs.wsv"))];
dogs[:Dogs] = size(dogs[:Y], 1);
dogs[:Trials] = size(dogs[:Y], 2);
dogs[:xa] = mapslices(cumsum, dogs[:Y], 2);
dogs[:xs] = mapslices(x -> 1:25 - x, dogs[:xa], 2);
dogs[:y] = 1 - dogs[:Y][:, 2:25];
```

The data goes into the `y` component and we need to create further components for the number (sizes) of dogs and trials, and slice and dice the data in order to create the following stochastic model:

```
model = Model(
    y = Stochastic(2,
        @modelexpr(Dogs, Trials, alpha, xa, beta, xs,
        Distribution[
            begin
                p = exp(alpha * xa[i,j] + beta * xs[i,j])
                Bernoulli(p)
            end
            for i in 1:Dogs, j in 1:Trials-1]), false),

    alpha = Stochastic(:Truncated(Flat(), -Inf, -1e-5)),
    A = Logical(@modelexpr(alpha, exp(alpha))), 

    beta = Stochastic(:Truncated(Flat(), -Inf, -1e-5)),
    B = Logical(@modelexpr(beta, exp(beta))) ),

    inits = [
        [:y => dogs[:y], :alpha => -1, :beta => -1],
        [:y => dogs[:y], :alpha => -2, :beta => -2]
    ],
);
```

Choosing model expressions for α and setting initial values to start off the runs, we can perform the simulation as:

```
scheme = [Slice(:alpha, :beta), [1.0, 1.0]];
setsamplers!(model, scheme);
sim =
    mcmc(model, dogs, inits, 10000, burnin=2500, thin=2, chains=2);
```

This is over 2 chains and has a long burn-in of 25 percent of the iterations; every other result (thin = 2) is used. So over two chains and for 10000 iterations, this will produce 3,750 results per chain:

```
## MCMC Simulation of 10000 Iterations x 2 Chains...
Chain 1:  0% [0:07:47 of 0:07:48 remaining]
Chain 1:  10% [0:00:47 of 0:00:53 remaining]
.....
.....
Chain 1: 100% [0:00:00 of 0:00:50 remaining]

Chain 2:  0% [0:00:17 of 0:00:17 remaining]
Chain 2:  10% [0:00:43 of 0:00:48 remaining]
.....
.....
Chain 2: 100% [0:00:00 of 0:00:49 remaining]

describe(sim)

Iterations = 2502:10000
Thinning interval = 2
Chains = 1,2
Samples per chain = 3750

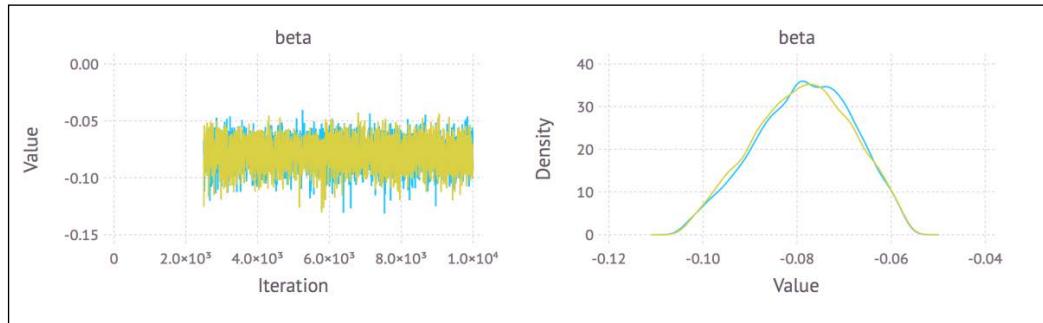
Empirical Posterior Estimates:
5x6 Array{Any,2}:
  ""      "Mean"      "SD"      "Naive SE"      "MCSE"      "ESS"
  "A"      0.782932  0.0191798  0.000221469  0.000323638  5132.34
  "B"      0.924379  0.0109682  0.000126649  0.000164441  5776.37
  "alpha"   -0.24501  0.0245322  0.000283274  0.000413172  5142.05
  "beta"   -0.0787042  0.0118855  0.000137241  0.000178353  5771.19

Quantiles:
5x6 Array{Any,2}:
  ""      "2.5%"      "25.0%"      "50.0%"      "75.0%"      "97.5%"
  "A"      0.745209  0.769936  0.783456  0.796246  0.819994
```

```
"B"      0.901388  0.917295  0.924706  0.932047  0.944471  
"alpha"   -0.29409  -0.261448  -0.24404   -0.227847  -0.198458  
"beta"    -0.103819 -0.086326  -0.0782797 -0.0703723 -0.0571298
```

Mamba also links very well with the `Gadfly` package to a graphic visualization of the fit:

```
Using Gadfly  
P = plot(sim)  
draw(p, filename="dog.svg");
```



Summary

This chapter has covered a diverse range of topics arising from the discipline of scientific computing.

We began by looking at classical linear algebra problems, the solutions of which are provided by routines from within the Julia base system. 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 partial differential equations.

Finally, we turned to an area in which the use of Julia is particularly well suited, the solution of non-vectorized problems such as those arising from stochastic processes modeled by Monte Carlo methods.

In the next chapter, we will look in greater detail at the question of production of graphics and data visualization and will see that Julia's various approaches are especially rich and diverse.

7

Graphics

It has been often noted that Julia has no built-in graphics command. 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 from source a variety of different operating systems and any libraries that are shipped, such as OpenBLAS and LibUV, must as be in source form and not interfere with the building process.

Graphics engines have a variety of different backends such as Gtk, Qt; and, whereas specialist packages may be restricted in their OS support, the overall Julia system may not.

At first, the inclusion of built-in graphics was seen as a long term goal and one that would be added in future releases. However with an emphasis on package compilation and rapid loading, this does not seem as pressing as it originally did. I will return to this topic at the end of this chapter.

One additional point to notice 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. Normally we will be working with a single package.

To tackle Julia's approach to graphics, I will classify the approaches in three main sections:

- Basic graphics
- Graphic engines
- Web graphics

The classification is far from perfect, but by basic graphics I mean where the bulk of the work is done in Julia coding, albeit utilizing calls to low-level libraries. Graphic engines rely on third-party support such as Gnuplot or Python's matplotlib and the final category utilizes web browsers and HTML.

Basic graphics in Julia

We have already encountered some graphics in the previous chapters of this book, both text plots and two-dimensional graphics using modules such as `ASCIIPlots` and `Winston`. In this section, we are going to look a little further into these and similar packages.

Text plotting

In *Chapter 1, The Julia Environment*, we had seen some simple graphics using the `ASCII plots` package.

This has largely been replaced with a second package, the `TextPlots` package. While it does not provide sophisticated visualization, it is worth a look as it is very lightweight and loads quickly, being independent of any graphic libraries or drivers.

The basic call uses a function of a single variable together with an optional range of values for that variable.

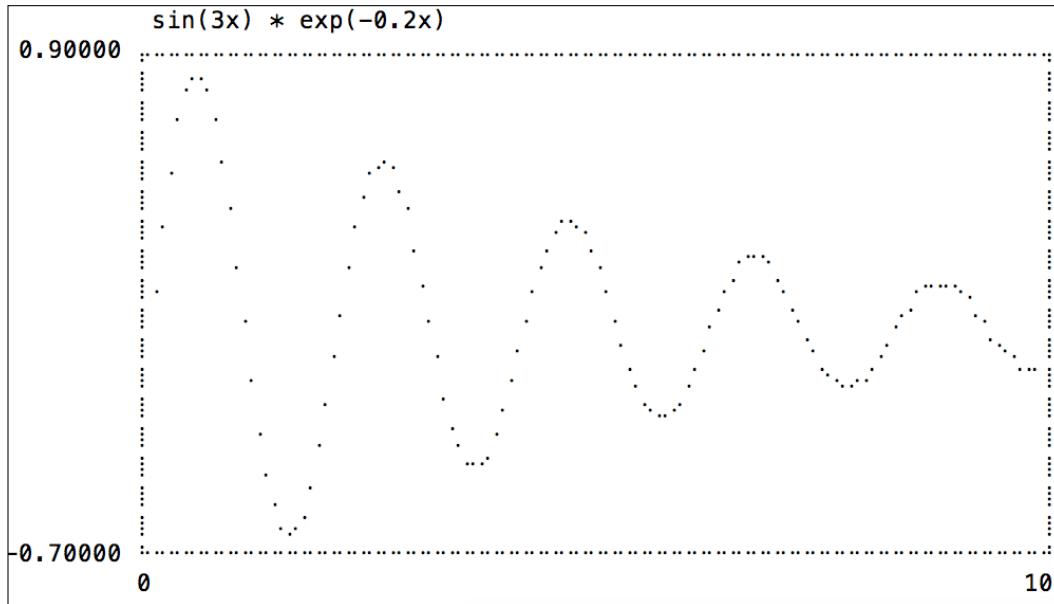
The range can be specified in the `x0:x1` form or as separate arguments `(, , , .x0, x1)`. If no range is provided, then a default of `-10:10` is used.

It is possible to define more than one function, but the display can be confusing.

Consider the following damped sinusoid in the `0:3π` range, which we pass as an anonymous function:

```
Julia> using TextPlots  
julia> plot(x->x*sin(3x)*exp(-0.3x),0,3pi)
```

The resulting plot is shown in the following figure:



The routine computes the bounding box for the generated points suitably scaled up, and provides a default title equivalent to the function definition. The limits for evaluating the function can be passed separately, as shown previously, or as a range. If the limits are not provided, the default `-10:10` is used.

Normally, the plot displays a title, labels, and a border, but all can be switched off by setting the appropriate optional parameters to `false`.

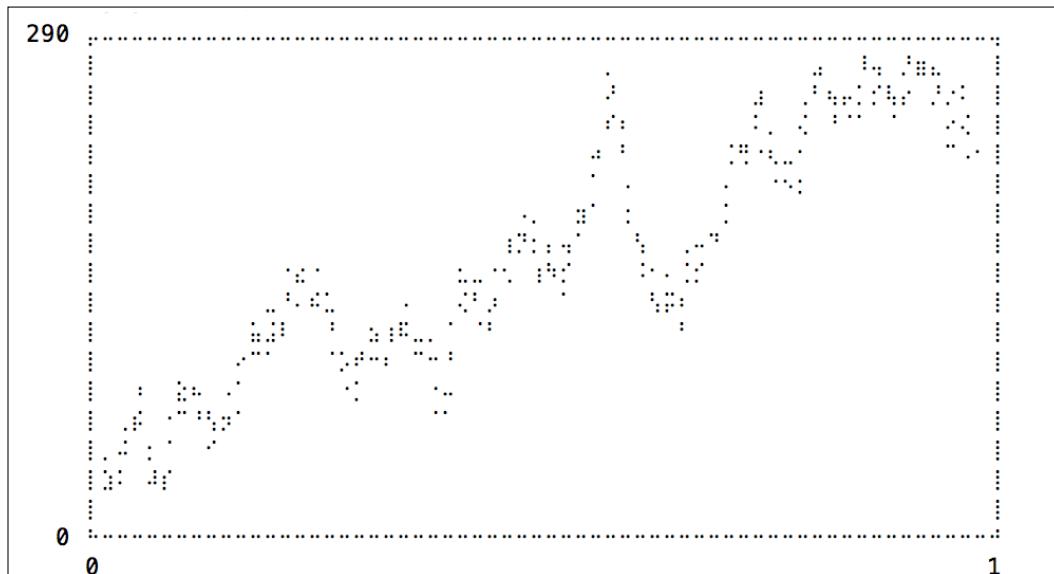
In order to control the size of the displayed text, the number of rows and columns and also a value for columns can be specified explicitly.

It is possible to define more than one function, but within the context of a text plot, the result can be often confusing.

However, passing arrays of values rather than a function definition creates a quite acceptable scatter diagram.

The following code generates a simple random walk (with drift) by generating some random numbers and displaying the cumulative sum:

```
julia> using TextPlots  
julia> t = linspace(0,1,400);  
Julia> x = 0.3*sqrt(t) + 10*randn(400);  
julia> plot(t, cumsum(x), title=false)
```



Cairo

Cairo is a 2D graphics library with support for multiple output devices and `Cairo.jl` is a pretty faithful correspondence to the C API.

This is the next level up from the base graphics as it implements a device context to the display system and works with X Window (Linux), OS X, and Windows. Additionally, Cairo can create disk files in PostScript, PDF, and SVG formats.

`Cairo.jl` is used by the `Winston` package and I find it is convenient to add it separately before installing `Winston`. The package addition script for Cairo will also install the required base software for the appropriate operating system and if this installation runs smoothly the higher level graphics should encounter no problems.

The examples provided in the Julia package correspond to those on the cairographics.org website and the documentation there on the C API is useful too.

The following is an example to draw a line (stroke) through four points:

1. First we create a `cr` context and add a 512x128 rectangle with a gray background:

```
using Cairo
c = CairoRGBSurface(512, 128);
cr = CairoContext(c);
save(cr);
set_source_rgb(cr, 0.8, 0.8, 0.8);      # light gray
rectangle(cr, 0.0, 0.0, 512.0, 128.0); # background
fill(cr);
restore(cr);
save(cr);
```

2. Next, we define the points and draw the line through the points:

```
x0=51.2;  y0=64.0;
x1=204.8; y1=115.4;
x2=307.2; y2=12.8;
x3=460.8; y3=64.0;
move_to (cr, x0, y0);
curve_to (cr, x1, y1, x2, y2, x3, y3);
set_line_width (cr, 10.0);
stroke (cr);
restore(cr);
```

3. Finally, we can add some text and write the resulting graphics to disk:

```
move_to(cr, 12.0, 12.0);
set_source_rgb (cr, 0, 0, 0);
show_text(cr,"Figure 7-2")
write_to_png(c,"fig7-2.png");
```

The result is shown in the following figure:



Winston

Winston is a 2D package and resembles the built-in graphics available within MATLAB.

The majority of the plots produced in the earlier chapters have made use of Winston, but there are a number of features that we have not yet covered.

The typical usage we have already seen is via the `plot()` function:

```
using Winston
t = linspace(0,4pi,1000);
f(x::Array) = 10x.*exp(-0.3x).*sin(3x);
g(x::Array) = 0.03x.*(2pi - x).*(4pi - x);
h(x::Array) = 1./(1 + x.^2);
y1 = f(t); y2 = g(t); y3 = h(t)
plot(t,y1,"b--")
```

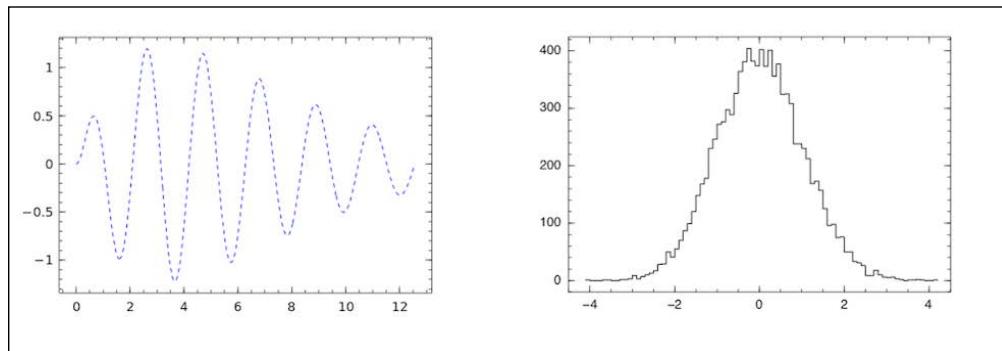
Alternatively, use `fplot()` and define the function directly that produces the same graph as shown in the following:

```
fplot(x->10*x.*exp(-0.3*x).*sin(3*x), [0,4pi],"b-")
```

In addition, there is a `plothist()` function that can take the result of the `histo()` function.

The following code generates a set of normally distributed numbers and displays the frequency histogram for 100 intervals:

```
a = randn(10000)
ha = hist(a,100)
plothist(ha)
```



When we need to plot multiple curves on a single graph, we have seen that this can be done using a single statement:

```
plot(t, y1, "r-", t, y2, "b-")
```

Alternatively, this can be done with separate statements in a couple of ways:

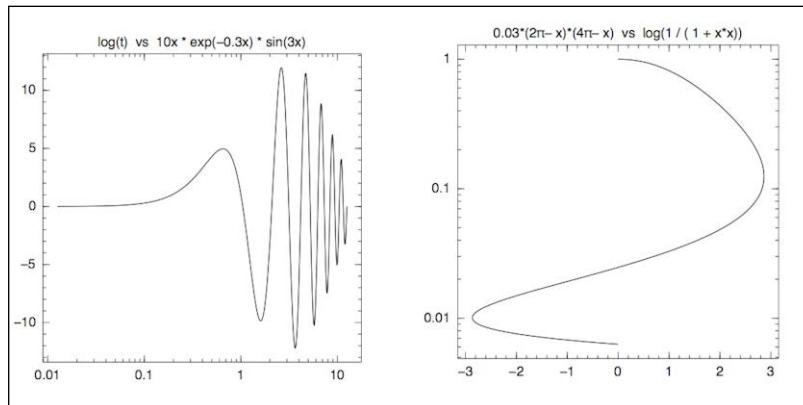
```
plot(t, y1, "r-")
oplot(t, y2, "b-")
plot(t, y1, "r-")
hold()
plot(t, y2, "b-")
```

This is particularly useful as calls such as `fplot()` do not take multiple arguments.

In addition to plotting linear scales, it is possible to plot logarithmic ones using the `loglog()`, `semilogx()`, and `semilogy()` functions:

```
semilogx(t,y1)
title("log(t) vs 10x * exp(-0.3x) * sin(3x)")
semilogy(y2,y3)
title("0.03*(2\pi - x)*(4\pi - x) vs log(1 / ( 1 + x*x))")
```

The two plots are shown in the following figure:



Notice the use of the `\\" syntax in the title statements to create non-ASCII characters using the LaTeX conventions for the symbols.`

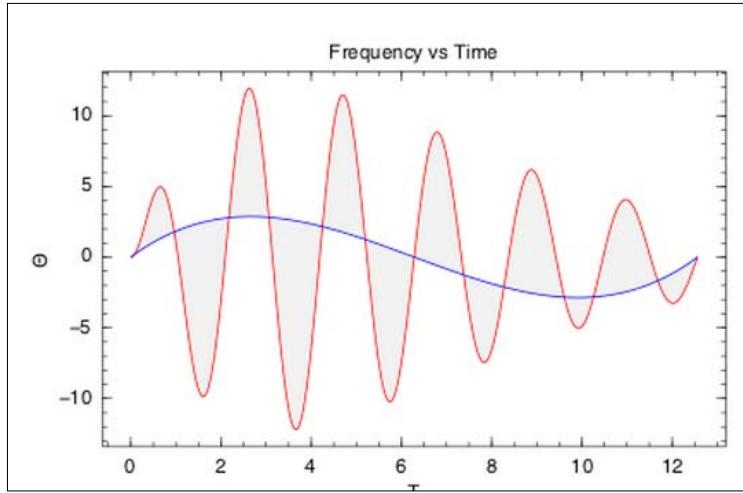
Graphics

The calls to routines such as `plot()` are convenience functions to write to the current plotting frame, the ID of which can be retrieved by a call to `gcf()`.

Working explicitly with framed plots can offer additional functionality as the following redrawing of two functions `f()` and `g()`, defined previously shows:

```
p = FramedPlot( title="Frequency vs Time",
                  ylabel="\Theta",
                  xlabel"\Tau");

add(p,FillBetween(t,y1,t,y2));
add(p,Curve(t,y1,color="red"));
add(p,Curve(t,y2,color="blue"));
display(p)
```



Finally, let's look at a more complex example using a framed plot:

```
p = FramedPlot(aspect_ratio=1,xrange=(0,100),yrange=(0,100));
n = 21;
x = linspace(0, 100, n);

# Create a set of random variates
yA = 40 .+ 10*randn(n);
yB = x .+ 5*randn(n);
```

```

# Set labels and symbol styles
a = Points(x, yA, kind="circle");
setattr(a,label="a points");
b = Points(x, yB);
setattr(b,label="b points");
style(b, kind="filled circle");

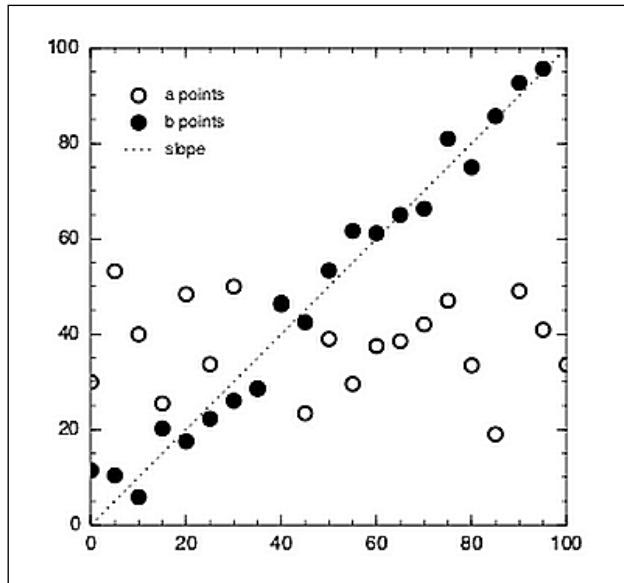
# Plot a line which 'fits' through the yB points
# and add a legend in the top LHS part of the graph
s = Slope(1, (0,0), kind="dotted");
setattr(s, label="slope");
lg = Legend(.1, .9, {a,b,s});
add(p, s, a, b, lg);
display(p)

```

The `setattr()` routine is a general routine for setting attributes to winston objects. In fact, calls to `title()`, `xlabel()`, and `ylabel()` are also convenience calls to set the appropriate attributes of the current plotting window.

As we can see, it is possible to attach a `label` attribute to an object and use that as part of the legend in conjunction with the objects style (`kind`).

The resulting graph is shown in the following figure:



In addition to displaying plot, it is possible to save it to file using `savefig()`.

The type of image saved is determined by the file extension and there are a couple of options, such as width and height (in pixels), that will rescale the plot to the desired dimensions.

Data visualization

Visualization is the presentation of data in a variety of graphical and pictorial formats. The reader will be familiar with examples such as pie and bar charts. Visualizations help us see things that are not immediately obvious, even when data volumes are very large patterns can be spotted quickly and easily.

Interactivity is one of the useful features of successful visualizations, providing the ability to zoom, filter, segment, and sort data.

In this section, we will look at the extremely powerful `Gadfly` Julia package.

Gadfly

`Gadfly` is a large and complex package, and provides great flexibility in the range and breadth of the visualizations possible in Julia. It is equivalent to the `ggplot2` R module and similarly is based on the *The Grammar of Graphics* seminal work by Leland Wilkinson.

The package was written by Daniel Jones, and the source on GitHub contains numerous examples of visual displays together with the accompanying code.

An entire text could be devoted just to `Gadfly`, so I can only point out some of the main features here and encourage the reader interested in print standard graphics in Julia to refer to the online website at gadflyjl.org.

The standard call is to the `plot()` function that creates a graph on the display device via a browser either directly or under the control of IJulia, if that is being used as an IDE.

It is possible to assign the result of `plot()` to a variable and invoke this using `display()`. In this way, output can be written to files including: `SVG`, `SVGJS/D3`, `PDF`, and `PNG`:

```
dd = plot(x = rand(10), y = rand(10));
draw(SVG("random-pts.svg", 15cm, 12cm) , dd);
```

Notice that if writing to a backend, the display size is provided and this can be specified in units of `cm` or `inch`.

Gadfly works well with C libraries of `cairo`, `pango`, and `fontconfig` installed.

It will produce SVG and SVGJS graphics, but for PNG, PS (PostScript), and PDF `cairo` is required. Also, complex text layouts are more accurate when `pango` and `fontconfig` are available.

The `plot()` call can operate on three different data sources:

- Dataframes
- Functions and expressions
- Arrays and collections

Unless specified, the type of graph produced is a scatter diagram.

The ability to work directly with data frames is especially useful.

To illustrate this, let's look at the GCSE result set we investigated in *Chapter 5, Working with Data*. Recall that this is available as part of the `RDatasets` suite of source data. So we begin by creating a dataframe as previously:

```
using Gadfly, RDatasets, DataFrames;
set_default_plot_size(20cm, 12cm);
m1mf = dataset("mlmRev", "Gcsemv")
df = m1mf[complete_cases(m1mf), :]
```

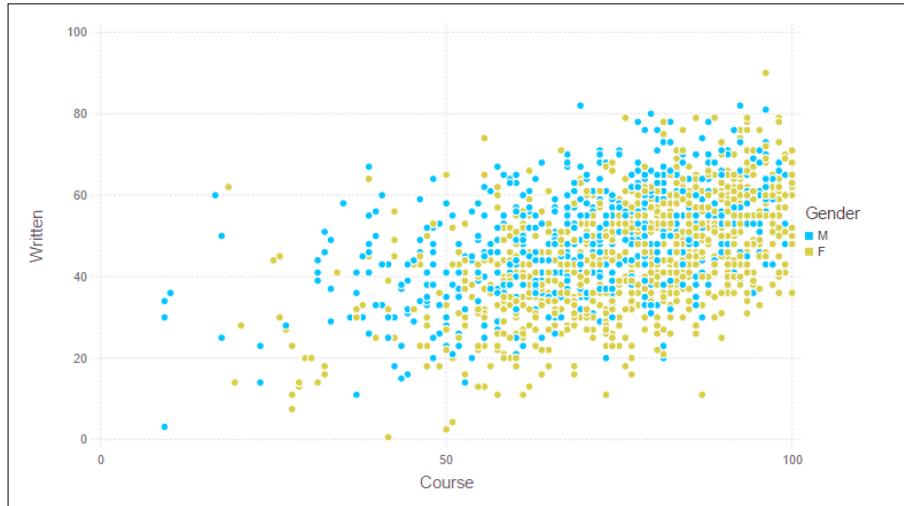
After extracting the data, we need to operate with values that do not have any `NA` values. So we use the `complete_cases()` function to create a subset of the original data.

```
names(df)
5-element Array{Symbol,1}: ;
# => [ :School, :Student, :Gender, :Written, :Course ]
```

Graphics

If we wish to view the data values for the exam and course work results and at the same time differentiate between boys and girls, this can be displayed by:

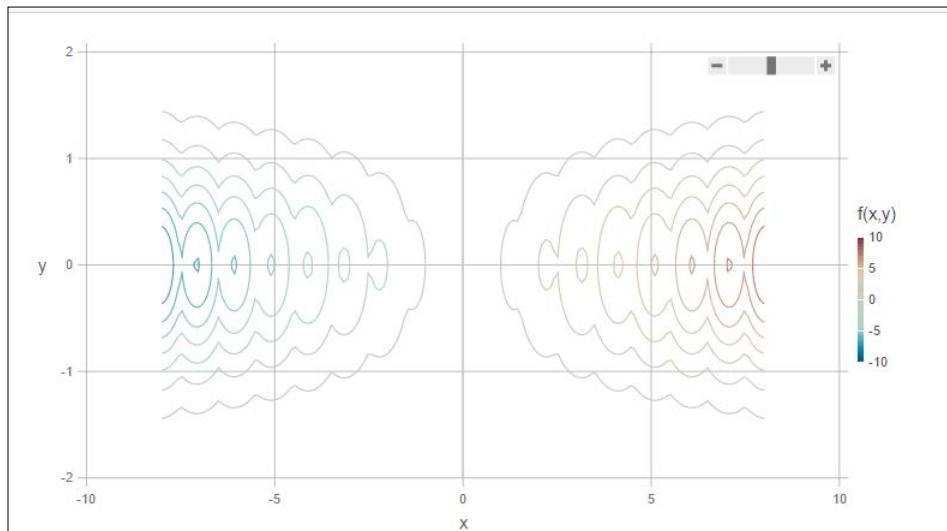
```
plot(df, x="Course", y="Written", color="Gender")
```



Notice that Gadfly produces the legend for the gender categorization automatically.

For an example of a function type invocation, here is one from the GitHub sources that shows what can be produced in a single call:

```
plot((x,y) -> x*exp(-(x-int(x))^2-y^2), -8., 8, -2., 2)
```

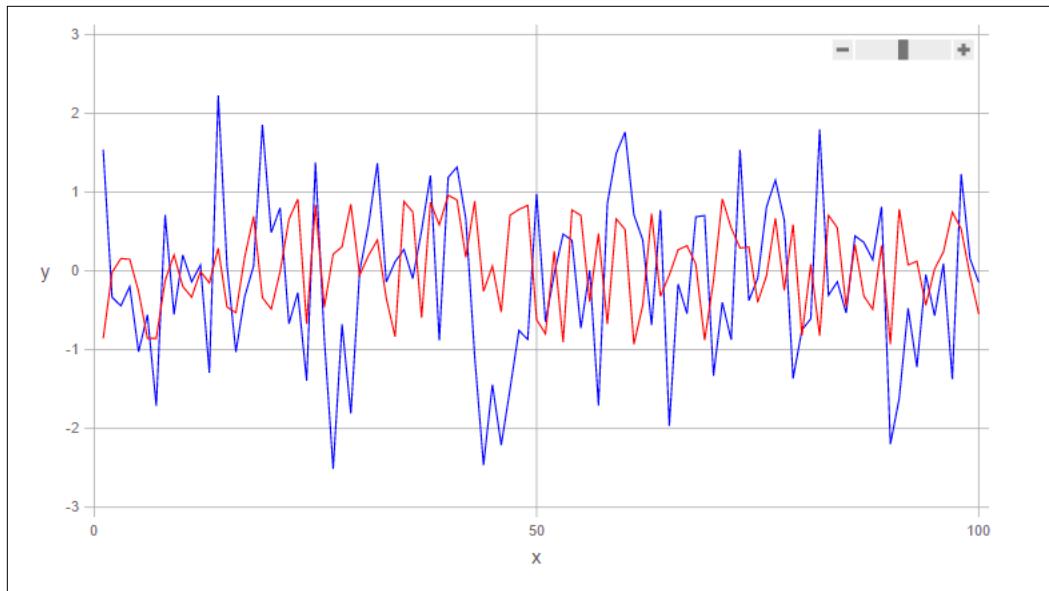


Looking at the third type invocation, let's plot two arrays of data but as line graphs rather than on a scatter diagram.

Gadfly produces multiline plots using the `layer()` routine and uses the concept of themes to overwrite the color schemes.

Here is a plot of 100 samples of a uniform variate (in red) together with a normal variate (in blue), both centered on zero and with unit variance:

```
x = [1:100];
y1 = 1 - 2*rand(100);
y2 = randn(100);
plot(
    layer(x=x,y=y1,Geom.line,Theme(default_color=color("red"))),
    layer(x=x,y=y2,Geom.line,Theme(default_color=color("blue")))
)
```



To summarize, in the spirit of the *Grammar of Graphics* seminal work, the basic form of the plot instruction is:

```
plot(data-source, element1, element2, ...; mappings )
```

In the preceding plot instruction, the `data-source` is a dataframe, function, or array: the elements are the various options such as themes, scales, plot markers, labels, titles, and guides that you need to adjust to fine-tune the plot: and the `mappings` are the symbols and assignments that link to the data.

Plot elements in Gadfly are categorized as statistics, scales, geometries, and guides and are bound to data aesthetics as follows:

- **Statistics:** These are functions that take as input one or more aesthetics, operate on these values, then output to one or more aesthetics. For example, drawing of boxplots typically uses the `boxplot` statistic (`stat.boxplot`) that takes as input the `x` and `y` aesthetic and outputs the middle, upper and lower hinge, and upper and lower fence aesthetics.
- **Scales:** Similar to statistics, apply a transformation to the original data, typically mapping one aesthetic to the same aesthetic, while retaining the original value. The `Scale.x_log10` aesthetic maps the `x` aesthetic back to the `x` aesthetic after applying a `log10` transformation, but keeps track of the original value so that data points are properly identified.
- **Geometries:** These are responsible for actually doing the drawing. A geometry takes as input one or more aesthetic, and uses data bound to these aesthetics to draw things. The `Geom.point` geometry draws points using the `x` and `y` aesthetics, the `Geom.line` geometry draws lines, and so on.
- **Guides:** These are similar to geometries. The major distinction is that geometries always draw within the rectangular plot frame, while guides have some special layout considerations such as themes, scales, plot markers, labels, titles, and guides that you need to adjust to fine tune the plot.

Compose

Compose is a declarative vector graphics system that is also authored by Daniel Jones as part of the Gadfly system, but which can be used in its own right.

Unlike most vector graphics libraries, Compose is thoroughly declarative. Graphics are defined 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`, and the assembly operation is achieved via the `compose()` function:

- `context`: An internal node
- `form`: A leaf node that defines some geometry, like a line or a polygon

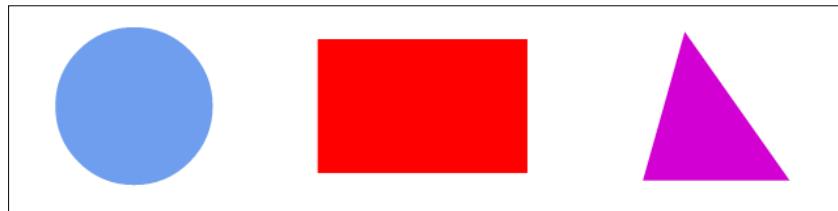
- `property`: A leaf node that modifies how its parent's subtree is drawn, such as fill color, font family, or line width
- `compose(a, b)`: This returns a new tree rooted at `a` and with `b` attached as a child

A typical invocation has a distinctly LISP-like feel.

The following code creates three filled shapes. The default context has a `cornflowerblue` fill color (the default is `black`), and the subcontexts define bounding boxes and call the `circle()`, `rectangle()`, and `polygon()` functions within those boxes.

Possible values for the fill color are given in the `Color.jl` module. Note that the default is a filled shape, but specifying `fill(nothing)` will produce a non-filled shape:

```
shapes = compose(context(), fill("cornflowerblue"),
                 (context( 0.1, 0.1, 0.15, 0.1 ), circle()),
                 (context( 0.35, 0.06, 0.2, 0.18 ),
                  rectangle(), fill("red")),
                 (context( 0.6, 0.05, 0.2, 0.2), fill("magenta3"),
                  polygon([(1, 1), (0.3, 1), (0.5, 0)])));
img = SVG("shapes.svg", 10cm, 8.66cm)
draw(img, shapes)
```



For a full discussion of drawing with Compose, the reader is directed to the package's website at <http://composejl.org/>.

Graphics

The site provides an example of building a complex drawing based on the **Sierpinski** gasket, which is a fractal and attractive fixed set with the overall shape of an equilateral triangle subdivided recursively into smaller equilateral triangles.

```
using Compose
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
```

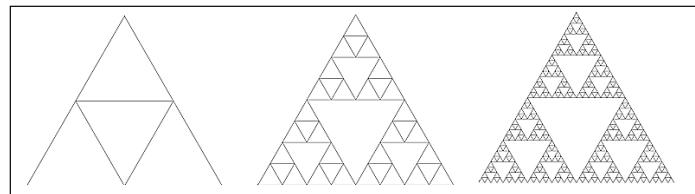
The triangle is composed using the `polygon()` function and built up recursively.

The following figure shows the result of three separate invocations for `n = 1, 3, 5`:

```
cx1 = compose(sierpinski(1), linewidth(0.2mm),
    fill(nothing), stroke("black"));
img = SVG("sierp1.svg", 10cm, 8.66cm); draw(img,cx1)

cx3 = compose(sierpinski(3), linewidth(0.2mm),
    fill(nothing), stroke("black"));
img = SVG("sierp3.svg", 10cm, 8.66cm); draw(img,cx3)

cx5 = compose(sierpinski(5), linewidth(0.2mm),
    fill(nothing), stroke("black"));
img = SVG("sierp5.svg", 10cm, 8.66cm); draw(img,cx5)
```



Graphic engines

Native graphics are constructed by developing the plots in Julia and using wrapper calls to libraries such as `cairo`, `pango`, and `tk`.

In this section, we will consider creating graphics by means of an external tasks (programs) rather than shared libraries. Clearly, this means that the program must be installed on the specific computer and not all operating systems are equivalent with respect to the various packages on offer.

PyPlot

`PyPlot` is a part of the work of Steven Johnson of MIT, which arose from the previous development of the `PyCall` module. Together with `Winston` and `Gadfly`, it is the third part in the Julia graphics triumvirate.

It provides an interface to the `matplotlib` plotting library from Python and in particular to `matplotlib.pyplot`. The API of `matplotlib` can be used as the basis for a comprehensive reference source to the various function calls: matplotlib.org/api.

Therefore in order to use `PyPlot`, the installation of Python and `matplotlib` is necessary. If this is successful, it will work either by creating an independent window (via Python) or embedding in a Jupyter 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 Windows, OS X, and Linux.

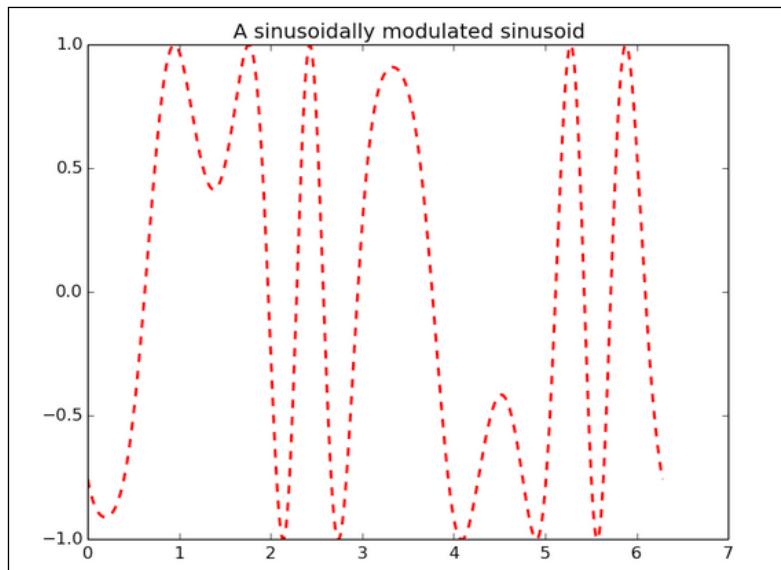
After the installation, it is important to update the distribution with the `conda` command. This command is also used for installing any additional Python modules that are not present in the standard distro. If Python is already present on the system, it is important that the execution search path is set up so that the Anaconda version is found first.

`PyPlot` makes direct calls to the API by passing data arrays to Python with very little overhead, so the production of the graphics is extremely swift.

Graphics

As an example, I have picked one from the early PyPlot documentation that of a sinusoidally modulated sinusoid. The flowing code creates the code, displays it (via a native Python window), and also writes the disk as an SVG file:

```
using PyPlot
x = linspace(0,2pi,1000)
y = sin(3*x + 4*cos(2*x));
plot(x, y, color="red", linewidth=2.0, linestyle="--");
title("A sinusoidally modulated sinusoid");
savefig("sinusoid.svg");
```

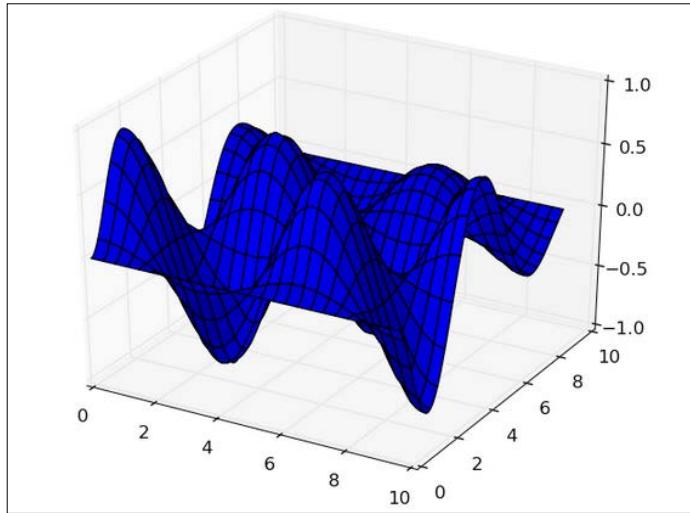


The `PyPlot` package also imports functions from Matplotlib's `mpl_toolkits.mplot3d` toolkit.

Unlike `matplotlib`, however, you can create 3D plots directly without first creating an `Axes3d` object, simply by calling `bar3D`, `contour3D`, `contourf3D`, `plot3D`, `plot_surface`, `plot_trisurf`, `plot_wireframe`, or `scatter3D`.

PyPlot also exports the MATLAB-like synonyms such as `surf` for `plot_surface` and `mesh` for `plot_wireframe`. The following is a simple 3D surface; this time displayed in an IJulia notebook:

```
y = linspace(0,3π,250)
surf(y, y, y .* sin(y) .* cos(y)' .* exp(-0.4y))
```



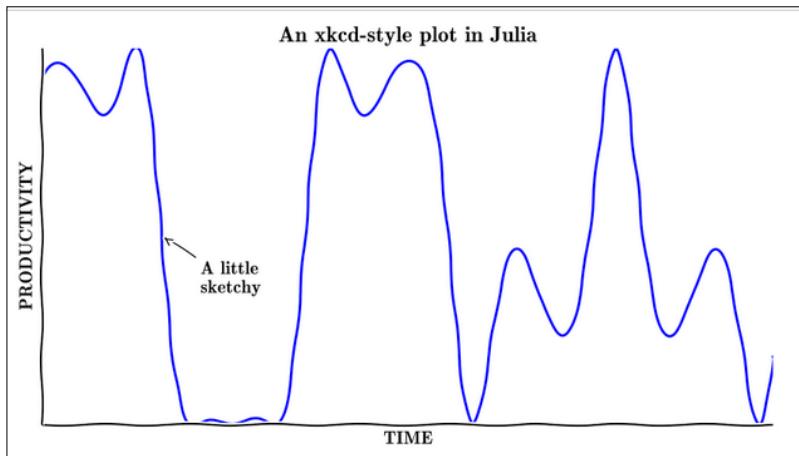
As a final example, let's create a more substantial display with axes, titles, and annotations using the XKCD comic mode.

The module includes an `xkcd()` call to switch to this mode:

```
xkcd()
fig = figure(figsize=(10,5))
ax = axes()
p = plot(x,sin(3x + cos(5x)))
ax[:set_xlim]([0.0,6])
annotate("A little\nsketchy",xy=[0.98,.001],
arrowprops={"arrowstyle"=>"->"},xytext=[1.3,-0.3])
xticks([])
yticks([])
xlabel("TIME")
ylabel("PRODUCTIVITY")
```

Graphics

```
title("An xkcd-style plot in Julia")
ax[:spines]["top"][:set_color]("none")
ax[:spines]["right"][:set_color]("none")
```



Note that plots in IJulia are sent to the notebook by default as PNG images.

Optionally, you can tell `PyPlot` to display plots in the browser as SVG images by calling `PyPlot.svg(true)` that has the advantage of being resolution independent.

Regardless of whether `PyPlot` uses SVG for browser display, you can export a plot to SVG at any time using the `matplotlib savefig()` routine.

Gaston

Gaston was written by Miguel Bazzdresch and is one of the earliest packages providing graphics in Julia. Although virtually untouched recently, it remains a flexible plotting package.

Gaston uses the `gnuplot` GNU utility program that is easily installed on Linux and also on OS X via XQuartz. On Windows there is a *sourceforge* project, but using it with Gaston is a little clunky.

With the rise of graphics via `PyPlot`, it has fallen somewhat out of favor but nevertheless is interesting when working on Linux and also architecturally. The package employs a messaging system to send commands from Julia to `gnuplot`, so all features of the latter are potentially available.

To use the system, it is necessary to set the terminal type with the `set_terminal()` routine. The default is `wxt` and is not universally available. On OS X, an alternative was `aqua` but this has been omitted since version 10.7; although it can still be added separately.

An alternative is `X11` that is a little less sophisticated than either `wxt` or `aqua`.

The terminal type can also be set to various backend file types such as SVG, PDF, and PNG. The main work is done via three levels of components such as `gaston_hilvl` (high), `gaston_midlvl` (middle) and `gaston_lowlvl` (low) plus some auxiliary functions:

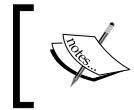
- **High:** This contains the sources for the callable routines such as `plot`, `histogram`, `surf`, and so on
- **Middle:** This contains an additional set of calls including the `llplot()` routine that is the workhorse of the system
- **Low:** This comprises just a single `gnuplot_send` call that writes the actual messages through a TCP socket (pipe) to do the actual work
- **Aux:** This is a set of additional routines to start up `gnuplot`, open a TCP pipe, validate and set the terminal type, and so on

Not all routines are exported, so it is useful to set a variable to `Gaston` (viz Python-style) and call the routines with reference to it. The package contains a useful `gaston_demo()` set of 20 different plots, two of which are reproduced as follows:

```
using Gaston
g = Gaston;
set_terminal("x11");
c = g.CurveConf();
c.legend = "Random";
c.plotstyle = "errorbars";
y = exp(-(1:.1:4.9));
ylow = y - 0.05*rand(40);
yhigh = y + 0.05*rand(40);
g.addcoords(1:40,y,c);
g.adderror(0.1*rand(40));
a = g.AxesConf();
a.title = "Error bars (ydelta)";
g.addconf(a);
g.llplot();
```

This code adds some random error bars to an exponentially decreasing function and displays the resultant graph.

The type curve is configured by a call to the `CurveConf()` routine that sets a legend and the style of plot. Axes are set up using `AxesConf()` and a title is defined. Finally, `llplot()` is called to create the display:

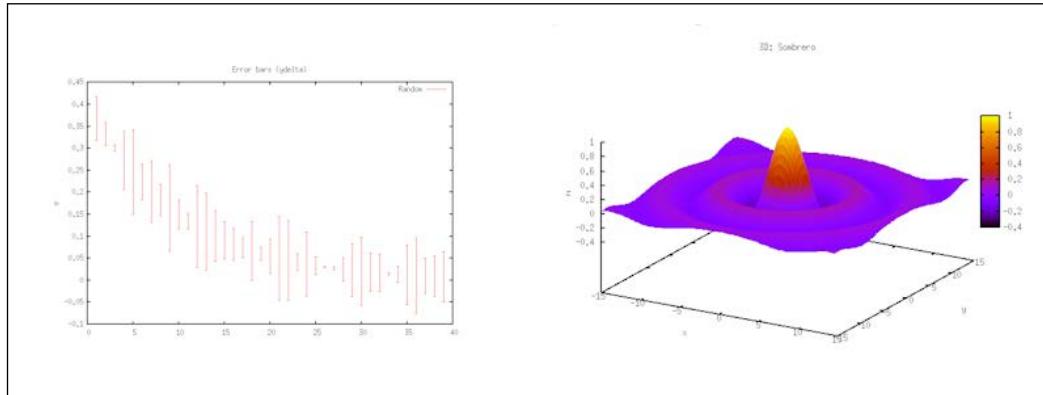


Note, as mentioned before, the use of the `g = Gaston` assignment and reference to routines via `g`, as not all functions are exported by the `Gaston` package.

```
using Gaston
g = Gaston;
set_terminal("x11");
c = g.CurveConf();
c.plotstyle = "pm3d";
x = -15:0.33:15;
y = -15:0.33:15;
Z = g.meshgrid(x, y, (x,y) ->
                           sin(sqrt(x .* x+y .* y)) / sqrt(x .* x+y
.* y));
g.addcoords(x,y,Z,c);
a = g.AxesConf();
a.title = "3D: Sombrero";
g.addconf(a);
g.llplot()
```

This second example illustrates the ability of `Gaston` to produce 3D plots. This is the Sombrero plot that is created as a $\sin(\phi)/\phi$ style curve, defined as an anonymous function and used in the `meshgrid()` routine.

The two resulting plots are shown in the following figure:



PGF plots

In contrast to `Gaston`, this is a relatively new package that uses the `pgfplots` LaTeX routines to produce plots. It integrates well with IJulia to output SVG images to the notebook.

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.

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 in order for `PGFplots` to work.

These are:

- `Pdf2svg`: This is required by `TikzPictures`
- `Pgfplots`: This is installed using a LaTeX package manager such as `texlive` or `MiKTeX`
- `GNUPlot`: This is required in order to plot contours

Graphics

The package operates by generating messages (as similar to `Gaston`) that are `LaTeX` strings, which need to be passed to the `TitzPictures` package to produce the `SVG` output.

Additionally, in order to use `TitzPictures`, `lualatex` must be installed. Note that both, the `texlive` and `MiKTeX` distributions, include `lualatex`.

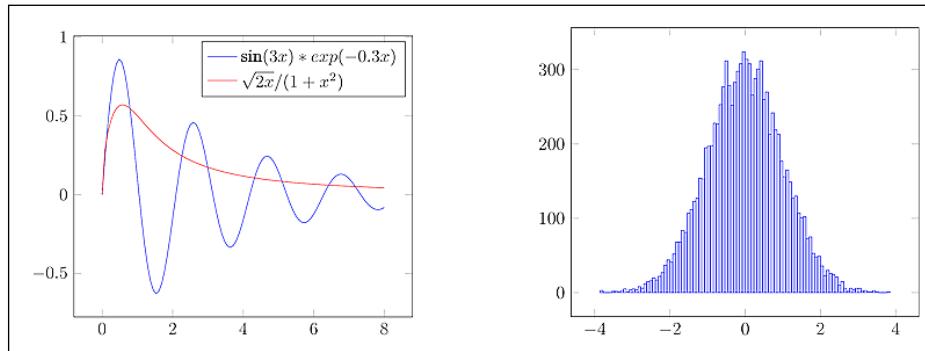
This following example generates multiple curves on the same axis and assigns their legend entries (in `LaTeX` format):

```
using PGFPlots;
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)" ] );
save("linear-plots.svg", p);
```

`Plots.Linear` is just one type of plotting style available in the `Plots` module.

It is very easy to make histograms with another type of style, the `Plots.Histogram` style:

```
fq = randn(10000);
p = Axis(Plots.Histogram(fp, bins=100), ymin=0)
save("histogram-plot.svg", p);
```



The `Plots` library also contains the `Image` and `Contour` styles, examples demonstrating their use can be found on the website: <http://pgfplots.sourceforge.net>

Furthermore, the reader is directed to the documentation on `tikz` and `pgfplots` that can provide more information about all the styles that are supported, the associated arguments, and possible values.

Using the Web

All operating systems are able to display web pages and modern standards of HTML5 and CSS3 including drawing capabilities. The addition of JavaScript client-side libraries, such as jQuery and D3, provides straightforward means to add interactive and drag and drop functionality.

These are available to some extent in the `Gadfly` package. This was classified as a native package even though it required additional of libraries such as `cairo`, `pango`, and `tk` because these were accessed by means of wrapper functions using `ccall()`.

In this section, we will explore a couple of approaches that use HTML graphics directly. The first runs on the workstation and the other is a different approach using an online system.

Bokeh

`Bokeh` is a Python interactive visualization library that utilizes web browsers for presentation.

The Julia package is written in native code. It is independent of the original Python module but works in the same fashion. The basic operation consists of creating HTML code and writing it to disk. This is then passed to the (default) browser.

The current implementation is relatively sparse when compared to the Python implementation, but is still very capable of producing basic line graphs.

All parameter values are defaulted using global parameter, but the packet includes a set of routines to change these.

The HTML output is sent to a `bokeh_plot.html` file (temporary) that will be overwritten using the `plotfile()` function.

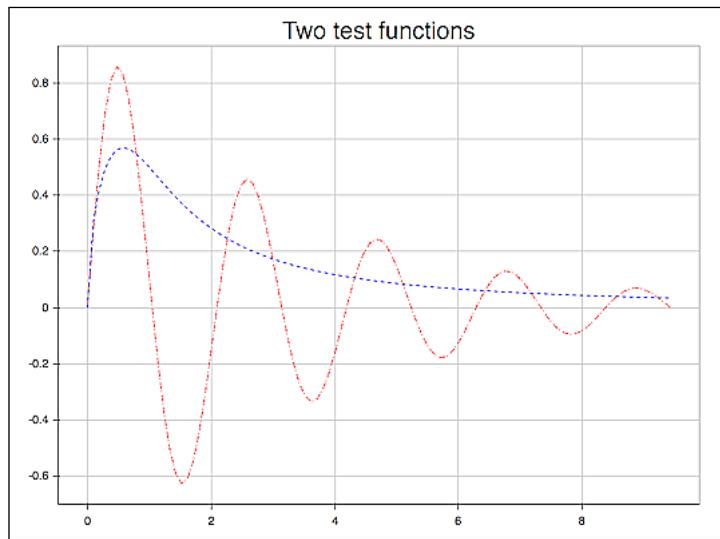
`Bokeh` will also work well with IJulia notebooks. In this case, it is necessary to call the `setupnotebook()` function that writes a CSS style sheet and set of JavaScript routines to the file.

The following code creates a plot of the two test functions that we defined when considering the `PGFPlots` package and this is run under IJulia:

```
using Bokeh  
setupnotebook()  
  
x = linspace(0,3pi)  
f1(x) = sin(3x).*exp(-0.3x)
```

Graphics

```
f2(x) = sqrt(x)./(1 + x.^2)
y = [f1(x) f2(x)]
title("Two test functions")
plot(x, y, "r;|b--")
```



The default title is changed using the `title()` function and displayed by a call to `plot()`.

In order to create a multicurve display, we pass an array of function definitions. Also, with `plot()` the array of `abscissa` (`x`) values is optional. We wish to specify a range of values, but when not provided this defaults to `1:500`; the upper limit for this range can be changed using a `countrerval()` routine.

The type of line drawn utilizes the set of glyphs we met in Winston. In order to specify multiple glyphs as a single string, use the `|` character as a separator.

Plotly

Plotly is a data analysis and graphing web application that is able to 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.

Originally, access to Plotly was via a REST API to the website <http://plot.ly> but a variety of programming languages now can access the API including Julia.

The `Plotly.jl` package is not listed (as yet) in the standard packages repository, so needs to be added using a `Pkg.clone()` statement:

```
Pkg.clone(https://github.com/plotly/Plotly-Julia)
```

To use `Plotly`, you will need to sign up for an account via <http://plot.ly> providing a unique username and e-mail address. On registration, an API key will be generated and emailed to you 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.

So all coding require a call to the `signin()` routine:

```
using Plotly  
Plotly.signin("myuserid", "abc32def7g")
```

On successful execution, the routine returns a `PlotlyAccount` data object and an online graph is created under that account by formulating and executing a response function. The response function posts the data to `Plot.ly` that creates the plot and generates a URL for it as a reply.

The following is a script to display some log-log plots. The data is passed as an array of arrays (allowing for multiple curves) and a layout array is constructed to set the axis to logarithmic.

Additionally, we need to pass a name under which the plot is to be stored and indicate that, if the script is rerun the plot can be overwritten:

```
trace1 = [  
    "x" => [0, 1, 2, 3, 4, 5, 6, 7, 8],  
    "y" => [8, 7, 6, 5, 4, 3, 2, 1, 0],  
    "type" => "scatter"  
];  
trace2 = [  
    "x" => [0, 1, 2, 3, 4, 5, 6, 7, 8],  
    "y" => [0, 1, 2, 3, 4, 5, 6, 7, 8],  
    "type" => "scatter"  
];  
data = [trace1, trace2];  
layout = [  
    "xaxis" => ["type" => "log", "autorange" => true],  
    "yaxis" => ["type" => "log", "autorange" => true]
```

Graphics

```
"yaxis" => ["type" => "log", "autorange" => true]
];
response = Plotly.plot(data,
  ["layout" => layout,
  "filename" => "plotly-log-axes",
  "fileopt" => "overwrite"]);
plot_url = response["url"]
```

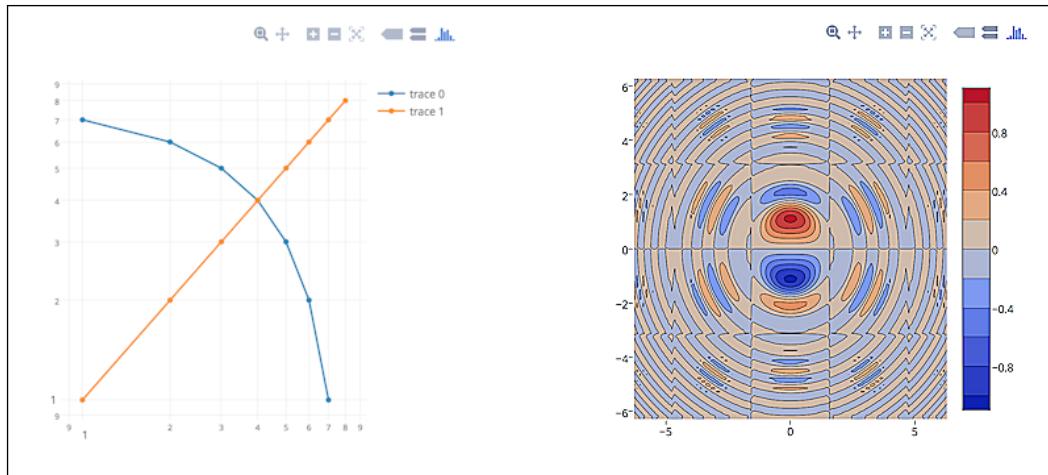
A value for `plot_url`, such as `http://plot.ly/~myuserid/17`, indicates that it is stored with ID 17 under the `myuser` ID account.

Logging on the `Plot.ly` site, you will see the plot stored as `plotly-log-axes`.

The site contains a wide variety of code examples that can be downloaded as templates for your graphics; moreover, they are tailored with your specific username and password.

As a second example, here is a contour plot of some sinusoids with a randomly generated component:

```
N = 100;
x = linspace(-2*pi, 2*pi, N);
y = linspace(-2*pi, 2*pi, N);
z = rand(N, N);
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
data = [ [ "z" => z, "x" => x, "y" => y, "type" => "contour" ] ];
response = Plotly.plot(data,
  ["filename" => "simple-contour",
  "fileopt" => "overwrite"]);
plot_url = response["url"];
Plotly.openurl(plot_url) # Display the plot via its URL
```



Once generated, it is possible online to switch from `public` to `edit` mode.

In edit mode, you can add/modify a main and the axis titles, show a legend, and apply one of the `Plotly` themes.

There is also the ability to export the plots in PNG, PDF, SVG, and EPS formats.

Raster graphics

Most of the types of displays we have been considering so far are termed vector graphics. These are defined in terms of points, curves, and shapes such as circles, rectangles, and polygons.

Working with images and colormaps is often referred to as raster graphics.

Since low-level packages eventually translate vector plots to rasters, these packages are capable of working with images directly.

So far in this book we have been working with simple netpbm format images, but in practice we will wish to operate with the more common formats such as PNG, GIF, and JPEGs.

So in this final section, we'll turn our attention to a brief overview of some of the ways you can manipulate images in Julia.

Cairo (revisited)

The Cairo package is a low-level package that we met earlier in the *Basic graphics in Julia* section and used it to create a curve between four points.

In the following example, we will create the graphics context from an `RGBSurface` method as before and fill the background with a light grey, but now we will load an image and clip it:

```
using Cairo

c = CairoRGBSurface(256,256);      # Canvas is 256x256
cr = CairoContext(c);
save(cr);
set_source_rgb(cr,0.8,0.8,0.8);    # light gray
rectangle(cr,0.0,0.0,256.0,256.0); # background
fill(cr);
restore(cr);
save(cr);
```

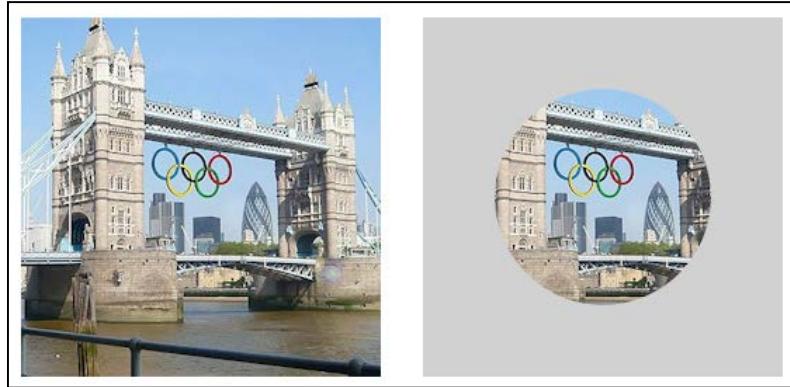
Next we create an arc of half the canvas dimensions and with length 2π , so in fact this is a complete circle. Calling `clip()` creates a window on the canvas:

```
arc (cr, 128.0, 128.0, 76.8, 0, 2*pi);
clip (cr);
new_path (cr);
```

All that now remains is to read the image from disk, and if necessary scale it to the 256x256 window size:

```
image = read_from_png ("towerbridge.png");
w = image.width;
h = image.height;
scale(cr, 256.0/w, 256.0/h);
set_source_surface (cr, image, 0, 0);
paint (cr);
write_to_png(c,"towerbridge-cropped.png");
```

The original and the resulting cropped images are shown as follows:



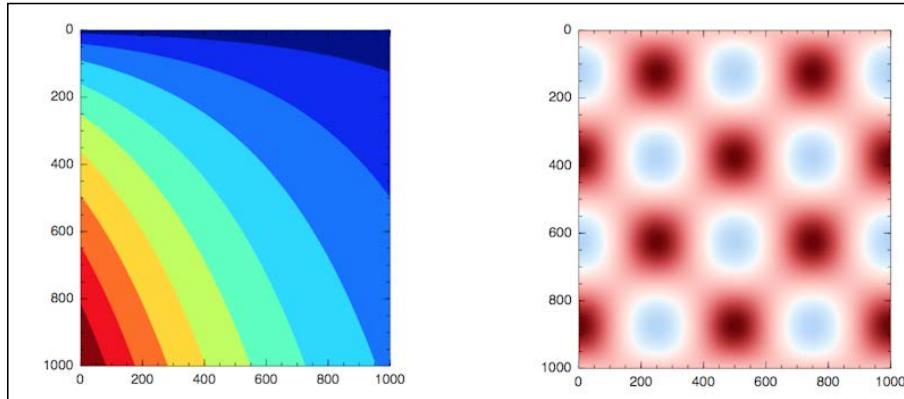
Winston (revisited)

Winston uses Cairo to create its displays and at present has some limited raster support via a couple of functions: `colormap()` to load RGB colormaps and `imagesc()` to display an image from a matrix of pixel values.

Winston defines one rainbow-style Jet colormap, but it is also capable of loading any maps defined in the Color package or indeed any maps created by the user:

```
using Winston, Color
x = linspace(0.,4π,1000);
y = linspace(0.,4π,1000);
z1 = 100*(x.^0.5).*exp(-0.1y)';
colormap("jet", 10);
imagesc(z1);
z2 = sin(x).*cos(y)';
colormap("rdbu");
imagesc(z2);
```

Here are a couple of images created on $[0 : 4\pi, 0 : 4\pi]$ using both the Jet and RdBu (from Red to Blue) colormaps:



Images and ImageView

The set of packages from Tim Holy provides the most comprehensive support for manipulating images in Julia.

The main package, the `Images.jl` package, uses the utility programs that form part of the ImageMagick suite. As such, `Images` falls in the same class of packages as `PyPlot` and `Gaston`, since it requires the presence of some specific third-party support.

The ImageMagick program needs to be on the executable path and installation of the `Images` package will attempt to install them whenever this is necessary.

There is a second package, the `ImageView.jl` package, that can be used to display an imagery to screen. When using IJulia, this is not needed as the workbook provides all that is required to display the imagery.

There is an additional package, the `TestImages.jl` package, that functions a little as `RDatasets` does in the statistics world. This is a set of common images that can be used in developing and testing Julia routines. The images are somewhat large and are not stored on GitHub, rather the installation and build process of `TestImages` will retrieve them and store them locally.

An image can be loaded with a statement of the form:

```
using TestImages;
img = testimage("earth_apollo17");
```

The image format is one of PNG, TIFF, or JPG and the extension can be supplied or omitted according to preference.

Images can also be loaded from local files or via the Internet using the `imread()` routine. Note that a few other packages define overlapping functions or types -- PyPlot defines `imread` and Winston defines `Image`.

When using both `Images` and these packages, you can always specify and fully qualify the call by using `Images.imread("earth_apollo17")`.

```
using Images, ImageView
img = imread("lena.png");
```

This code reads the grayscale image of Lena Söderberg, which we worked with earlier, from disk in PNG format. It could be loaded from the test images as `lena_grey_512`, this time as a TIFF.

Once loaded, the representation in Julia is completely equivalent for either a PNG or TIFF image

```
julia> names(img)
2-element Array{Symbol,1}:
 :data
 :properties
```

An image has two named symbols `:data` and `:properties`.

```
julia> img.properties
Dict{ASCIIString,Any} with 4 entries:
 "suppress"      => Set{Any}({"imagedescription"})
 "imagedescription" => ""
 "spatialorder"    => ASCIIString["x", "y"]
 "pixelspacing"   => [1,1]
```

In this image, the `"spatialorder"` property has value `["x", "y"]`.

This indicates that the image data is in *horizontal-major* order, meaning that a pixel at the (x, y) spatial location would be addressed as `img[x, y]` rather than `img[y, x]`.

`["y", "x"]` would indicate *vertical-major*.

```
typeof(img.data); # => Array{Gray{UfixedBase{UInt8,8}}},2}
size(img.data);   # => (512,512)
```

Graphics

The `type` indicates it is a grayscale image. An alternative when working with color images is an array of the `RGB{UfixedBase{UInt8, 8}}` data values.

The `size` array can be used to determine the dimensions of the image data:

```
w = size(img.data)[1]; w0 = w/2;
h = size(img.data)[2]; h0 = h/2;
```

The actual data can be extracted by either using `convert()` or referencing the `img.data` or `data(img)`:

```
imA = convert(Array, img);
julia> summary(imA)
"512x512 Array{Gray{UfixedBase{UInt8,8}},2}"
imB = reinterpret(UInt8, img.data);
```

This is the raw data as a byte array.

For RGB images, it will be a $512 \times 512 \times 3$ array.

Individual pixel values can be referenced by indexing and also by using slice ranges:

```
data(img)[w0,h0];
data(img)[w0-16:w0+15, h0-16:h0+15]
```

To illustrate some image processing techniques, let's define a 5×5 convolution kernel and redo the edge detect filtering on the Lena that we first demonstrated in the previous chapter:

```
kern = [ 0  0 -1  0  0
         0 -1 -4 -1  0
        -1 -4 24 -4 -1
        -1 -4 -1  0
        0 -1  0  0 ];
# => 5x5 Array{Int64,2}
imgf = imfilter(img, kern)
Gray Image with:
data: 512x512 Array{Gray{Float64},2}
properties:
  imagedescription: <suppressed>
  spatialorder: x y
  pixelspacing: 1 1
```

The original image of Lena together with the result of applying the convolution kernel is shown as follows:



Summary

This chapter has presented the wide variety of graphics options available to the Julia programmer.

In addition to the more popular packages, such as `Winston`, `Gadfly`, and `PyPlot`, we identified others such as `Gaston`, `Bokeh`, and `PGFPlots`. Although, they may be less familiar, but they offer additional approaches in the production of graphical and pictorial displays.

Also, we looked at how the `Plotly` system can be utilized in Julia to generate, manipulate, and store data visualizations online.

Finally, we discussed the means by which raster graphics and imagery 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.

9

Networking

In the previous chapter, we concluded the review of database support by looking at RESTful access. While this can be applicable to data stored in a local environment, it is more often associated with data held on servers accessed over an Internet or Intranet connection.

In this chapter, we will develop methods of working with network connections further, starting with the basic IP sockets and then looking at services that are more commonly associated with the World Wide Web (WWW).

Following this, we will consider the use of messaging services, such as e-mail and Twitter, and conclude with a brief discussion of cloud service such as those offered by Amazon and Google.

Sockets and servers

In this section, we are going to look at a basic cornerstone of network services, the socket, and see how this leads to the creation of familiar operating system tasks such as the web and e-mail servers.

First, we need to introduce the concept of well-known ports.

Well-known ports

The concept of networked services over 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 this port number.

For example, originally the file transfer protocol used port 21, SSH port 22 and sendmail port 25. Today other (additional) ports may be used for these services

One port that many readers will be familiar with is 80, this is the one 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 and we saw a few examples for some of the SQL and NoSQL servers in the previous chapter. For instance, the MySQL database commonly uses port 3306, the MongoDB server 28017, and the Redis key-value store 6379.

It is possible to start any of these servers on other ports, but the use is so widespread that the numbers assigned may be considered to be "slightly less well-known".

A network service will be started to run in a loop listening for traffic on a particular port and its purpose is to deal with any requests coming from that port. Usually when a request has been 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 this socket and in most cases it will establish a connection to facilitate bidirectional communication

It is a primary function of the operating system that it will deny the creation of additional sockets to avoid network clashes occurring that would lead to ambiguities in servicing the requests. However, for services that have to deal with a high volume of requests from different sources, such as web and database servers, it is clearly not practical to have a single resource dealing with each synchronously. So a sophisticated method of a server creating copies of itself, running in parallel, is often the norm.

UDP and TCP sockets in Julia

The IP protocol specifies two types of sockets, unreliable and reliable.

The concept of an unreliable socket may seem at first a little strange but there are some requests, which if not serviced can be merely ignored, are 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 this.

The unreliable sockets are termed as operating via the **User Datagram Protocol (UDP)** and are connectionless. Alternatively, the reliable sockets use the **Transmission Control Protocol (TCP)** and are connection full. The latter are so common that they are often merely referred to as sockets.

Julia supports both UDP and TCP sockets and also named pipes.

The source code is mainly provided in the `socket.jl` and `streams.jl` base modules. These make API calls to the Julia core and rely on the `libuv` Joyent library.

We'll look at an example of a more extensive TCP server in the next section, so let's first spend a little time with an example using UDP:

```
s1 = UdpSocket(); bind(s1,ip"127.0.0.1",8001) # => true
s2 = UdpSocket(); bind(s2,ip"127.0.0.1",8002) # => true
```

This creates two sockets to the localhost on the 8001 and 8002 ports, the operations are asynchronous so then IO does not block.

For a UDP socket the process is termed binding and the IP address of the localhost is defined by the `ip "127.0.0.1"` special string form or alternatively using the `IPv4(127,0,0,1)` function call:

```
using Dates
send(s2,ip"127.0.0.1",8001,string(Dates.now()));
```

This sends the current date and time over the `s2` socket to the `s1` socket, which is bound to the 8001 port:

```
msg = recv(s1);
bytestring(msg);
"2015-03-20T15:50:40"

close(s2);
close(s1);
```

The message is a byte stream that can be read by the `recv()` function, converted to an ASCII string, and 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()` functions to send and receive messages, TCP services need to establish a duplex channel between server and client and use `connect()`, `listen()`, and `accept()`.

There are several forms for a TCP client connection; a typical one would be a call such as:

```
connect(host::ASCIIString, port::Integer)
```

If only the port is specified, the `localhost` (`ip"127.0.0.1"`) is assumed.

As an alternative to the host string, a predefined `TCPSocket()` method may be passed as the first argument.

Connecting to a socket is simple and if successful, it will return an open status. Then it is 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 as to continue listening.

```
julia> sock = connect("ljuug.org", 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("ljuug.org")
ip"82.165.158.3"
```

Servers operating over TCP sockets use `listen()` and `accept()` to wait for connection. A common example is a simple server that accepts text inputs and simply sends it back to the client process.

The following is my take on a "Through the Looking Glass" version that reverses the text before echoing it back.

We need to munge the returns by stripping them off and reappending at the EOLs.

This is written as a script that uses the `!#` convention to indicate the specify that it must be run via Julia. The `-q` switch is necessary to suppress the Julia opening banner.

The initial part of the script has more complex arguments than with the database ETL example, and so we make use of the `ArgParse` module to simplify the parsing:

```
#! /Users/malcolm/bin/julia -q
#
using ArgParse

const ECHO_PORT = 3000
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()
```

The server defaults to the `localhost` and will listen on the `3000` port.

Also, it will echo responses to `STDOUT`. This can be suppressed with `-q` or `-quiet`, and it would be possible to provide additional information to log the information to disk file. I'll leave this as an exercise to the reader:

```
ehost = pa["server"]
eport = pa["port"]
vflag = !pa["quiet"]

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, and the processing is run asynchronously using the `@async` macro:

```
if vflag println("Listening on port $eport") end
server = listen(eport)
while true
    conn = accept(server)
    @async begin
        try
            while true
                s0 = readline(conn)
                s1 = chomp(s0)
                if 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
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 by using `chomp()` and then reappended when writing to the connection.

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:

```
./echos.jl &
# Listening on port 3000
```

Testing this is straightforward, open a connection to the port (3000) and create an asynchronous task to wait for any responses sent back over the socket:

```
sock = connect(3000)
TcpSocket(open, 0 bytes waiting)

@async while true
    write(STDOUT,readline(sock))
end
Task (waiting) @0x00007fc81d300560
```

Since this is a "Looking Glass server", it seems appropriate to test this from some poetry from *Thorough the Looking Glass, and What Alice Found There*.

The following reads the first verse of *You are Old Father William* and returns in a manner that Alice would comprehend:

```
fw = readall("/Users/malcolm/Work/Father-William.txt");
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

println(sock,"."); # This will shut down the server
```

Named pipes

A named pipe uses a special file as a communication channel.

It is a first-in, first-out (FIFO) stream that is an extension to the traditional pipe mechanism on Unix / OS X and is also present on Windows, although the semantics differ substantially.

```
epipe = "/tmp/epipe"; # => Define a file as named pipe
server = listen(epipe)
```

Testing is possible using the echo server example. It clearly requires some changes to the argument parsing as we need to replace the port information with the named pipe.

From the command line, the process is exactly equivalent except for the `connect()` call.

```
np = connect("/tmp/epipe")
Pipe(open, 0 bytes waiting)

@async while true
    write(STDOUT, readline(np))
end
Task (waiting) @0x00007fe80df9c400

println(np,"You are old Father William");
mailliW rehtaF dlo era uoY
println(np,"."); # => Signal to shut down the server
```

Working with the Web

The World Wide Web (WWW) has become a popular medium to return information. Initially this was for web browsers as a result of CGI or server-side scripting, but recently it has been for more generalized services. We saw an example of the latter with the use of REST in conjunction with the CouchDB document database.

In addition to browsers being almost universally available, we noted that most firewalls are configured to permit the traffic of HTML data. Also, programs such as `wget` and `curl` can be used to query web servers in the absence of a browser.

In this section, we will consider Julia's approach and in particular to the facilities available under the group heading, JuliaWeb.

A TCP web service

Essentially, a web server is functionally little different from the echo server developed in the previous section. After some initial set up, 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 has to be able to distinguish between them. The simplest are textual data in plain form such as HTML markup, but also there are imagery (JPEG, PNG, GIFs, and so on) and attachments such as PDFs and Word documents.

To do this, the server knows about a series of formats termed MIME types, and sends metadata information as part of a response header using the "Content-Type": key.

The header precedes the data information, such as text or imagery, and in the case of the binary information it may be necessary to specify the exact length of the datastream using some additional metadata via the "Content-Length:" key.

In order to signal the end of the header information, a web server needs to output a blank line before sending the data.

The simplest form of header will just consist of the status code followed by the content type, and in the following example I will use the `quotes` file we looked at when considering string processing to return a random quote as plain text:

```
function qserver(sock::Integer)
    fin = open("/Users/malcolm/Work/quotes.txt");
    header = """HTTP/1.1 200 OK
Content-type: text/plain; charset=us-ascii

""";
    qa = readlines(fin);
    close(fin);
    qn = length(qa);
    @async begin
        server = listen(sock)
        while true
            qi = rand(1:qn)
            qs = chomp(qa[qi])
            sock = accept(server)
```

```
    println(header*qs)
  end
end
end
```



Note that the blank line in the response header indicates,
a string is required.



The 200 status code sent is the expected `OK` value -- we will revisit status codes again in the next section.

The operation of the server program is very simple. As a prelude, this program opens the text file; ideally, the program should exit with an exception if this fails. Then it reads all the quotes, which are all one-liners, into an array and gets the number of entries (`length`) of the array.

The server action is just to generate an appropriate random number in range, look up the quote, append the response header, and return it.

Because the server is running asynchronously, it is simple to test it from the REPL:

```
qserver(8000); # Run it on port 8000 as port 80 may be in use
conn = connect(8000);
HTTP/1.1 200 OK
Content-Type: text/plain

Sailing is fun but scrubbing the decks is aardvark
```

The JuliaWeb group

Most web servers are constructed to respond to requests issued for a browser, rather than for a utility program such as `curl`.

Recall that a web server response may arise when a URL such as `google.com` is entered or as the result of submitting information, and clicking on a button or image. The former is termed a `GET` operation and the latter a `POST` operation. In fact, there is a set of other operations such as `PUT` and `DELETE` that are more common in RESTful services than in browsers.

We noted previously that the server is required to send back a response code and that the most common one is `200`, corresponding to `OK`. However, there are others that may occur, such as in case of redirection and error, one being the infamous **404, Page not found**.

So construction of an HTTP server requires considerable more effort than it may have appeared from the preceding example. Couple this with the fact that *POSTing* data may be denied and the `GET` operations may be expected to handle query strings, and the programmer may be pleased to have some assistance in constructing both web servers and web clients.

To this end, a series of routines grouped as `JuliaWeb` is available.

We used one of these `HTTPClient` previously while exploring RESTful services, which used the underlying `LibCURL` package to do much of the heavy lifting.

However, there is a separate subset of interrelated **Julia Webstack** packages and a separate web page is available at <http://juliawebstack.org> that provides some summary information and links to the individual GitHub repositories.

It is worth noting that the underlying request-response mechanism uses the `GnuTLS` package that is a wrapper around a shared library. The build process using the `Pkg` manager on, for example, Ubuntu Linux platform is not always straightforward and it requires additional development modules that the base distro does not provide. If you have built Julia from source, then these modules should be present. On the Red Hat/CentOS flavors of Linux, I have encountered no problems and also on the Windows operating system

The installation of `GnuTLS` under OS X uses Homebrew and I have had problems where brew becomes 'borked' and requires removing and reinstalling. If `GnuTLS` is not present then virtually the entire set of Julia Webstack routines is not available, but it is still possible to work with `HTTPClient` and the `curl` library as a method of generating HTTP requests.

In the rest of this section, I'll briefly give an overview of the Julia Webstack family of modules routines.

In the order of high-level to low-level functionality, these are:

- `Morsel.jl`: This is a Sinatra-like routable web framework
- `Meddle.jl`: This is a rack-like request middleware stack
- `WebSockets.jl`: This is an implementation of the websockets protocol
- `HttpServer.jl`: This is a basic HTTP service over TCP
- `HttpParser.jl`: This is a wrapper for Joyent's `http-parser` lib
- `HttpCommon.jl`: This is a set of shared types and utilities

There is also an additional `Requests.jl` related package that can be used to deliver HTTP requests, similar to the `HTTPClient` request, but using the `GnuTLS` library as a means of transport.

I'll look at Morsel and WebSockets in the next two sections.

Using `Requests` is very simple; here is an example that gets the London Julia website's home page:

```
using Requests;
ljuug = "http://www.londonjulia.org";
resp = get(ljuug)
Response(200 OK, 10 Headers, 19289 Bytes in Body)
```

This returns a string that gives the status, number of response headers, and length of the HTTP body (in bytes). All these are available as fields in the response object.

```
names(resp)'
4-element Array{Symbol,2}:
 :status      :headers      :data      :finished
```

Here, of interest, is the response header that is returned as a dict hash:

```
julia> resp.headers
"Date"          => "Thu, 19 Mar 2015 10:30:03 GMT"
"http_minor"    => "1"
"Keep-Alive"    => "1"
"status_code"   => "200"
"Server"        => "Apache"
"X-Powered-By"  => "PHP/5.4.38"
"Content-Length"=> "19289"
"http_major"    => "1"
"Content-Type"  => "text/html"
"Content-Language"=> "en"
```

And also, the actual data that is returned by the web server as a XHTML document:

```
resp.data
"<?xml version=\"1.0\" encoding=\"utf-8\"?><!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN\" \"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">\r\n<html xmlns=\"http://www.w3.org/1999/xhtml\" xml:lang=\"en\" lang=\"en\"\r\n>\r\n<head>\r\n<title>London Julia Users Group</title>\r\n<meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\" />\r\n</body>\r\n</html>\r\n"
```

As we saw with `HTTPClient`, the `Requests` package is capable of appending query strings to a `GET` request and emulating web forms when executing `POST` requests. This makes `requests` a viable alternative in working with RESTful systems.

For example, this executes a Google search for the London Julia (ljuug) group:

<https://www.google.co.uk/?q=ljuug>

The "quotes" server

In this section, I will look at the development of a more comprehensive "quotes" server using the Julia Webstack routines, similar to the simple one we saw earlier.

In order to add information, such as author and category, we will use the SQLiteDB quotes database created in the previous chapter and return the quote as formatted HTML text.

We will want to restrict the client requests to only those using `GET`, and not `POST`, `PUT` and so on. It is possible to deal with a query string such as `?cat=Politics` to select a quote from a particular group and I have included this within the code accompanying this book.

Deciding on how the server responses to client requests, and indeed with the validity of a request, is called routing. In our case all `GET` requests will be routed to the same routine and any others denied.

The server needs to do a little start up work in opening the quotes database and establishing the number of quotes available. This is similar to using the simple text file but a little more elaborate.

Random quotes are returned now with three fields: Author/Category/Text and these are marked up as HTML5, all inline, although a fully-fledged web server would probably refer to a separate CSS style sheet to decorate the text.

The function of the `Morsel` package is to provide a framework for complex routing. It is quite easy to do this using `HttpServer` directly, but `Morsel` will allow us to differentiate between `GET` and other requests:

```
using Morsel;
using SqliteDB
db = SQLiteDB("quotes.db");
try
    res = query(db,"select count(*) from quotes");
catch
    error("Can't find quotes table");
end
nq = res[1][1];
@assert nq > 0;
```

The first function of the server is to open the `quotes.db` database and get a count of total number of quotes. SQLite has the habit of creating empty databases when they do not exist, so we should check for their existence. However, we will get an exception error while querying the `quotes` table, if it does not exist, so effectively this confirms the database's existence.

Also, we need to check that there are some entries in the table, otherwise all the queries will give no result. Since we wish to output HTML markup to the browser, we define a `htmlH` header string and a `htmlF` footer string:

```
htmlH = """<!DOCTYPE html>
<html lang='en-US'>
<head>
<title>Julia Quotes Server</title>
</head>
<body>""";
htmlF= "</body></html>";
```

In our case, creating the web server using `Morsel` is easy and we use the `route()` function to handle any requests other than `GET` requests:

```
app = Morsel.app();
route(app, POST | PUT, "/") do req, res
    htmlB = "<h2>Posting data to the server is not allowed</h2>";
    string(htmlH,htmlB,htmlF);
end
```

To handle acceptable queries we could use `route()` again, but there are convenient short forms for various request types, so in this case we can use `get()`:

```
get(app, "/") do req, res
    qid = rand(1:nq);
    sql = "select q.author,c.catname,q.quoname from quotes q ";
    sql *= "join categories c on q.cid = c.id and q.id = $qid";
    res = query(db,sql);
    author = res[1][1];
    catname = res[2][1];
    quotext = res[3][1];
    htmlB = "<p>$quotext<br/><i>$author ($catname)</i></p>";
    string(htmlH,htmlB,htmlF);
end
```

The payload is exactly equivalent to that previously used. We generate a random number in range and extract the quote with that ID, together with the author and category.

Then we need to format the body text (as `htmlB`), append the header and footer, and return the result to the browser.

Finally, we start the server by specifying the port number:

```
start(app, 8000)
```

If we point a browser to `http://localhost:8000`, we should get a random quote from the such as:

To love oneself is the beginning of a lifelong romance.
Oscar Wilde (Books and Plays)

WebSockets

Probably all readers will be familiar with the process where when typing in the search box of Google, for example, the server returns a set of suggestions and they change as the search term is refined. The method used is called **Asynchronous JavaScript and XML (AJAX)**, although the mnemonic is largely anachronistic as far as the use of the XML term is concerned.

What is clear is that the overhead using HTTP protocols, in terms of client request and server response headers, is very large while providing such small payloads, perhaps hundreds of bytes of headers for just a few tens of bytes of information.

Web sockets were devised as a way of overcoming these problems.

The name "web socket" gives the impression that it's a traditional socket. In practice, it combines the parts of UDP and TCP; it is message-based such as UDP, but is reliable like TCP. WebSockets are upgraded from HTTP/1.1 through a system previously defined as part of the HTTP specification (which is, the upgrade header).

Through the HTTP handshake, it becomes feasible to proxy the connection without losing information. WebSockets send messages that can be ASCII or UTF-8 text, but also can comprise of binary.

Each message can also carry control information and is wrapped in a frame. Messages sent from the client to the server are obfuscated with a basic transmission mask of 4 bytes that is sent within each web socket package rather than the traditional AJAX overhead.

Conveniently, a WebSocket will use the normal HTTP/HTTPS ports, that is, 80 and 443 respectively. However, there are problems when operating over the Internet in that many firewalls will block the connection. This has lead to a slow uptake in their use on the Web as compared with conventional AJAX. In general, HTTPS connections are more successful than plain HTTP ones.

WebSockets are ideal for vehicles for chat services, as an alternative to the more usual ones using IRC and JABBER. A good example is provided in the package source of `Websockets.jl`.

Instead we will look at a version of the ELIZA program. ELIZA was a computer program written at MIT by Joseph Weizenbaum around 1965 and was considered at the time as an early example of primitive natural language processing. In essence, it was a simple chat program where the user types a question to ELIZA and the program acts as a therapist by replying to the query.

The concept was simple; ELIZA has a set of "keywords", and if the user query contains one of the words, a suitable ambiguous response is selected, otherwise an especially vague reply is given.

The following is the Julia version that uses WebSockets to keep the connection open. The user signals the end of the session by typing `Bye` or `Goodbye`.

Only a small set of keywords have been included and these are hardcoded into the source code, so that these plus the responses are a dictionary hash. In practice, these would be more extensive, probably contained in a text file that is read during startup.

To be a more convincing therapist, a large set of keywords should be used.

For greater flexibility, ease of maintainability, and better memory usage, a key-value store could be used and Redis makes a very good choice.

On startup, only the keys are read into an array using the `KEYS *` Redis command, and in case of a match the appropriate response string is retrieved from the server rather than being in memory:

```
using HttpServer
using WebSockets

const SDEF = "Tell me more ...";
const SBYE = "Goodbye, please let's talk again soon";

d = Dict();

d["hello"] = "Hi, my name is Eliza";
d["no"]    = "Tell me more about the detail.";
d["yes"]   = "Tell me more about the detail.";
d["you"]   = "Let's not talk about me.";
d["think"] = "Why do you think that?";
d["hate"]  = "So you hate something, tell me more.";
d["what"]  = "Why do you ask ?";
```

```
d["want"] = "I'm here help you with what you want";
d["need"] = "We all need many thing, is this is important?";
d["why"] = "Remember therapy is good for you.";
d["know"] = "How do you know that?";
d["can't"] = "Don't be negative -- be positive!";
d["never"] = "Don't be negative -- be positive!";
d["unhappy"] = "Why are you unhappy?";
d["like"] = "Why do you like that?";
d["help"] = "I'm here to help you.";

looping = true;
```

The SDEF and SBYE strings are used for default and closing responses.

The looping flag is used so that the `bye` keyword can be used to exit the chat loop. We have set up a few keyword primitives in a hash dictionary and these reference all other responses when a known word is used.

The main loop generates a WebSocket and pattern matching is very simplistic. It uses case-insensitive comparison but would not differentiate between words that are substrings of longer ones. For example, we use the `"unhappy"` word, so we can't have `happy` too. Also, if we had used `happy` then a sentence such as `"I'm really unhappy"` might result in an inappropriate reply:

```
wsh = WebSocketHandler() do req,clt
    while looping
        s0 = lowercase(read(client))
        s1 = SDEF
        for k in keys(d)
            s = lowercase(k)
            if s == "bye"
                looping = false
                s1 = SBYE
            elseif contains(s0, s)
                s1 = d[k]
                break
            end
        end
        write(clt, "ELIZA> $s1")
    end
end
```

Running ELIZA merely consists of creating a `server` object and running it on a convenient port:

```
server = Server(wsh)
run(server, 8080)
```

Testing with WebSockets needs less involvement. It can be done using some JavaScript as part of a web browser. Most browsers now support WebSockets, although probably the best browser is Google Chrome, which provides an extension: "Simple Websocket Client".

The following is a typical dialogue when chatting with ELIZA:

```
Hello
ELIZA> Hi, my name is Eliza
I need some help
ELIZA> I'm here to help you
I am so unhappy
ELIZA> Why are you unhappy?
I have problems coping with daily life
ELIZA> Tell me more ...
Bye, I'll call back tomorrow
ELIZA> Goodbye, please let's talk again soon
```

Messaging

Messaging is older than the web services and in some ways easier to provide. In this section, I'm first going to look at the elder statesman of messaging services: e-mail.

Nowadays, we also use systems that send SMS textual information and we will look at two of these: Twitter and Esendex.

E-mail

Sending e-mails conventionally happens on the well-known port 25, although other ports such as 465 and 587 are now more commonly used (<http://www.esendex.co.uk>).

This uses the **Simple Mail Transport Protocol (SMTP)**, which was first formulated in the early 1980s and consists of formulating a message the SMTP server can understand, with fields such as **To:**, **From:**, and **Subject:** together with the text of the message and then deposit in the mail service's outbound queue.

Receiving emails is a little different. This depends on one of the different protocols such as the **Post Office Protocol (POP)** or alternatively the **Internet Message Access Protocol (IMAP)**. Each has its own related set of ports, and in addition an e-mail client program may use a different server to receive e-mails from the SMTP server that it uses to send them.

In addition, we usually want the transmissions between e-mail client and server to be encrypted using SSH/TLS-type methods and this includes other well-known ports and some form of user/password authentication.

As part of JuliaWeb, Julia has a `SMTPServer.jl` package. This uses `LibCURL.jl` as its underlying method of sending e-mails and I've included an example in this book's source code.

Rather than discussing this, we will include some code that uses a common Python module, in a way similar to how we communicated with the MySQL database in the previous chapter:

```
using PyCall;
@pyimport smtplib;
fromaddr = "malcolm.sherrington@gmail.com";
toaddrs = "malcolm@amisllp.com";

messy = """From: $fromaddr
To: $toaddrs
Subject: Test SMTP using PyCall

Testing - 1,2,3
""";

# Note that the blank line is necessary to distinguish
# the SMTP header from the message text.

username = fromaddr;
password = "ABCDEF7890"; # Not my real password

server = pycall(smtplib.SMTP,PyAny,"smtp.gmail.com:587");
server[:ehlo]();
server[:starttls]();
server[:login](username,password);
server[:sendmail](fromaddr,toaddrs,messy);
server[:quit]();
```

This code follows a fashion that is equivalent to the script in Python, the main difference being the nature of the function versus method calls between the two languages. For reference, the equivalent of a Python script is included with this book's code examples.

Returning briefly to the question of receiving e-mails via an e-mail "client" program, the POP-style of delivery is the much simpler to implement than IMAP. This can be done in Julia using the `LibCURL` package, although as yet there is no higher level package for POP to match that from SMTP.

Alternatively, the Python method can also be used by utilizing the `poplib` module, similar to the one in the preceding code that used `smtplib`. Again much of the coding in Julia can be deduced from first coding and testing in Python.

Twitter

`Twitter.jl` is a Julia package designed to work with the Twitter API v1.1. Currently, only the REST API methods are supported, although the author promises streaming methods, so please check. To understand the full functions, the reader is referred to Twitter API at <https://dev.twitter.com/overview/documentation>.

Twitter authentication used to be very simple, but now requires OAuth endpoints (see <http://oauth.net>) to provide connections, and to use this package you will need to pick up an API key from <https://dev.twitter.com/>. It is free, but you do need to set up an app on the dev site to have the key generated.

The `Twitter.jl` source on GitHub also has a pretty comprehensive `tests.jl` script that can be used to exercise the various function calls, once you have acquired a valid API key.

In the example I have chosen, I'm using the Twitter account for `@LondonJuliaUG`, the London Julia User Group, but the key has been changed from that used to run the code:

```
using Twitter;
twitterauth("12345678ABCDEFGH123456789",
"1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890abcdefgefhn"
"1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890abcdefgefhn"
"1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890abcde");
```

This constructs a structure of the `TWCRED` type that consists of 4 keys. Authentication occurs when a function is called, so it is only at this time we can determine that the authentication is correct.

In the case of an error, it returns a standard response structure via the `Requests.jl` package, with a status of `401 Unauthorized`.

We can confirm the name of the Twitter account from the `account_settings`:

```
settings = get_account_settings();
twit_name = settings["screen_name"]
println("This account is $twit_name");
This account is LondonJuliaUG
```

The following snippet gets the last set of tweets to the London Julia UG account using the `home_timeline()` function:

```
home_timeline = get_home_timeline();
typeof(home_timeline); # => Array{TWEETS,1}
length(home_timeline); # => 20
```

This is an `Any` array of the `TWEETS` type that has a large set of properties:

```
names(home_timeline[1])
29-element Array{Symbol,1}:
```

Some of the interesting ones are `:text`, `:user` and `:created_at`.

Many of the `TWEET` properties consist of arrays, which themselves may consist of hash dictionaries and these may point to other arrays as values. So navigation of the data returned is not straightforward. In particular, the `:user` structure is very complex. In order to explore this, one approach is to pick up the dataframe for `home_timeline[1]` and use `names()` to work the way down.

As an illustration, here is a dump of the last 20 tweets (today) to `@LondonJulia`:

```
for i in 1:length(home_timeline)
    df = DataFrame(home_timeline[i])
    user = df[:user][1]["name"]
    created = df[:created_at][1]
    text = df[:text]
    @printf "%2d %s : %s =>\n%s\n\n" i user created text
end
1 The Julia Language : Sun Mar 03 04:28:59 +0000 2013 =>
("RT @dyjh: Free #OpenSource in #QuantFinance event 02.04. London filling
up quickly http://t.co/ujByhPEwcE #python #julialang #rstats #scala...")
2 Data Science London : Sat Jan 28 11:55:07 +0000 2012 =>
("Shelloid: open source IoT real-time big data platform w/ Node.js &
Clojure https://t.co/HSyEXH9cwx cc @yoditstanton")
```

SMS and esendex

As a final example in this section, I will consider sending messages to mobile phones via a service called esendex.com. The REST API is lightweight and easy to use. Messages are sent and received as XML documents rather than in JSON format.

Using the APIs is possible to send, track, and receive SMS messages, although receiving requires additional setup. It is also possible to send voice as well as text messages, and if a message is routed to a phone that does not accept SMS, this is automatically converted from text to voice.

The service is not free but esendex does provide a 7-day free trial with up to 25 credits.

Registration is via an email address, which serves as the username. The system associates this with an account number and generates a password, both of which can be changed by logging on to the esendex website.

Then it is easy to test simple messaging by sending an SMS to your mobile with the ECHO service, which will confirm that it is set up correctly, but will cost you one credit!

For the developer, a number of APIs are available including SOAP and REST.

The first step is to try and send a text `t` directly from the shell using `curl`:

```
malcolm> MSG=<?xml version='1.0' encoding='UTF-8' ?>
<messages>
<accountreference>EX123456</accountreference>
```

```
<message>
<to>447777555666</to>
<body>Every message matters</body>
</message>
</messages>

curl -u Malcolm@amisllp.com:passwd -X POST -d $MSG \
https://api.esendex.com/v1.0/messagedispatcher
```

The syntax of this message is very simple, it consists of a `<to>` field corresponding to the mobile phone number and a `<body>` tag for the message text, of up to 140 characters per credit.

The whole also needs to include the account reference name and it is possible to send a series of messages within a single API request. This is *POSTed* to RESTful message dispatcher service and authorization is provided using basic authentication.

The system returns a message ID that can be used for tracking purposes. The status of the message is also available by logging on the esendex website.

Also, there is a set of SDKs in many popular languages but not yet one in Julia.

However, the above procedure can be used in Julia by spawning a task or using either `HTTPClient` or `Request` to dispatch the message:

```
using UriParser, Requests

eacct = "EX123456"
uname = "malcolm@amisllp.com"
pword = "passwd"

dispatcher = "api.esendex.com/v1.0/messagedispatcher"
uri = URI(string(https://,uname,'@',pword,'/',dispatcher))

message = """?xml version='1.0' encoding='UTF-8' ?>
<messages>
<accountreference>EX123456</accountreference>
<message>
<to>447777555666</to>
<body>Every message matters</body>
</message>
</messages>"""

post(uri; data = message)
```

Cloud services

The Julialang web home page states that *Julia is designed for parallelism and cloud computing*. We looked at running tasks in parallel and in this section, I'll close with a discussion of the cloud computing.

Using the Cloud is a relatively new computing business model, based on the concept of providing software as a utility service.

The idea is to call on resources as needed to meet demand and to pay for it as required in a way similar to utilities such as electricity and water, and therefore you do not have to provision your own resources to meet an estimated peak loading. This is often termed as **Software as a Service (SaaS)**.

Two of the early cloud providers were Amazon and Google. Both have slightly different emphasis of approach, but either can be used to run Julia, and we will consider each briefly in a Julia context later.

Among the other major cloud providers are Rackspace, Microsoft, and Oracle:

- Rackspace uses an open source system OpenStack, which uses Swift for parallelization processing and is becoming increasingly popular over Hadoop with the non-Java fraternity.
- Other providers, such as City Cloud and IBM Bluemix, are also using OpenStack and as an OS project you have the means to build your own corporate cloud. Currently it comes with machine instances for both Ubuntu and Red Hat Linux distributions.
- The Microsoft Cloud service is called Azure and is naturally very useful when running .NET and SQL Server applications. It does have some virtual machines based on Ubuntu Linux that run in a MS Windows server environment.
- Oracle Cloud is a relatively new entry to the marketplace. In addition to its flagship database, Oracle now has acquired Java and database products such as MySQL and Berkeley DB, now rebadged. As one might expect, the Oracle Cloud is extensively based on Java services.

If you have been working on JuliaBox, this works on Amazon Web Services by provisioning Docker "sandboxed" containers --so you have been using Julia in the cloud!

Introducing Amazon Web Services

Amazon is possibly the best known provider of cloud services for an individual or a small company. They are popular starting points.

To sign up only requires an Amazon account with an associated credit/debit card, which most users (of Amazon) may already have. Furthermore, Amazon offers a package for a year, which comprises of one small virtual machine. Amazon terms this as the *micro-instance*. Also, it offers a set of the most useful services termed as its "Free Tier". So it is possible to try Julia on AWS at no cost.

An instance (<http://aws.amazon.com/ec2/instance-types/>) is a type of virtual machine based on the processor type, number of cores, available memory, and runs from micro to extra-large. Naturally, bigger instances are priced at a higher rate.

Instances can be started by incorporating a choice of operating systems, including Debian/Ubuntu, Red Hat Enterprise, and CentOS, in addition to Amazon's own brand of Linux.

Once a running instance is started, additional software can be added, which does not come with the standard OS distro. Since at present this includes Julia, it is necessary to install it as a binary or build it from source, and then add in any required packages. This only needs to be done once, since the whole can be saved as an **Amazon Machine Instance (AMI)**, stored and the AMI used to spin up (create) new instances.

AWS comes with a very large set of services including SQL and NoSQL databases, data warehousing, application, and networking applications.

When using the free tier, the common services available are as follows:

- **S3:** Simple Storage Service
- **EC2:** Elastic Compute Cloud
- **DynamoDB:** Fast flexible NoSQL database

Among the non-free services, **Elastic Map Reduce (EMR)** is worth some further study.

Control of AWS, including procedures such as starting and stopping instances, copying files to and from S3, can be achieved via a web-based command program.

Also, there are various APIs and SDKs available, all fully discussed in the documentation (<http://aws.amazon.com/documentation/>) pages. Note that all of the previously mentioned services have RESTful interfaces and also there is a Python SDK, *boto*, with which it may be possible to communicate via *PyCall*.

In addition, Amazon provides a set of command-line tools that are useful for operations on files and instances, so in Julia the ability to execute CLI tasks directly offers another way to interact with AWS.

The AWS.jl package

At the time of writing the `AWS.jl` package is described as "work in progress", requesting more testing and users feedback.

It is worth noting that this package is concerned with running a virtual machine on AWS, not just running Julia, however, the two are not mutually exclusive.

The AWS package covers the S3 and EC2 services. The basis of the function calls is an AWS environment structure that is passed as the first argument:

```
type AWSEnv
    aws_id::String      # AWS Access Key id
    aws_seckey::String   # AWS Secret Key
    ep_host::String      # region endpoint (host)
    ep_path::String       # region endpoint (path)
    timeout::Float64     # request timeout (secs)
    dry_run::Bool         # dry run flag
    dbg::Bool            # print request and raw response
end
```

All these arguments are named parameters in the `AWSEnv` constructor, so need not be specified, in this case they revert to their default values.

A typical (full) call may be as follows:

```
AWSEnv(; id=AWS_ID, key=AWS_SECKEY, ep=EP_EU_IRELAND,
        timeout=0.0, dr=false, dbg=false)
```

The `AWS_ID` and the `AWS_SECKEY` are generated on signing up for AWS and need to be provided for (simple) authentication purposes. Clearing other forms of authentication is possible based on TLS public/private keys.

`EP_EU_IRELAND` is a constant defined in AWS as `ec2.eu-west-1.amazonaws.com` and is the AWS area where we keep our files.

The constructor will split is on `/` into host and path, so in the preceding call.

```
ep_host = ec2.eu-west-1.amazonaws.com
ep_path = ""
```

The AWS.S3 module uses the REST API and communication is via `HTTPClient` and not via `requests`; so it depends on `LibCURL` and not `GnuTLS`, although this may change in future releases of the package.

For storage associated with a specific account, S3 uses the concept of "buckets". A bucket is a collection of objects that we may think of as analogous to a folder, that is, as a collection of files.

An AWS account can create up to 100 buckets. When creating a bucket, we need to provide a bucket name and AWS region in which it is to be created.

The following code will create a private bucket and an object (file) in the bucket:

```
using AWS
using AWS.S3
env = AWSEnv(timeout = 90.0)
bkt = "amis1lp_bucket_1"; # Name needs to be 'globally' unique
acl = S3.S3_ACL(); acl.acl="private"
resp = S3.create_bkt(env, bkt, acl=acl)
resp = S3.put_object(env, bkt, "first_file", "Hi there")
```

The return is of the package defined with the `S3Response` type:

```
type S3Response
    content_length::Int
    date::String          # The date and time S3 responded
    server::String         # Server that created the response
    eTag::String
    http_code::Int
    delete_marker::Bool   # Common Amazon fields
    id_2::String
    request_id::String
    version_id::String
    headers::Dict          # All header fields
    obj::Any
    pd::Union(ETree, Nothing)
end
```

The `http_code` field contains the status of the call and the `obj` payload.

If the `obj` response is XML representing a Julia S3 response type, then it is parsed and assigned by the package, otherwise it contains an `IOBuffer` object.

So the following will indicate the success of the call and the returned data:

```
println("$(resp.http_code), $(resp.obj)")
```

The following sequence will add a second file, list the bucket, then retrieve the first file, delete both files, and finally delete the bucket:

```
resp = S3.put_object(env, bkt, "second_file", "Blue Eyes ")
resp = S3.get_bkt(env, bkt)
resp = S3.get_object(env, bkt, "first_file")

myfiles = [S3.ObjectType("file1"), S3.ObjectType("file2")]
resp = S3.del_object_multi(env, bkt,
S3.DeleteObjectsType(myfiles))
resp = S3.del_bkt(env, bkt)
```

Now let's consider support for the EC2.

EC2 has two sets of APIs: a simple API that provides limited functionality and a low-level API that directly maps onto Amazon's **Web Services Description Language (WSDL)** for the service.

Here, we will just consider the simple API, although `AWS.jl` also has some support for WDSL.

Currently, the following are available and their purpose is reasonably clear from their function names:

- `ec2_terminate`
- `ec2_launch`
- `ec2_start`
- `ec2_stop`
- `ec2_show_status`
- `ec2_get_hostnames`
- `ec2_instances_by_tag`
- `ec2_addprocs`
- `ec2_mount_snapshot`

The general call to a `ec2_basic()` basic function has the following form:

```
ec2_basic(env::AWSEnv, action::String, params::Dict{Any, Any})
```

This bundles together the `Dict` `params` with an EC2 request. The keys in `params` must be those as listed in AWS EC2 documentation, values can be basic Julia types, dicts, or arrays.

Running a typical computational task on EC2 to use Julia may involve most or all of the following steps:

1. Set up the AWS configuration.
2. Create a `config.jl` file specifying EC2 related configuration.
3. Launch (or Start a previously stopped) a bunch of EC2 instances.
4. Optionally log in to one of the newly started hosts and perform computations from an EC2 headnode.
5. Or execute from a Julia session external to EC2.
6. Terminate (or stop) the cluster.

As mentioned, when looking at S3, `AWSEnv()` requires `AWS_ID` and `AWS_SEC` and also an AWS Region. Rather than hardcoding the keys, it is possible to include them as part of an `.awssecret` file in the home directory.

The configuration file will require the following entries:

```
ec2_ami = "ami-0abcdef"; # Specify the AMI to use.
ec2_install_julia = true; # If true install from
ec2_sshkey = "xxxxxx"; # SSH key pair to use
ec2_sshkey_file = "/home/malcolm/keys/keyfile.pem" # Location of private
key
ec2_insttype = "m1.large"; # AWS type of instance
ec2_instnum = 21 # Number of instances
workers_per_instance = 8; # Workers per instance
ec2_julia_dir = "/home/aalcolm/julia/usr/bin" # Julia installed path
ec2_clustername = "mycluster" # An EC2 cluster name.
```

`AMI.jl` has a series of help scripts that interpret the `config.jl` file and encapsulate the desired EC2 operation.

These include `launch.jl`, `start.jl`, `stop.jl`, and `terminate.jl`.

Also, there is a `setup_headnode.jl` script that creates `addprocs_headnode.jl` to copy the scripts into `/home/malcolm/run` of one of the EC2 nodes, prints instructions to connect, and execute from the headnode.

Using these scripts from the localhost, that is machine external to EC2, would be similar to the following:

- Change and edit the `config.jl` script:
 - `julia launch.jl`
 - `julia setup_headnode.jl`

- ssh into a headnode
- Then working at the headnode:
 - cd run
 - julia
 - julia include("compute_headnode.jl")
 - work ...

After all work is done, exit from the host and from localhost, and it is important to run `julia terminate.jl`, otherwise the instances will continue to run and rack up charges.

The Google Cloud

Google Cloud services offer a realistic alternative to Amazon to run Julia in the cloud. Originally, it seemed to me that the two were quite different but now they seem to be converging in what they provide to the user, even if not how they provide it.

In fact when thinking of big data, Google were first on the scene with their academic paper on "big table" and the `BigTable` database is available in a fashion similar to Amazon's `DynamoDB`.

In a way similar to Amazon, if you have a Google account (Gmail/Google+) and provide billing details, then you have access to the Google Cloud.

Google uses the term "engine" to describe the services they provide and there are two distinct "flavours" of engine:

- App Engine
- Compute Engine

Google describes the App Engine as a PaaS. It is possible to develop applications in Java, Python, PHP, or Go without the need for operating system administration and servers management, nor to be concerned with load balance and elasticity.

Combined with the target development languages, there are existing frameworks such as Spring, Django, webapp2, and applications have access to Google core services such as mail, calendar, and groups.

The Google Compute Engine is based on a different paradigm much more akin to that of Amazon Web Services.

Virtual servers can be provisioned as Linux Virtual Machines based on either Ubuntu or Centos operating systems, and Google provides a "gallery" of well-known O/S databases including Cassandra, MongoDB, and Redis, plus popular web applications such as WordPress, Drupal, Moodle, and SugarCM. Both Apache and Tomcat web servers are also available.

Moreover, applications on the App Engine can communicate with the Compute Engine.

To control the Compute Engine, Google provides a web-based developer console. Using this is relatively straightforward, to create an instance specify the machine type and operating system plus the location (zone) of the Google Cloud.

Also, there is a command line set of utilities bundled as an SDK, and after installing and configuring it with the developer console, you can create the instance using the `gcloud` command.

In addition to the SDK, there is a `butil` package that includes Hadoop-specific configuration tools, such as the deployment tools, Datastore, and BigQuery connectors.

To use Julia on an instance requires some installation, either as a binary or building from source. Binaries are now available for both Ubuntu (using the `apt-get` command) and CentOS (using `yum`), and it is also possible to grab a distro from the Julia GIT repository and build it from source. Remember that the supplied operating systems do not come with all the prerequisite tools and libraries, so it is necessary to grab those first.

For example, on Ubuntu we may need to add `gfortran`, `g++`, `ncurses`, and so on, so it is a good idea to test the build procedure on a local machine before trying to execute the same on a Compute Engine instance.

```
sudo apt-get install g++ gfortran m4 make ncurses-dev
git clone git://github.com/JuliaLang/julia.git
make OPENBLAS_USE_THREAD=0 OPENBLAS_TARGET_ARCH=NEHALEM
```

Once Julia is built, without errors, it is possible to SSH to the instance and run it. Then access the package manager to download any required packages and copy across any code.

If you have created a multicore instance, the maximum is currently 32, it is possible to use the Julia parallel processing macros to test running tasks in a highly parallel fashion.

Parallel and distributed processing is one of the strengths of Julia and is being actively and rigorously pursued at the current time by the Julia Parallel community group. The reader is referred to their web pages at <https://github.com/JuliaParallel> for a full list of the workings of the group.

Recall that earlier we computed and displayed a Julia set.

Since each point is computed independently of its neighbors, it is possible to adapt the procedure to operate on separate sections of the image.

If we construct a distributed array, specify what function to use and what the array's dimensions are, it will split up the task automatically:

```
function para_jset(img)
    yrange = img[1]
    xrange = img[2]
    array_slice = (size(yrange, 1), size(xrange, 1))
    jset = Array(UInt8, array_slice)
    x0 = xrange[1]
    y0 = yrange[1]

    for x = xrange, y = yrange
        pix = 256
        z = complex( (x-width/2)/(height/2) ,
                     (y-height/2)/(height/2) )
        for n = 1:256
            if abs(z) > 2
                pix = n-1
                break
            end
            z = z^2 + C
        end
        jset[y - y0 + 1, x - x0 + 1] = uint8(pix)
    end
    return jset
end
```

This function computes the pixel shading for the section of the imagery on which it is operating. Let's save it in a `para_jset.jl` file.

Assuming we have a 4-core CPU (such as an Intel x86), we can start Julia by using a command such as `julia -p 3` or use the `addprocs(3)` command later to utilize the additional cores.

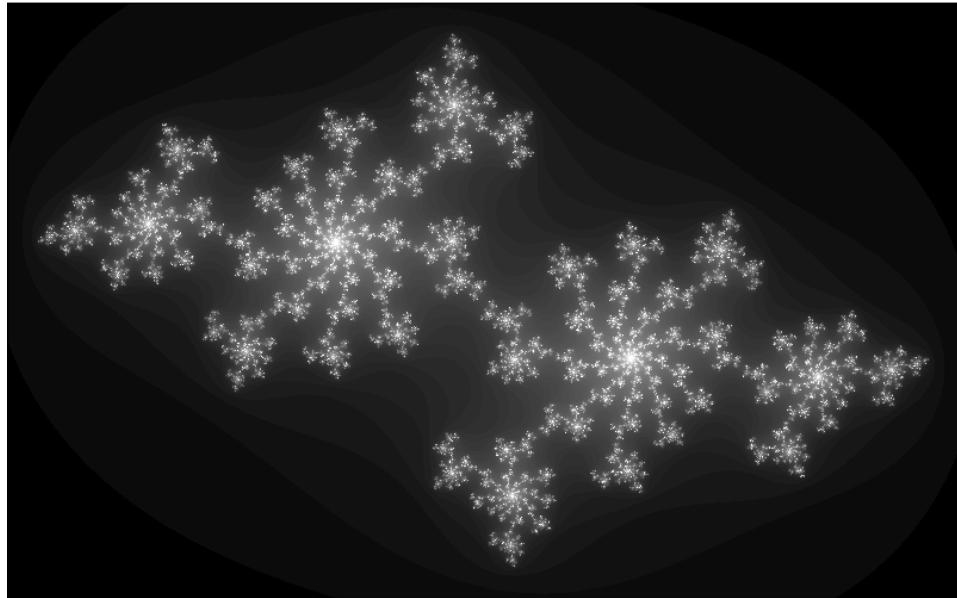
We can then generate the complete image by using the following script:

```
using Images
require("para_jset.jl")

@everywhere width = 2000
@everywhere height = 1500
@everywhere C = -0.8 - 0.156im
dist_jset = DArray(para_jset, (height, width))
full_jset = convert(Array, dist_jset)
imwrite(full_jset, "jset.png")
```

The `DArray()` constructor will split the data depending on the number of processors in the image, so we just need to ensure that the height, width, and constant C are set on all the processors and run `para_jset()` on each slice.

The `convert()` function will reassemble the distributed array to give the full image:



Summary

In this chapter, we outlined the means by which Julia can be used to develop and interact with networked systems.

First, we looked at services that use the UDP and TCP protocols, and then we looked at how to implement a simple echo server. Also, we investigated the emerging subject of WebSockets and continued by considering messaging systems involving e-mail, SMS, and Twitter.

Finally, we concluded with an overview of running Julia on distributed systems in the cloud and ended with an example of asynchronous execution to speed up (appropriately) the generation of a Julia set.

In the next chapter, we will be returning to explore some more advanced features of the Julia language by going under the hood, examining code generation and the low-level API, testing, and profiling. We will finish with a discussion of community groups and highlight the work of a couple of these groups that match some of my personal and professional interests.

8

Databases

In *Chapter 5, Working with Data*, we looked at working with data that is stored in disk files, looked at plain text files, and also datasets that take the form of R datafiles and HDF5.

In this chapter, we will consider data residing in databases but will exclude the "big data" data repository as networking and remote working is the theme of the next chapter.

There are a number of 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.

It is not the intention of this chapter to delve into the working 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 section is to look at the connection and interfacing techniques of a few common examples.

A basic view of databases

In this section, we are going to cover a little bit of groundwork to work with databases. It's a large topic, and many texts are devoted to specific databases and also to accessing databases in general.

It is assumed that the reader has knowledge of these, but here we will briefly introduce some of the concepts that are necessary for the topics in the remaining sections of this chapter.

Packages are being developed and modified at an ever increasing rate, so I urge the reader to review the availability for access to a specific flavor of database, which may not be available at the time of writing, with reference to the standard Julia package list at <http://pkg.julialang.org> and also by searching at <http://github.com>.

The red pill or the blue pill?

A few years ago discussing databases was simple. One talked 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, but the whole world was relational. Even products that were not relational, such as Microsoft Access, tried to present a relational face to the user.

We usually use the term 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. Google produced a "white" paper proposing its **BigTable** solution to deal with large data flows, and the classification of NoSQL databases was born.

Basically, 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. A common analogy is a set of sheets in an Excel file, readers familiar with Excel will recognize this parallel example.

NoSQL data stores are often deemed to be schemaless. This means it is possible to add different datasets with some fields not being present and other ones occurring. The latter is a particular difficulty with relational databases, as the schema has to be amended and the database rebuilt. Data stores conventionally comprising of text fall into this category, as different instances of text frequency throw up varying metadata and textual structures.

There are, of course, some databases that are termed NoSQL, but nevertheless have quite a rigorous schema and SQL-like (if not actual conventional SQL) query languages. These usually fall into the classification of columnar databases and I will discuss these briefly later in the chapter.

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 ones can be scaled horizontally, that is, distributed over several machines, which in the era of *Big Data* is responsible for their new-found popularity.

Interfacing to databases

Database access is usually 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" and aimed at a single user. A well-known example of this is SQLite and we will discuss the Julia support package for this in the next section.

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 **ACID (atomicity, consistency, isolation and durability)**. 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 rule**, that is **Consistency, Availability, and Partition Tolerance**. We are speaking of data stores spread over a number of 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:

1. A DBMS will be bundled as a set of query and maintenance utilities that communicate with the running database via a shared library. This is exposed to the user as a set of routines in an application program interface (API). The utilities will often be written in C and Java, but scripts can also be written in Python, Perl, and of course Julia. In fact Julia with its zero-overhead `ccall()` routine is ideal for this form of operation, and we will term the packages that use this as their principal mode of interface as wrapper packages. Often there is a virtual one-to-one correspondence between the Julia package routines and those in the underlying API. The main task of the package is to mimic the data structures that the API uses to accept requests and return results.

2. 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 individual database. If a driver is available, then the coding is the same regardless of the database that is being accessed. The first such system was **Open Database Connectivity (ODBC)** developed by Microsoft in the 1990s. This was an early package and remains one of the principal means of working with databases in Julia. There are couple of other intermediate layers: **Java Database Connectivity (JDBC)** and **Database Interface (DBI)**. The former naturally arises from Java and necessitates a compatible JNDI driver, whereas the latter was introduced in Perl and uses DBD drivers that have to be written for each individual database and in each separate programming language. All three mechanisms are available in Julia, although ODBC is the most common.
3. When we considered graphics in the previous chapter, one mode of operation was to abrogate the responsibility of producing the plots to Python via `matplotlib` using the `JavaPlot` package. The same approach can be utilized for any case where there is a Python module for a specific database system, of which there are many. Routines in the Python module are called using the `PyCall` package, which will handle the interchange of datatypes between Julia and Python.
4. The mode access is 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. The most common form of messaging protocols are called **Representational State Transfer (RESTful)**, although in practice it is possible to use similar protocols, such as SOAP or XMLRPC. Certain database APIs may expose a shared library and also a REST-type API. The REST one is clearly an advantage when accessing remote servers, and the ubiquitous provision of HTTP support in all operating systems makes this attractive, especially in situations where firewall restrictions are in place.

Other considerations

In the remainder of this chapter, I'll endeavor to cover all the preceding means of access to database systems looking at, in most cases, a specific example using a Julia package.

There is one class of systems that is outside the scope of this chapter and this is where the dataset essentially comprises a set of separate text documents organized as a directory structure (folders and file), which can be implemented either as a physical (real) or logical directory structure. Such a system is less common, but EMC's Documentum system may be seen as an example of this type.

Also, we will not be dealing in detail with XML-based database systems, as we discussed the general principles when we looked at working with files in *Chapter 5, Working with Data*, and the reader is encouraged to re-read that chapter if dealing with XML data.

In terms of the mechanism for storing XML data, these are sometimes held individually as part of a directory structure as records in a document data store, such as MongoDB, or alternatively as **Character Large Objects (CLOBS)** in a relational database, such as Oracle.

What we need to be concerned with is the speed of retrieval for queries. Exhaustive searches on large datasets would lead to unacceptable performance penalties, so we look at the underlying database to provide the necessary metadata and indexing, regardless of the means by which the data is stored.

One example that links these together is BaseX, which is an open source, lightweight, high performance, and scalable XML Database engine and incorporates XPath/XQuery processing. There is a REST API and several language implementations, although there is not yet one that is implemented in Julia. However, Python does have one such module: the `PyCall` module, mentioned previously and this can be employed here when working with BaseX.

Relational databases

As we have said, the primary difference between relational and non-relational databases is the way data is stored. 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 smallest 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 relational databases are on a single server and partition tolerance is not an option, in terms of the CAP classification they are consistent and accessible.

Building and loading

Before looking at some of the approaches to handling relational data in Julia, I'm going to create a simple script that generates a SQL load file.

The dataset comprises a set of "quotes", of which there are numerous examples online. We will find this data useful in the next chapter, so we will create a database here.

There are only three (text) fields separated by tabs and separate records per line, for example:

```
category <TAB> author <TAB> quote
```

Some examples are:

```
Classics, Aristophanes You can't teach a crab to walk straight.  
Words of Wisdom Voltaire Common sense is not so common.
```

For the script (`etj.jl`), the possible choices for the command line will be:

```
[julia] etl.jl indata.tsv [loader.sql]
```

The second choice is:

```
[julia] etl.jl [-o loader.sql] [-d '\t'] indata.dat
```

The [] designates an optional argument, and I've made the Julia command optional as (under OS X and Linux) the script can be marked as executable using the **shebang** convention (see following section).

The second format is more flexible as it is possible to specify the field separator rather than assuming it is a tab.

We wish to create a table for the quotes and another for the categories. This is not totally 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 definition of a foreign index (with corresponding DB 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,
    catname varchar(40) not null,
    primary key(id)
);
```

```
create table quotes (
    id      integer not null,
    cid     integer not null,
    author  varchar(100),
    quoname varchar(250) not null,
    primary key(id)
);
```

The simpler case of our command line script can be implemented as:

```
#!/usr/local/bin/julia
# Check on the number of arguments, print usage unless 1 or 2.
nargs = length ARGS;
if nargs == 0 || nargs > 2
    println("usage: etl.jl infile [outfile]");
    exit();
end
# Assign first argument to input file
# If second argument this is the output file otherwise STDOUT
infile = ARGS[1];
if nargs == 2
    outfile = ARGS[2];
    try
        outf = open(outfile, "w");
    catch
        error("Can't create output file: ", outfile);
    end
else
    outf = STDOUT;
end
```

In the case of the more complex command line, Julia has a ArgParse package. This has similarities with the Python argparse module, but some important differences too.

In building the load file, we need to handle single quotes (') that are used for text delimiters. The usual convention is to double them up (' '), but some loaders also accept escaped backslashing (\').

For a short data file, we can use `readdlm()` to read all the data with a single call. If the file is very large, it may be more appropriate to read records and process the data on a line-by-line basis.

```
# One liner to double up single quotes
escticks(s) = replace(s,"'","''");
# Read all file into a matrix, first dimension is number of lines
qq = readdlm(infile, '\t');
n = size(qq)[1];
```

Because we are going to create a separate table for categories, we will keep track of these in a hash (dictionary) using sequentially generated IDs and output the hash as a table later.

```
# Going to store all categories in a dictionary
j = 0;
cats = Dict{String,Int64}();

# Main loop to load up the quotes table
for i = 1:n
    cat = qq[i,1];
    if haskey(cats,cat)
        jd = cats[cat];
    else
        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]), "'');");
    write(outf,"$sql\n");
end
```

After creating the quotes table, we create the second one for the categories:

```
# Now dump the categories
for cat = keys(cats)
    jd = cats[cat];
    write(outf,"insert into categories values($jd,'$cat') ;\n");
end
close(outf); # Will have no effect if outf = STDOUT
```

This produces an intermediate SQL load file that can be used with the most standard loaders that can output to `STDOUT` and piped into the loader. Indeed if the input file argument was also optional, it could be part of a Unix command chain.

The merits of using this approach is that it can be used to load any relational database with a SQL interface, and also 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.)

On the downside, we have to 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 DB package we are using.

Native interfaces

By a native interface, I am referring to the paradigm under which a package makes calls to an underlying shared library API.

As an example, we are going to look at the case of SQLite, which is a simple DBMS-less style system. It will be built from source and a wide variety of precompiled binaries, all of which are obtainable from the main website at www.sqlite.org, along with installation instructions, which in some cases, is little more than unzipping the download file.

As a dataset, we are going to work with the "queries" tables for which we created a build and load file in the previous section.

To start SQLite, assuming it is on the execution path, we 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 a number of options that can be listed by typing `sqlite3 -help`. The SQLite interpreter accepts direct SQL commands terminated by a `;`.

In addition to this, instructions to SQLite can be applied using a special set of commands prefixed by a dot (.) , all these can all be listed with the .help command.

Usually these commands can be abbreviated to the short forms so long as it is unique. For example .read can be shortened to .re. Note that 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:

```
/home/malcolm> sqlite3
sqlite> .read build.sql
sqlite> .read load.sql
sqlite> .save quotes.db
sqlite> .ex
```

On OS X or Linux, we can also pipe the build scripts and use the loader script as:

```
cat build.sql | sqlite3 quotes.db
julia etl.jl indata.tsv | sqlite3 quotes.db
```

Now let's turn our attention to the `SQLite.jl` Julia package and run some queries. If we wish to list the quotes and their particular category, we need to join the two tables together based on the category ID.

We can connect to our quotes database by calling the `SQLiteDB()` routine that returns a database handle. The following query returns the number of quotes in the database:

```
using SQLite
db = SQLiteDB("quotes.db")
res = query(db,"select count(*) from quotes");
size(res)
res[1][1]; # => 36
```

This is returned as a `ResultSet` type, which can be thought of as an `{Any}` matrix with each row corresponding to a result and each column to the queried fields.

In the case of a simple count, there is just one row with a single column.

If we return a count of all the query table entries:

```
res = query(db,"select * from quotes");
size(res); # => (36,4)

res[21,4]; # => 35
```

This query will return all the quotes from the database:

```
sql = "select q.quoname, q.author, c.catname from quotes q ";
sql *= "join categories c on q.cid = c.id limit 5";
res = query(db,sql);
```

This query finds all the quotes of Oscar Wilde:

```
sql = "select q.quoname from quotes q ";
sql *= " where q.author = 'Oscar Wilde'";
res = query(db,sql);

nq = size(res)[1]; # Number of records returned
for i = 1:nq
    println(res[i,1]);
end;

I am not at all cynical, I have merely got experience, which 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.
```

There are a number of other functions in the package. One of the most important is the `execute()` statement that is used to run non-query (DML/DDL) statements.

It is possible to create a SQLite statement, which is a combination of the connection handle and the SQL string, as `stmt = SQLiteStmt(db, sql)`. So executing a statement can take the form:

```
execute(stmt) or execute(db, sql)
```

Statements are useful when using placeholders and passing parameterized values.

Passing values can be achieved either by using the `bind()` function of an optional argument of array values.

Database tables can be set up using a single statement using an array of data values, passing the column names and column types:

```
create( db::SQLiteDB, name::String, table::AbstractMatrix,  
       colnames=String[], coltypes=DataType[]; temp::Bool=false)
```

The final optional argument is a flag to indicate that the table is temporary and will be deleted when closed.

The package also provides an `append()` function to add data to an existing table:

```
append(db::SQLiteDB, name::String, table::AbstractMatrix)
```

This takes the values in the table and appends (by repeated inserts) to the SQLite table name. No column checking is done to ensure correct types, so care should be taken as SQLite is "typeless" since it allows items of any type to be stored in columns.

SQLite provides syntax for calling the `REGEXP` function from inside the `WHERE` clauses.

Unfortunately, however, SQLite does not provide a default implementation of the `REGEXP` function, so `SQLite.jl` creates one automatically when you open a database.

For example, selecting the quotes from Oscar Wilde we could use:

```
query(db, "SELECT quoname FROM quotes  
        WHERE author REGEXP 'e(?=Oscar)'")
```

ODBC

The stock way of handling SQL databases in Julia is based on the use of the ODBC layer. As mentioned earlier, this approach has been in operation for many years and imposes certain performance penalties but on the plus side there is a vast wealth of ODBC drivers available.

To use ODBC requires the installation of a specific driver and setting up of a connection string by use of an administration interface. For Windows, this comes as standard and is found by use of the control panel: system and security -- administration tools group.

With Unix and OS X, there are two administration managers available: `unixODBC` and `iODBC`. The former is more standard in approach that sets up the drivers by means of editing configuration files, while the latter is more GUI-based. I tend to use `iODBC` when working on OS X and `unixODBC` on Linux, but both work and it is largely a matter of choice.

unixODBC does have a separate GUI wrapper based on Qt, but I have found it as easy to use the command utility.

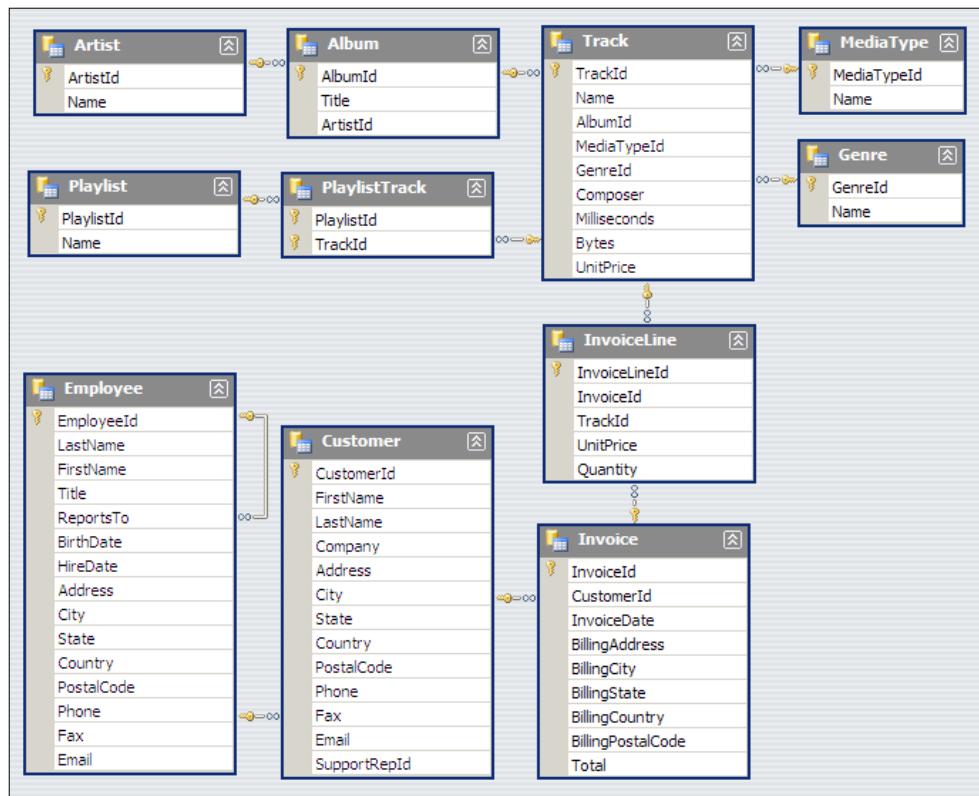
To look at some more sophisticated queries than in the previous section, I am going to introduce the Chinook database (<https://chinookdatabase.codeplex.com>). This is an open source equivalent to the Northwinds dataset that accompanies Microsoft Office.

Chinook comes with load files for SQLite, MySQL, Postgres, Oracle, and many others.

The data model 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 naturally customer and employee information was manually created using fictitious names, addresses, emails, and so on. Information regarding sales was auto-generated randomly.

The schema is shown in the following figure:



In the Chinook schema, there are two *almost* separate subsystems:

- **Artist / Album / Track / Playlist**, and so on
- **Employee / Customer / Invoice**

These are linked together via the **InvoiceLine (TrackId)** table to the **Track** table.

Notice that the schema is not totally normalized as the **InvoiceLine** table can also be joined to **PlaylistTrack** bypassing the **Track** table.

I am going to look at using Chinook with the popular database MySQL; an ODBC driver can be obtained from the development site at <http://dev.mysql.com>.

It is worth noting that since the purchase of MySQL by Oracle, the original team now produces an alternate database MariaDB (v5) that is a direct drop in for MySQL.

The MariaDB driver can also be obtained from <https://downloads.mariadb.org/driver-odbc/>.

Maria v10 is an additional product that promises other features such as columnar access and is well worth a look-see.

The following queries have been run under Ubuntu 12.10, using unixODBC and the MySQL 5.6 driver. For installation and setup of the connection string, the reader is referred to the web page at <http://www.unixodbc.org/odbcinst.html>.

Both unixODBC and iODBC require two configuration files, odbcinst.ini and odbc.ini, the first specifies the drivers and the other the data sources. These can be placed in the /etc directory, which requires admin privileges, or as hidden files (.odbcinst.ini and .odbc.ini).

So to interface with the Chinook database using a MySQL ODBC driver my configuration files look like the following:

```
cat /etc/odbcinst.ini
[MySQL]
Description=MySQL ODBC Driver
Driver=/usr/lib/odbc/libmyodbc5a.so
Setup=/usr/lib/odbc/libmyodbc5S.so

cat odbc.ini
[Chinook]
Description=Chinook Database
Driver=MySQL
```

```
Server=127.0.0.1
Database=Chinook
Port=3306
Socket=/var/run/mysqld/mysqld.sock
```

There is also a Qt-based utility to manage the configuration files, if they are system wide, that is, in /etc, this needs to be run in Linux as root:

```
sudo ODBCManageDataSourcesQ4
```

Getting the configuration files for the drivers and data sources correct can be a little tricky at first, so it is possible to use the `odbcinst` utility to check them and also to connect with `isql`, and run some queries:

```
odbcinst -q -d; # => [MYSQL]
odbcinst -q -s; # => [Chinook]
```

```
isql -v Chinook malcolm mypasswd
```

Assuming you can connect with `isql`, then using it in Julia is straightforward:

```
conn = ODBC.connect("Chinook", usr="malcolm", pwd="mypasswd")
res = ODBC.query("select count(*) from Customers", conn)
println("Number of customers: $(res[:,1][1])")
Number of customers: 59
```

Note that at present the connection handle, if present, comes after the query, although the documentation suggests the reverse. If in doubt, use `methods(query)` to check.

To demonstrate some more complex queries, let's join the `Customers` and `Invoice` tables by the customer ID, and by running a `group by` query select the customers who are the highest spenders, that is, spending more than \$45:

```
sql = "select a.LastName, a.FirstName, ";
sql *= " count(b.InvoiceId) as Invs, sum(b.Total) as Amt";
sql *= " from Customer a";
sql *= " join Invoice b on a.CustomerId = b.CustomerId";
sql *= " group by a.LastName having Amt >= 45.00";
sql *= " order by Amt desc;";

res = ODBC.query(sql);
```

Databases

```
for i in 1:size(res) [1]
    LastName = res[:LastName] [i]
    FirstName = res[:FirstName] [i]
    Invs      = res[:Invs] [i]
    Amt       = res[:Amt] [i]
    @printf "%10s %10s %4d %10.2f\n" LastName FirstName Invs Amt
end
Holy      Helena      7      49.62
Cunningham Richard    7      47.62
Rojas     Luis        7      46.62
O'Reilly Hugh        7      45.62
Kovacs   Ladislav    7      45.62
```

Next, we will select the tracks purchased by Richard Cunningham.

This comprises a set of joins on tables as follows:

```
sql = "select a.LastName, a.FirstName, d.Name as TrackName";
sql *= " from Customer a";
sql *= " join Invoice b on a.CustomerId = b.CustomerId";
sql *= " join InvoiceLine c on b.InvoiceId = c.InvoiceId";
sql *= " join Track d on c.TrackId = d.TrackId";
sql *= " where a.LastName = 'Cunningham' limit 5;";

res = ODBC.query(sql);

for i in 1:size(res) [1]
    LastName = res[:LastName] [i]
    FirstName = res[:FirstName] [i]
    TrackName = res[:TrackName] [i]
    @printf "%15s %15s %15s\n" LastName FirstName TrackName
end

Cunningham Richard      Radio Free Europe
Cunningham Richard      Perfect Circle
Cunningham Richard      Drowning Man
Cunningham Richard      Two Hearts Beat as One
Cunningham Richard      Surrender
```

Other interfacing techniques

ODBC remains largely universal today, with drivers available for most platforms and databases. It is not uncommon to find ODBC drivers for database engines that are meant to be embedded, such as SQLite, and also for non-databases such as CSV files and Excel spreadsheets; also, recently there are ODBC drivers for NoSQL data stores such as MongoDB.

There are a number of alternate approaches in Julia to use ODBC that I will explore now.

DBI

The DBI method of working with databases is very close to my heart as a self-confessed Perl monger, since the mid-1990s. DBI/DBD was introduced by Tim Bunce with Perl 5 to replace a mishmash of techniques that existed at the time.

The Julia `DBI.jl` package (see <https://github.com/JuliaDB/DBI.jl>) offers the same promise, that of an abstract layer of calls to which specific DBD (drivers) need to conform.

At the time of writing this, there are only three examples of Julia DBD drivers.

SQLite

The DBD driver for SQLite is available in the `DBDSQLite.jl` package. (see <https://github.com/JuliaDB/DBDSQLite.jl>).

It is used by importing SQLite and it is worth noticing the `SQLite.jl` native module, earlier with the `SQLiteDB` namespace, in order to distinguish with this driver. Since I have looked at accessing the `sqlite` files using the native package previously, the reader is referred to the online documentation for the alternative use of the DBD driver.

MySQL

A DBD driver for MySQL was provided at the same time as the DBI package (see <https://github.com/johnmyleswhite/MySQL.jl>). It has (currently) a couple of severe restrictions that makes it impractical at present:

- It looks for shared libraries of the `dylib` type, which only makes it usable on the OS X operating system
- It does not implement the `select` statements and record sets, so can't be used to query the database

It is worth checking the current status of the driver, since when fully functional it is a more convenient approach to work with MySQL than via an ODBC middle layer. I will illustrate an alternate approach in the next section.

Nevertheless, it is possible to do some database ETL scripts to create and manipulate tables, grant access privileges and to insert, amend and delete records. If we consider the `etl.jl` file and script developed earlier, it was necessary to run the SQL output file through the `mysql` utility in order to insert the records; this could be written in Julia as:

```
# Skipping the argument processing
# Assuming that the input file is quotes.tsv
# and using the test database

Using DBI
Using MySQL

escticks(s) = replace(s,"'", "''");
qq = readdlm("quotes.tsv",'\'t');
n = size(qq)[1];
conn = connect(MySQL5,"localhost","malcolm","mypasswd","test");

# We can create the categories table and quotes tables:
sql = "create table categories ";
sql *= "(id integer primary key, ";
sql *= "catname varchar(40) not null";
execute(prepare(conn,sql));
if (errcode(conn) > 0)
    error("Can't build categories table")
end;

# Note: Similar code for building the quotes table ...

j = 0;
cats = Dict{String,Int64}();

# Main loop to load up the quotes table
for i = 1:n
```

```
cat = qq[i,1];
if haskey(cats,cat)
    jd = cats[cat];
else
    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]), ""');");
stmt = prepare(conn,sql); execute(stmt);
if errcode(conn) > 0 error(errstring(conn)); end;
end

# Now dump the categories
for cat = keys(cats)
    jd = cats[cat];
    sql = "insert into categories values($jd,'$cat')";
    stmt = prepare(conn,sql); execute(stmt);
    if errcode(conn) > 0 error(errstring(conn)); end;
end;
disconnect(conn);
```

PostgreSQL

PostgreSQL.jl (see <https://github.com/iamed2/PostgreSQL.jl>) is a Julia package that obeys the DBI.jl protocol and works on multiple platforms using libpq (the C PostgreSQL API) which is included as part of the Postgres installation.

I have been an advocate of PostgreSQL for a long time, as it contained transactions, stored procedures, and triggers, long before these were added to MySQL. With the dichotomy between commercial and community editions of MySQL, it is becoming more popular of late.

We can load the `Chinook_PostgreSQL.sql` Chinook data in to a running system using the `createdb` and `psql` utility from the command line as follows:

```
createdb Chinook
psql -d Chinook -a -f Chinook_PostgreSQL.sql

using DBI, PostgreSQL
connect(Postgres) do conn
    stmt = prepare(conn, "SELECT * FROM \"MediaType\"");
    res = execute(stmt);
    for row in res
        @printf "ID: %d, Media: %s\n" row[1] row[2]
    end
    finish(stmt);
end

ID: 1, Media: MPEG audio file
ID: 2, Media: Protected AAC audio file
ID: 3, Media: Protected MPEG-4 video file
ID: 4, Media: Purchased AAC audio file
ID: 5, Media: AAC audio file
```

A couple of points to note here:

- The Chinook dataset follows the Northwind-style, that is, it uses capitalized and CamelCased identifiers. These are all converted (by default) to lowercase, so it is necessary to encase them in quotes and in Julia to escape the quotes using a backslash.
- Executing the `SELECT` query returns a `PostgresResultHandle`, which is a pointer to an in-memory data structure with a set of results, each of which is an `Any { }` array.

PyCall

We have seen previously that Python can be used for plotting via the `PyPlot` package that interfaces with `matplotlib`. In fact, 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.

Any database that can be manipulated by Python is also available to Julia. In particular, since the DBD driver for MySQL is not fully DBT compliant, let's look at this approach to run some queries.

Our current MySQL setup already has the Chinook dataset loaded, we will execute a query to list the `Genre` table.

In Python, we will first need to download the `MySQL Connector` module.

For Anaconda, this needs to use the source (independent) distribution rather than a binary package, and the installation is performed using the `setup.py` file.

The query (in Python) to list the `Genre` table would be:

```
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 to the `mysql.connector` module and the form of the coding is remarkably similar:

```
using PyCall
@pyimport mysql.connector as mc

cnx = mc.connect (user="malcolm", password="mypasswd");
csr = cnx[:cursor] ()
```

```
query = "SELECT * FROM Chinook.Genre"
csr[:execute](query)

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
...
...
csr[:close]()
cnx[:close()]
```

Note that the form of the call is a little different from the corresponding Python method, 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](qry)`.

Also, be aware that although Python arrays are zero-based, this is translated to one-based by PyCall, so the first value is referenced as `vals[1]`.

JDBC

JDBC is another middle layer that functions similar to ODBC, but most operating systems have Java installed so JDBC offers a different option when connecting to databases. Also, databases may have JDBC connectors but not ODBC ones.

In Julia, using JDBC is achieved via the `JavaCall` module, in a fashion similar to calling methods in any Java classes.

For example, consider the Derby database, which is Java-based and distributed by the Apache projects. The prerequisite JAR files need to be downloaded and available via the Classpath Java.

```
CP = $DERBY_HOME/lib/derby.jar:$DERBY_HOME/lib/derbytools.jar
```

Using Derby tools IJ interface and previous SQL build/load files, it is possible to populate the database and then run simple query to count the number of quotes in the database:

```
java -cp $CP org.apache.derby.tools.ij
ij version 10.3
ij> connect 'jdbc:derby:Books; create=true';
ij> help;
ij> run 'build.sql';
ij> run 'qloader.sql';
ij> select count(*) as K from quotes;
K
--
36
```

Now let's look at how to do the same in Julia via JavaCall and JDBC:

```
Using JavaCall
jsd = @jimport java.sql.DriverManager;
dbURL = "jdbc:derby:Books1";
conn = nothing;
try
    db = jcall(jsd,"getConnection",JavaObject,(JString,),dbURL);
    jsp = @jimport java.sql.PreparedStatement;
    jsr = @jimport java.sql.ResultSet;
    sql = "select count(*) as K from quotes";
    stmt = jcall(db,"prepareStatement",isp,(JString,),sql);
    res = jcall(stmt,"executeQuery",jsr,());
    k = jcall(res,"getString",JString,(JString,),"K");
catch e
    println(e);
finally
    if conn != nothing
```

```
jcall(conn, "close", Void, ());
end;
end;
println("\nNumber of quotes in database: $k);

Number of quotes in database: 36
```

The points to note are the use of the `@jimport` macro used to import the various classes and the `jcall()` function then called to run methods in the classes.

NoSQL datastores

When compared to relational databases, NoSQL databases are more scalable and provide superior performance, and their data model addresses several issues that the relational model is not designed to address when dealing with large volumes of structured, semi-structured, and unstructured data.

To achieve this, requires a wide variety of different database designs developed in response to a rise in the volume of data stored, the frequency in which this data is accessed, and performance and processing needs.

Types of NoSQL databases can be (roughly) classified under the following four headings:

- **Key-value stores:** These are among the simplest NoSQL databases. Every single item in the database is stored as an attribute name (or "key"), together with its value.
- **Document databases:** These pair each key with a complex data structure known as a document, which themselves may contain many different key-value pairs or key-array pairs, or even nested documents.
- **Wide-column stores:** These are optimized for queries over large datasets, and store columns of data together instead of rows.
- **Graph stores:** These are used to store information about networks using a data model consisting of nodes, with associated parameters, and relations between the nodes.

NoSQL database technologies are especially prevalent in handling big data systems that have a large volume of data and high throughput, and of necessity are distributed over multiple physical servers.

The Julia community has responded with a number of packages to meet several types of NoSQL systems and undoubtedly more will exist when this book comes to print. A useful online reference is at <https://github.com/svaksha/Julia.jl/blob/master/Database.md> and we will explore a few examples in the rest of this chapter.

Key-value systems

Key-values (KV) systems are the earliest of databases, preceding relational ones. The first were known as **ISAM (index sequential access method)** -- and continued in various forms when used in LDAP and **Active Directory (AD)** services, despite the onset of the SQL database era.

Two Julia implementations are for the Redis and memcached datastores, and I'll discuss the former in this section. Redis is very fast as it operates totally in memory and is sometimes referred to as a data cache, although it is configured to write to disk asynchronously after a series of Redis write operations.

It is a very simple database to install. Downloads are available from redis.io, both binaries and source code. The source code builds quite easily and runs in the user space, that is, does not need administrator privileges.

Also, it is possible to get a cloud database from <https://redislabs.com/>, which is free for the first 25 MB. Redis Labs runs on Amazon Web Services and offers both Redis and memcached.

Redis uses a simple set of commands to create and retrieve records.

Its main feature is that in addition to simple (basic) string keys, it is possible to create lists, hashes, sets, and sorted sets. Also, string keys may be bit-mapped and has a new data structure the **hyperloglog**.

So Redis has great flexibility in how data can be stored in it.

In Julia, there are two package implementations:

- `Redis.jl`: This is a native mode package that sends messages to the Redis server, this is available via the standard Julia repository at [pkg.julialang.org](https://pkgs.julialang.org)
- `Hiredis.jl`: This is a wrapper package around the `hiredis` C API, a shared library dependency, clone it from <https://github.com/markmo/HiRedis.jl>

Here, I will look at the native package.

The connection is made using `RedisConnection`, and simple keys can be stored and retrieved by the `set()` and `get()` functions:

```
using Redis;
conn = RedisConnection()
set (conn, "Hello", "World");
keys(conn, "*")
Hello
println(get(conn, "Hello"));
World
disconnect(conn);
```

As an example, we will grab the stock market prices using the `Quandl` package, store it in Redis, retrieve them and display 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 four (or less) character code and a set of values such as `Open`, `Close`, `High`, `Low`, `Volume`, plus the `Date`.

For Apple stocks, a choice of key for a closing price may be `APPL~Date~Open` where the `~` 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.

Better would be 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.

The following gets the Apple (`APPL`) and Microsoft (`MSFT`) stocks, stores them in Redis, retrieves them, and displays them using `Winston`:

```
using Quandl, DataFrames, Redis, Winston

qapi_key = "aaaaaaaaaaaaaaaaaaaa" # Your Quandl key
set_auth_token(qapi_key)

qdf1 = quandl("WIKI/AAPL", format="DataFrame", order="asc");
aapl = convert(Array, qdf1[:Close]);
```

```
scf1 = 1.0/aapl[1];  aapl = scf1 .* aapl;

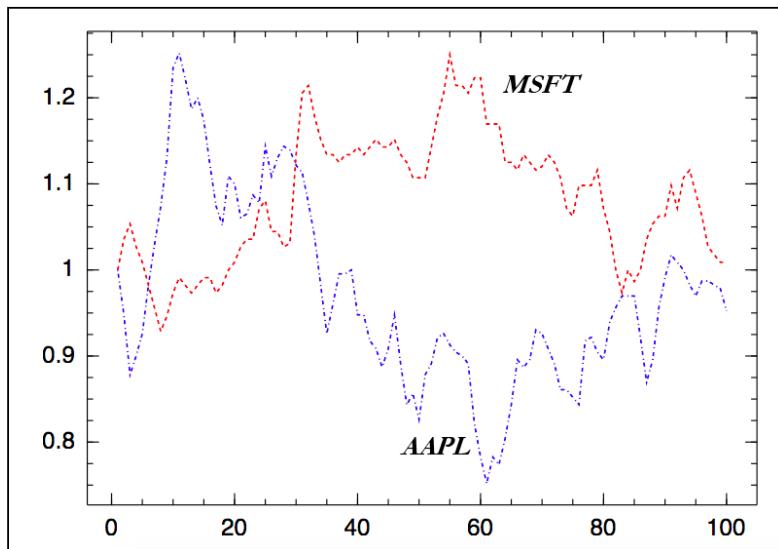
qdf2 = quandl("WIKI/MSFT", format="DataFrame", order="asc");
msft = convert(Array, qdf2[:Close]);
scf2 = 1.0/msft[1];  msft = scf2 .* msft;

n = [1:length(aapl)];
conn = RedisConnection()
for i = 1:n
    rpush(conn,'APPL~Close',aapl[i])
    rpush(conn,'MSFT~Close',msft[i])
end

t = [1:n];  # Just plot the data not the dates

aapl-data = float32(lrange(conn,"AAPL~Close",0,-1);
msft-data = float32(lrange(conn,"MSFT~Close",0,-1));
plot(t, aapl-data,"b--", t, msft-data, "r.");
```

The resulting plots are shown in the following figure:



Note the following points:

- The data is retrieved from Quandl as a dataframe
- It is necessary to have an API key to Quandl, this is free on registration
- The key is Stock~Price and is pushed on to the list at the end using RPUSH
- It is possible to return the list in a single call to LRANGE
- Lists in Redis are zero-based and the indexing can be negative, meaning it counts backwards from the end, so the -1 corresponds to the last value

Document datastores

Document-oriented database are designed for storing, retrieving, and managing semi-structured data.

MongoDB is one of the most popular document databases that provides high performance, high availability, and automatic scaling. It is supported by a Julia package that is a wrapper around the C API, and 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 similar to JSON objects. The values of 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 is in data that must exist and be writeable.

There is a Mongo shell client program (`mongo`) that can be used to add and modify records and as a query engine:

```
MongoDB shell version: 2.6.7
> use test
switched to db test
> db.getCollectionNames()
[ "quotes", "system.indexes" ]
> qc = db.quotes
test.quotes
>qc.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"})
```

```
>qc.insert({category:"Computing",quote:"Old programmers never die. They  
just branch to a new address"})  
.....  
.....  
> qc.count()  
35
```

Note that because Mongo is schemaless, in cases where the author is unknown, the field is not specified and also 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:

```
> qc.find({author:"Oscar Wilde"})
```

```
{"_id" : ObjectId("54db9741d22663335937885d"), "category" : "Books &  
Plays", "author" : "Oscar Wilde", "quote" : "The only way to get rid of a  
temptation is to yield to it." }
```

Loading Mongo up with all 35 quotes (which are in the file quotes.js) we can now use Julia to run some searches:

```
using Mongo, LibBSON  
client = MongoClient(); # Defaults to localhost on port 27017  
  
mc = MongoCollection(client,"test","quotes");  
bob = BSONObject({"author" => "Albert Einstein"});
```

The client, database, and collection are combined together with the MongoCollection() routine and the query string as a BSON object:

```
println("Quotes by Albert Einstein numbers ", count(mc, bob));  
Quotes by Albert Einstein numbers 2
```

To list Einstein's quotes, we must create a cursor to the records and traverse the record set sequentially.

It is a one-way cursor, so the query needs to be rerun to get the records a second time:

```
cur = find(mc,bob); # => MongoCursor(Ptr @0x00007fadb1806000)  
  
for obj in cur  
    println(obj["category"],": ",obj["quote"])
```

```
end  
Politics: I know not with what weapons World War IIII will be fought, but  
World War IV will be fought with sticks and stones.  
Science: The true sign of intelligence is not knowledge but imagination
```

The Julia package also contains routines to add, modify, and delete documents.

Consider the following three BSON Objects that describe a new quote by Einstein in a "Test" category:

```
bin = BSONObject({ "author" => "Albert Einstein", "category" =>  
    "Test", "quote" => "I don't believe that God plays dice."});  
bup = BSONObject({ "author" => "Albert Einstein", "category" =>  
    "Test", "quote" => "A person who never made a mistake, never  
    made anything."});  
brm = BSONObject({ "author" => "Albert Einstein", "category" =>  
    "Test" });
```

Then we can insert a new insert document, modify, and delete it as follows:

```
insert(mc,bin); # => BSONOID(54e1201c52f2alf4f8289b61)  
count(mc,bob); # => 3  
update(mc,bin,bup); # => true  
count(mc,bob); # => 3  
delete(mc,brm); # => true  
count(mc,bob); # => 2
```

By running the `find(mc,bob)` query, the reader can check the addition and modification of the new quote.

RESTful interfacing

The term REST refers to, which is a software architecture style to create scalable web services. REST 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 the more common for complex requests as it contains meta information to identify the various fields.

Historically this was returned as XML, which is still common in SOAP web services, but more popular recently is JSON, 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 by use of a "parser".

To access them we need a method to mimic the action of the web browser programmatically and to capture the returned response.

We saw earlier that this can be done using a task, such as `wget` or `curl`, with the appropriate command line:

```
rts = run(`wget -q -O- http://amisllp.com/now.php`)
rts = run(`curl -X GET http://amisllp.com/now.php`)
```

These will run the REST web page now, and PHP will return the current (UK) date and time. Either of the tasks will produce the same result:

```
println(rts);
2015-02-18 12:11:56
```

Alternatively in Julia, we can use the `HTTPClient.jl` package that utilizes `LibCURL.jl` to do the heavy lifting or `Requests.jl` that uses to joyent HTTP parser via `HttpParser.jl`.

The following examples use `HTTPClient`, but we will look at the `Request` package in the next chapter.

JSON

In *Chapter 5, Working with Data*, we saw how to parse XML documents, so we now turn our attention to how to handle JSON data.

JavaScript Object Notation (JSON) is a set of markup rules that uses a few special characters, such as `{},[],",:,:,\`, to present data in the `name:value` form. There is no termination tagging (as in XML) and it can handle arrays and other JSON subobjects. For those unfamiliar with JSON, the full specification is given on the official website <http://json.org>.

The following is a simple webservice using `httpbin.org` to return my IP addresses:

```
using HTTPClient.HTTPPC, JSON
url = "http://httpbin.org/ip"
uu = HTTPPC.get(url);
names(uu)
1x4 Array{Symbol,2}:
:body :headers :http_code :total_time
```

The `HTTPClient` returns an array that contains both the HTTP header and the HTTP body, as well as the return code (which is usually `200`, denoting `OK`), and the time taken for the request.

Assuming the return status was `OK`, the body is of the most interest. It has a data object comprising a byte (`UInt8`) array, and this needs to be converted to a string and then this can be parsed into JSON.

Parsing creates a `DICT` object that, in our examples, contains the requesting IP address:

```
ubody = bytestring(uu.body.data);
ss = JSON.parse(ubody);
myip = ss["origin"]
@printf "My IP address is %s\n" myip
My IP address is 10.0.141.134
```

For a more detailed example that returns a complex JSON structure, we can query the Google Books API to return data on a book with an ISBN number of `1408855895`:

```
using HTTPClient.HTTPPC
using JSON
api = "https://www.googleapis.com/books/v1/volumes"
url = api * "?q=isbn:1408855895"
uu = HTTPPC.get(url)
json = JSON.parse(bytestring(uu.body.data));
```

From the returned JSON data structure, we can extract extensive information on the book and I'll leave it to the reader to explore all the data in detail.

Here is the title of the book, the author (there is only one), the publisher, its publication date, and page count:

```
volumeInfo = json["items"][1]["volumeInfo"]
title     = volumeInfo["title"]
author    = volumeInfo["authors"][1]
publisher = volumeInfo["publishedBy"]
pubdate   = volumeInfo["publishedDate"]
ppcount   = volumeInfo["pageCount"]

printf "%s by %s\nPublished by %s (%s), Pages: $d"
        title author pubdate ppcount
```

```
Harry Potter and the Philosopher's Stone by J. K. Rowling
Published by Bloomsbury Childrens (2014-09-01), Pages: 354
```

Web-based databases

Examples of database systems that provide a REST API are:

- **Riak:** This is an alternative key-value datastore, which operates more as a conventional on-disk database than the in-memory Redis system (<http://basho.com>).
- **CouchDB:** This is an open source system, now part of the Apache project that uses JSON-style documents and a set of key-value pairs for reference (<http://couchdb.apache.org/>).
- **Neo4j:** This is a prominent example of a graph database that stores nodes, parameters, and interconnect relationships between nodes in a sparse format rather than as conventional record structures (<http://neo4j.com>).

In this section, we will look briefly at CouchDB and in the final section at Neo4j.

CouchDB is not to be confused with the commercial product Couchbase, which is similar in operation but is provided as both an Enterprise and Community system.

Apache provides a number of binary downloads for CouchDB for Linux, OS X, and Windows. As it is written in Erlang, building from source is a little more involved but presents no real difficulties.

Starting up the server daemon on the localhost (127.0.0.1) runs the service on the 5984 default port. Querying this confirms that it is up and running:

```
curl http://localhost:5984
{
  "couchdb": "Welcome",
  "uuid": "d9e1f98c96fc513a694936fb60dc96f8",
  "version": "1.6.1",
  "vendor": { "version": "1.6.11", "name": "Homebrew" }
}
```

Next, we need to create a database comprising the quotes dataset using either the PUT or POST command and show it exists:

```
curl -X PUT http://localhost:5984/quotes
curl http://localhost:5984/_all_dbs
[ "_replicator", "_users", "quotes" ]
```

To add records to quotes, we have to specify the content type as a JSON string:

```
curl -H 'Content-Type: application/json' \
-X POST http://127.0.0.1:5984/quotes \
-d '{category: "Computing", author: "Scott's Law",
"quote": "Adding manpower to a late software project makes it later"})'
```

I'll look at querying CouchDB in Julia after adding all the quotes we used previously when discussing SQLite and Mongo. As with Mongo, anonymous quotes are specified by omitting an author field.

To do this in curl is possible but not very flexible, so CouchDB contains a utility IDE called Futon that can run in the browser using the URL http://127.0.0.1:5984/_utils:

```
using HTTPClient.HTTPPC, JSON
cdb = "http://localhost:5984";
dbs = bytestring(get("$cdb/_all_dbs").body.data);
JSON.parse(dbs); # => [ "_replicator", "_users", "quotes" ]
```

As with curl, in order to post data to CouchDB, we will need to specify the MIME type and we do this in `HTTPClient` by setting some request options:

```
qc = cdb * "/quotes";
ropts = RequestOptions(content_type="application/json");
```

So to add a quote to the preceding code, we would use the following:

```
json = "{category:\"Computing\",author:\"Scott's Law\",
\"quote\":\"Adding manpower to a late software project makes it
later\"}";
HTTPPC.post(qc, json, ropts);
```

To return a list of all documents in the database, we run a command in the form:

```
docs = bytestring(get(qc*"/_all_docs"))
json = JSON.parse(docs.body.data);
```

This returns the number of records, together with an array of records keys:

```
json["total_rows"]; # => 35
json["rows"][24];   # Pick out an specific record
Dict{String,Any} with 3 entries:
  "key"    => "d075d58c26d256c49b965a02cc00b77a"
  "id"     => "d075d58c26d256c49b965a02cc00b77a"
  "value"   => ["rev"=>"1-524b4b2c577e2eaff31f4b665cd48055"]
key = json["rows"][24]["key"];
rev = json["rows"][24]["value"]["rev"];
```

To display the full record, we simply do an HTTP GET method against this key:

```
JSON.parse(bytestring(get("$qc/$key").body.data))
Dict{String,Any} with 5 entries:
  "quote"      =>
    "To love oneself is the beginning of a lifelong romance"
  "_rev"        => "1-524b4b2c577e2eaff31f4b665cd48055"
  "author"     => "Oscar Wilde"
  "_id"         => "d075d58c26d256c49b965a02cc00b77a"
  "category"   => "Books & Plays"
```

The revision field is needed if we want to delete a record. HTTPPC.
delete("\$qc/\$key?rev=\$rev");

Updating a record is just the process of posting the new data in JSON format to an existing record ID and must use the revision key and also the request options, as now we are executing a POST operation:

```
json = "{category:\"Books & Plays\",author:\"Oscar Wilde\",
\"quote\":\"I can resist everything but temptation\"}";
HTTPPC.post("$qc/$key", json, ropts);
```

Graphic systems

A graph database is based on graph theory and uses nodes, properties, and relationships to provide index-free adjacency.

One of the most popular graph database is Neo4j. It is open source, implemented in Java and Scala, began in 2003, and has been publicly available since 2007. The source code and issue tracking are available on GitHub.

Neo4j comes in both Community and Enterprise edition and is downloaded from the main website <http://neo4j.com/download>, and being a database on the JVM runs on all operating systems. By default, it is bundled with an interactive, web-based database interface bound to <http://localhost:7474>.

Any language on the JVM can interface with Neo4j directly. In addition, there is a REST API and a group of languages, including Python, Ruby, and NodeJS, that have modules that simplify the use of the API. At present, Julia is not one of these and so it is necessary to interact directly using HTTP commands.

Neo4j comes with some example data that corresponds to some movie information in the form of persons, movie titles, and relationships that connect the two as either actors or directors and properties such as the name of the character portrayed. This can be loaded through the web interface using the Neo4j **Cypher** language via the browser.

One advantage of using the browser is that it provides the output visually in graphic form that can be drilled down to display the underlying data.

However, there is also a `neo4j-shell` utility program, which is useful in getting familiar with Cypher queries.

So after loading the database with the sample data, we can see the total number of nodes and relationships as follows:

```
neo4j-sh (?)$ start r=node(*) return count(r);
+-----+
| count(r) |
+-----+
| 171      |
+-----+
neo4j-sh (?)$ start r=rel(*) return count(r);
+-----+
| count(r) |
+-----+
```

```
| 253      |
+-----+
```

Also, we can list some names of people in the database (limiting it to 3):

```
neo4j-sh (?) $MATCH (people:Person) RETURN people.name LIMIT 3;
+-----+
| people.name      |
+-----+
| "Keanu Reeves"  |
| "Carrie-Anne Moss" |
| "Laurence Fishburne" |
+-----+
```

We can also get a list of co-actors who worked with Tom Hanks (again limiting to 3):

```
neo4j-sh (?) $ MATCH (tom:Person {name:"Tom Hanks"}) -[:ACTED_IN]->(m)-[:ACTED_IN]->(coActors)
RETURN coActors.name LIMIT 5;
+-----+
| coActors.name    |
+-----+
| "Meg Ryan"      |
| "Greg Kinnear"   |
| "Parker Posey"   |
+-----+
```

So let's now look at how to do some queries in Julia using the REST API.

Again we use `curl` and must `POST` the query from JSON to Neo4j, and specify that the posted data MIME type is `JSON`.

The translation from the neo4j-shell to Julia is pretty straightforward and the first two queries are:

```
using HTTPClient.HTTPPC, JSON;
cypher = "http://localhost:7474/db/data/cypher";
ropts = RequestOptions(content_type="application/json");
match = "START r=node(*) RETURN count(r)"
rts = HTTPPC.post(cypher,"{\"query\":\"$match\"}",ropts);
json = JSON.parse(string(rts));
json.body.columns; # => [ "count(r)" ];
```

```
json.body.data;      # => [[ 171 ]];

# And using MATCH = "START r=rel(*) RETUrN count(r)"
json.body.columns; # => [ "count(r)" ];
json.body.data;    # => [[ 253 ]];
```

For more complex matches we have:

```
match = "MATCH (people:Person) RETURN people.name LIMIT 3";
json.body.columns; # => [ "people.name" ]
json.body.data
[["Keanu-Reeves"], ["Carrie-Ann Moss"], ["Laurence Fishburne"]]

match = "MATCH ((tom:Person {name:'Tom Hanks'})-
[:ACTED_IN] ->(m)<- [:ACTED_IN] -(coActors)
RETURN coActors.name  LIMIT 3";

json.body.columns; # => [ "coActors.name" ]
json.body.data'
[["Meg Ryan"], ["Greg Kinnear"], ["Parker Posey"]]
```

Summary

This chapter has looked at the means by which Julia interacts with data held in databases and data stores. Until recently, the great 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 the heading NoSQL and are categorized as key-value, document, and graphic databases. With such a large field to cover, we identified some definitive examples in each category.

Julia's approaches are largely specific to each individual case, and the appropriate packages and methods for loading, maintaining, and querying the different types of databases have been presented.

In the next chapter, we will discuss working with various networked systems. We will look at developing Internet servers, working with web sockets, and messaging via e-mail, SMS, and Twitter. Finally, we will explore the use of the cloud services such as those provided by Amazon and Google.

10

Working with Julia

If you have reached this point in the book, probably you will have worked through some of the examples, looked at the Julia modules that are useful to you, and written some of your own code.

Possibly you will want to package your code in a module and share it with the Julia community. Alternatively, you may want to work within a community group that develops and maintains existing packages and adds new ones.

When sharing your code and working with others, it is important to bear in mind that the source will be looked at and possibly mentored, so certain questions of style, efficiency, and testing come more into play.

This final chapter addresses some of these topics and will hopefully start you a long and fruitful romance with Julia.

Under the hood

This section looks at some topics outside conventional Julia coding, so you may want to omit it for the first reading. However, you may also find it interesting!

Working with Julia is about enabling the analyst/programmer to develop enterprise grade code without the need to develop in a second (compiled) language due to a necessity to improve the performance.

It is true Julia that is fast, but this is not what makes it special. However, it is quite nice that it is fast and I'll first look at some of the reasons why.

Stephen Hawking in *A Brief History of Time* quotes a well-known anecdote concerning a lecture Bertrand Russell was giving in the 1920s on cosmology, when the known universe was considered to be comprised of just our Milky Way galaxy.

Russell described how the Earth orbits around the sun and how the sun, in turn, orbits around the center of a vast collection of stars called the galaxy.

At the end of the lecture, a little old lady at the back of the room got up and said, "What you have told us is rubbish. The world is really a flat disc supported on the back of a giant tortoise."

Russell replied, "But what is the tortoise standing on?"

"You're very clever, young man," said the old lady. "It's turtles all the way down!"

Julia is turtles all the way down, almost.

First, we will consider what the bottom turtle is standing on.

Femtolisp

Julia was preceded by an earlier project by one of the three main Julia developers, Jeff Bezanson, who developed a lightweight Lisp parser, which would be quick, require only a small amount of memory, and remain flexible. At that time this space was occupied by **PicoLisp**. The name, presumably, was chosen since **femto** is the next order of 1000 down from **pico**.

The Julia system encompasses **femtolisp** at its heart. It is responsible for parsing the code before passing it down to the LLVM compiler to produce machine code.

It is not always known that Julia can be started in Lisp mode using the `-lisp` switch:

```
julia --lisp  
;  
; _  
; |_ _ _ |_ _ | . _ _  
; | ( - | | | _ ( _ ) | _ | _ ) | _ )  
;-----|-----  
>
```

While this book is about Julia and not Lisp, the following example of a recursive definition for the factorial function should be clear enough:

```
(define (fac n)
  (if (<= n 0) 1
      (* n (fac (- n 1)))))
> (fac 10); # => 3628800
> (fac 20); # => #int64(2432902008176640000)
> (fac 30); # => #uint64(9682165104862298112)
> (fac 40); # => -70609262346240000
```

Clearly, some of the results for higher factorials look distinctly odd—why this is so we will discuss later.

Here is a second (recursive) example of the Fibonacci sequence:

```
(define (fib n)
  (if (< n 2) n
      (+ (fib (- n 1))
          (fib (- n 2)))))

> (fib 10); # => 55
> (fib 20); # => 6765
> (fib 30); # => 832040
> (fib 40); # => 102334155    ( very slow )
```

Here recursion is not a great choice, since each call generates two additional calls and the process proceeds exponentially.

Tail recursion or just plain loopy coding, which essentially is the same thing, is a better option, and I've included the alternates in the code accompanying this book.

The Julia API

Building Julia from source generates a set of shared libraries, whereas a binary package comes with these libraries prebuilt. Most are coded in C but some of the dependencies, such as `lapack`, contain Fortran code, which is why a `gfortran` compiler is needed to be installed from source.

Central to the system is `libjulia`, which comprises the `femtolisp` compiler and a `libsupport` support library. The latter may be considered as the CORE and is written (mainly) in C, in contrast to the BASE written in Julia.

The CORE corresponds to the glue between Julia runtime and the underlying operating system, and the routines by which Julia communicates with the CORE is the Julia API.

Looking at source listing in BASE will show calls to functions with the `ji_` prefix. The first argument does not specify a shared library, which is equivalent to using `libjulia`.

For example, in `utils.jl` the `time_ns()` function makes a call to `jl_hrtime` as:

```
time_ns() = ccall(:jl_hrtime, UInt64, ())
```

This returns the system time as an unsigned 64-bit integer in nanoseconds.

This is used by `@elapsed` and other macros to give more accurate execution timings.

To review the API, pull a copy of the `Julia.jl` distribution from GitHub and look at the C listings in the `src` subdirectory.

Real applications will not just need to execute expressions, but also return their values to the host program. `jl_eval_string` returns a `jl_value_t*`, which is a pointer to a heap-allocated Julia object.

Storing simple data types such as `Float64` in this way is called boxing, and extracting the stored primitive data is called unboxing.

The following C program uses the API routines to output the value π^2

First, we need to point to the location of the Julia core library by calling the `jl_init()` routine and also execute the `JL_SET_STACK_BASE` C-macro, which is defined in the `julia.h` file:

```
#include <stdio.h>
#include <math.h>
#include <julia.h>
#define LIBEXEC "/usr/lib64/julia"

int main() {
    jl_init(LIBEXEC);
    JL_SET_STACK_BASE;
    jl_function_t *func = jl_get_function(jl_base_module, "^");
    jl_value_t *arg1 = jl_box_float64(M_PI);
    jl_value_t *arg2 = jl_box_float64(2.0);
    jl_value_t *ret = jl_call2(func, arg1, arg2);
    if (jl_is_float64(ret)) {
        double pi2 = jl_unbox_float64(ret);
        printf("PI (squared) is %f\n", pi2);
    }
    return 0;
}
```

The power function is implemented in the CORE and retrieved by `jl_get_function_t()` that returns a pointer. We need to box-up both the arguments, π and 2, in order to pass them to the routine using `jl_box_float64()`.

The power function (`^`) has 2 arguments, so we call `jl_call2()` to run it. It returns a pointer to a `float64`, which we can confirm via `jl_is_float64()`.

If true than we can unbox the function value and print it in C.

When building the program (`pisqd.c`), we need to pass the location of the Julia header file and the Julia library to `gcc`:

```
export JINC=/usr/include/julia
export JLIB=/usr/lib64/julia
gcc -o pisqd -I$JINC -L$JLIB -lJulia pisqd.c
```

Code generation

Julia generates code right down to the machine assembly level, which is the reason that its performance is close to that of C and Fortran code. Other languages such as JavaScript and Go, both using the Google V8 engine and LuaJIT, use similar approaches and provide equivalent performance.

Python has numerous ways of speeding up performance. This also includes Numba, which uses an LLVM approach but (at present) comes with some limitations.

It is difficult to envisage R and MATLAB/Octave embracing LLVM without major rewrites.

In Julia, code is generated (conceptually) in various stages and the generated source can be inspected via a set of `code_*` routines.

These can get complex quite quickly, so I'm going to illustrate the process with a simple function that increments its argument:

```
inc(x) = x + 1
```

The first two stages (lowered, typed) create a functional representation in the form of a lambda expression, and then create a specific instance for a specific set of argument(s):

```
julia> code_lowered(inc,(Int64,))
1-element Array{Any,1}:
 :($Expr(:lambda, {:_x}, {},{{:x,:Any,0}},{}, :(begin
           return x + 1
       end)))
julia> code_typed(inc,(Int64,))
1-element Array{Any,1}:
 :($Expr(:lambda, {:_x}, {},{{:x,Int64,0}},{}, :(begin
           return (top(box))(Int64,(top(add_int))(x:Int64,1))::Int64
       end)::Int64)))
```

The third stage is to create the LLVM code, which would run on the virtual machine. This could be thought of as equivalent to Java/JVM or .NET/CLR code:/

```
julia> code_llvm(inc,(Int64,))
define i64 @julia_inc_20105(i64) {
top:
    %1 = add i64 %0, 1, !dbg !1248
    ret i64 %1, !dbg !1248
}
```

This code is pretty obvious: `%0` and `%1` are pseudo-registers.

`1` is added to `%0` and the result written to `%1`, which is returned as the function value.

Unlike Java or .NET that require runtime environments to execute their code, this is passed to the LLVM compiler for the host machine and then turned into native code.

For Windows, recent Macs, and Linux, this will be in Intel x86 assembly language:

```
julia> code_native(inc,(Int64,))
.section __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 1
push RBP
mov RBP, RSP
Source line: 1
lea RAX, QWORD PTR [RDI + 1]
pop RBP
ret
```

`push/mov` at the beginning and `pop` at the end of the code are for housekeeping that will occur for any function call to preserve the stack base pointer (`RBP`).

The actual statement that does the addition and loads the return register (`RAX`) is:

```
lea RAX, QWORD PTR [RDI + 1]
```

If we call `inc()` with a real argument rather than an integer, we can look at the code by specifying a `Float64`; skipping the first two stages, we get:

```
julia> code_llvm(inc,(Float64,))
define double @julia_inc_20106(double) {
top:
    %1 = fadd double %0, 1.000000e+00, !dbg !1251
    ret double %1, !dbg !1251
}
julia> code_native(inc,(Float64,))
```

```
.section __TEXT,__text,regular,pure_instructions
Filename: none
Source line: 1
    push  RBP
    mov   RBP, RSP
    movabs RAX, 4540233472
Source line: 1
    addsd XMM0, QWORD PTR [RAX]
    pop   RBP
    ret
```

Here the add instruction in the LLVM is replaced with a fadd and now the constant is 1.0

The x86 code reflects this. The curious 454 0233472 value is actually 1.0E+00 when written out as bits, and the value is now returned in the XMM0 first extended register rather than in RAX.

Whereas this code is relatively short, you may wish to see what code is generated when the argument type is Rational and work through the LLVM and x86 assembly listings.

For a more complex example, we will look at Julia version of the Fibonacci function seen earlier in femtolisp.

A simple Julia (recursive) definition of the Fibonacci function would be as follows:

```
function fib(n::Integer)
    @assert n > 0
    return (n < 3 ? 1 : fib(n-1) + fib(n-2))
end
```

Here, I am restricting the argument of the type integer and asserting that it is positive.

Just look at the first stage (lowered code):

```
code_lowered(fib,(Integer,))
1-element Array{Any,1}:
 :($(Expr(:lambda, {:_n}, {{},{{:_n,:Any,0}}},{}), :(begin
    unless n > 0 goto 0
    goto 1
    0:
    ((top(getfield))(Base,:error))("assertion failed: n > 0")
    1:
    unless n < 3 goto 2
    return 1
```

```
2:  
    return fib(n - 1) + fib(n - 2)  
end)))
```

I'll leave it as an exercise to generate the remaining stages of the coding.

Performance tips

Julia is quick, but poor coding can slow it down markedly. If you are developing enterprise software and/or distributing on GitHub, you need to pay more attention to some of the features and nuances of the language.

The online manual has a long section on performance tips, including a useful style guide and a section of frequently asked questions, which will prove to be illuminating. I've cherry-picked a few that I found of interest.

Best practice

First, a few points about variables, arrays, and types:

- **Global variables:** These may change at different points, so it is hard for the compiler to optimize the code. Wherever possible, variables should be locally defined or passed to functions as arguments. If a variable is actually being used to store a value that does not change, this should be declared as `const`.
- **Changing a variable type:** If you are accumulating a sum of floating point numbers, be careful to initialize the starting point for total as `0.0` and not `0`, as the latter will generate an implicit integer or specify the type explicitly as `total::Float64 = 0`. Similarly, for example, when allocating an array for a list comprehension use a construct such as `zeros(N)` or `Array(Float64, N)`.
- **Accessing arrays:** Julia stores matrices and higher dimensional arrays in column/last index order. This is similar to Fortran, R, and MATLAB but differs from C and Python that use row-ordering. So looping over a matrix, for example, `A[M, N]` is significantly quicker when done as follows:

```
for j = 1:N, i = 1:M  
    A[i, j] = randn()  
end
```

- **Explicitly type fields:** When creating composite types, ensure that the fields are given a specific type as shown in the following code, otherwise the compiler will generate them as `{Any}`:

```
type OrderPair  
    x::Float64
```

```
    y::Float64  
end
```

The next few points refer to functions and their arguments:

- **Allocating function outputs:** If a function returns an array or a composite type, it is better to pre-allocate the output rather than have it allocated within the function body.
- **Measure performance:** The `@time` macro is very useful as it returns both the elapsed time and memory allocation. If memory location is unusually high, it is probably an indication of type instability, that is, a variable is defined as one type but is being converted to a different one.
- **Type-stability:** When possible, ensure that a function returns the same 'type' of variable. As an example, suppose we have a function that returns its argument for positive values but zero when the argument is negative. This should be written as: `pos(x) = (sign(x) < 0) ? zero(x) : x` -- using `zero(x)` rather than just 0, which would return an integer regardless of the type of `x`.
- **Multiple function definitions:** Because different versions of the code are generated for differing argument types, it is better to break a function down to its elemental constituent code as many small definitions allows the compiler to directly call the most applicable code.
- **Kernel functions:** A common functional paradigm is to perform some initial setups, execute the main core (kernel) code, and then do some post-processing to return the function value. It can be beneficial to encapsulate the kernel as a subroutine.

Here are a few miscellaneous tips that seem worthy of note.

You should attend to deprecation warnings, which can arise from changes in the next version of Julia. In addition to being a source of annoyance, a deprecation warning by its nature is only reported one time but the fact that it has been issued needs to be checked each time it is called. This check imposes a real overhead to the execution times.

Here are some annotation macros that can be used as pragmas to the compiler to speed up the execution times:

- `@inbounds` can be used to eliminate array bounds checking within expressions. Out of bounds accesses may cause a crash or corruption of the data, so we need to be sure that the references are appropriate.

- `@fastmath` allows the floating point optimizations that are correct for real numbers but leads to differences for IEEE numbers.
- `@simd` can be placed in front of the `for` loops that are amenable to vectorization, but the documentation for this is may be depreciated at a future date.

Profiling

Profiling is a procedure to determine the time spent in various parts of the code.

In Julia prior to version v0.4, the profiler package used was `IProfile.jl` and profiling was done by means of the `@iprofile` macro.

From v0.4, profiling has been moved in to the Julia base and the macro is now `@profile`, but the operation in both is similar, and here I'm describing the `IProfile` package.

As an example, recall the calculation of the price of an Asian option by the Monte Carlo simulation, which we saw in the first chapter of this book.

All that is needed is to wrap the function code in a `@iprofile begin . . . end` block and then it will be profiled each time it is called:

```
@iprofile begin
    function asianOpt(N=10000,T=100,
                      S0=100.0,K= 100.0,r=0.05,v=0.2,tma=0.25)
        dt = tma/T;
        S = zeros(Float64,T);
        A = zeros(Float64,N);
        for n = 1:N
            S[1] = S0
            dW = randn(T)*sqrt(dt);
            for t = 2:T
                z0 = (r - 0.5*v*v)*S[t-1]*dt;
                z1 = v*S[t-1]*dW[t];
                z2 = 0.5*v*v*S[t-1]*dW[t]*dW[t];
                S[t] = S[t-1] + z0 + z1 + z2;
            end
            A[n] = mean(S);
        end
        P = zeros(Float64,N);
        [ P[n] = max(A[n] - K, 0) for n = 1:N ];
        price = exp(-r*tma)*mean(P);
    end
end
```

The `asianOpt()` function must be in a file (`asian-profile.jl`), which is included in the REPL. This is necessary so that the source code is available to the profiling system.

Now we execute the function once to force its compilation, clear the profile stack, and run the function again:

```
using IProfile
include("asian-profile.jl")
asianOpt(1);
@iprofile clear
asianOpt(100000); # => 2.585150431830136
```

The profile report is output by using `@iprofile report`, as shown in the following:

```
@iprofile report
  count  time(%)   time(s) bytes(%) bytes(k)
    1    0.00    0.000000   0.00      0 # line 3, dt = tma / T
    1    0.00    0.000001   0.00      1 # line 4, S = zeros(Float64,T)
    1    0.11    0.000650   0.44     800 # line 5, A = zeros(Float64,N)
100000  0.17    0.000965   0.00      0 # line 7, S[1] = S0
100000  34.34   0.196964  98.68  179200 # line 8, dW = randn(T)*sqrt(dt)
9900000 16.29   0.093405   0.00      0 # line 10,
    z0 = (r - 0.5*v*v)*S[t-1]*dt
9900000 14.95   0.085731   0.00      0 # line 11,
    z1 = v*S[t-1]*dW[t]
9900000 12.55   0.071965   0.00      0 # line 12,
    z2 = 0.5*v*v*S[t-1]*dW[t]* dW[t]
9900000 20.04   0.114914   0.00      0 # line 13,
    S[t] = S[t-1] + z0 + z1 + z2
100000  1.31    0.007534   0.00      0 # line 15, A[n] = mean(S)
    1    0.06    0.000317   0.44     800 # line 17, P = zeros(Float64,N)
    1    0.18    0.001019   0.44     800 # line 18,
$(Expr(:comprehension,:(P[n] = max(A[n]-K,0)),:(n = 1:N)))
    1    0.01    0.000042   0.00      0 # line 19,
    price = exp(-r * tma) * mean(P)
```

We can see that over 34.34 percent of the time is spent in the generation of the `dW` stochastic array and almost all the remainder time in calculating the `S[]` array.



Profiling can be disabled by using `@iprofile off` and re-enabled by `@profile on`.

Lint

Lint ("fluff") was the name of a program that flagged some suspicious and non-portable constructs in C language source code. The term is now applied to suspicious usage in software written in any computer language and can also refer more broadly to syntactic discrepancies in general.

For example, modern lint checkers are often used to find code that doesn't correspond to certain style guidelines.

In Julia, linting is implemented via the `Lint.jl` package.

It checks for over 50 warnings/errors. For a full list please, look at the online documentation at <https://github.com/tonyhffong/Lint.jl>.

Running this is as easy as follows:

```
using Lint
lintfile( "your_.jl_file" )
```

This becomes recursive in any 'included' source, so there's no need to be checked separately.

The output is in the `file.jl [function name] Line# CODE Explanation` format.

Here, the code is INFO, WARN, ERROR, or FATAL.

I ran the code provided with this book through Lint, and came up with a warning on `etl.jl` (which I've left in!):

```
lintfile("etl.jl")
etl.jl:37 [           ] WARN
  "cat" as a local variable might cause confusion
  with a synonymous export from Base
etl.jl:54 [           ] WARN
  "cat" as a local variable might cause confusion
  with a synonymous export from Base
```



There is no function name as these warnings are in the main part of the code.

If `lintfile()` finds no errors it generates no output, so it is possible to apply an `@isempty()` check on the linting to keep the source(s) 'clean'.

Lint understands about Julia versioning, so it will not complain, for example, about missing `Base.Dates` in 0.3 or missing `Dates` in 0.4:

```
if VERSION < v"0.4-"
    using Dates
else
    using Base.Dates
end
```

Review the documentation to look at the use of the `@lintpragma` macro to suppress unwanted checks, print out values, and add information messages.

Lint can be used with installed packages and it finds them from `.Julia`, in the usual way. As a convenience, a `lintpkg()` separate function is provided:

```
lintpkg("Lint")
```

This unsurprisingly shows no errors!

I'll leave it to you to look at some of the packages you have installed.

To generate more output from file, I've altered in a couple of places the file comprising the Asian option calculation:

```
function asianOpt( N::Int=10000.0,T::Int=100;
    S0=100.0,K= 100.0,v=0.05,r=0.2,tma=0.25)
```

And the following:

```
S[t] = S[t-1] + z0 + z1 + z2 + z3;
```

Also, I added a two lint pragmas near end of code:

```
    .
    .
P = zeros(Float64,N);
@lintpragma "Print type P"
[ P[n] = max(A[n] - K, 0) for n = 1:N ];
price = exp(-r*tma)*mean(P);
end
@lintpragma "Info me I've deliberately hacked this"
```

Running this through Lint gives the following:

```
lintfile("asian-lint.jl")
typeof( P ) == Array{Float64,1}
asian-lint.jl:1 [asianOpt      ]
    ERROR  Duplicate argument: v
```

```
asian-lint.jl:1 [asianOpt      ]
  ERROR  N type assertion and default inconsistent
asian-lint.jl:9 [asianOpt      ]
  ERROR  Use of undeclared symbol r
asian-lint.jl:12 [asianOpt      ]
  ERROR  Use of undeclared symbol z3
asian-lint.jl:19 [asianOpt      ]
  ERROR  Use of undeclared symbol r
asian-lint.jl:21 [           ]
  INFO   I've deliberately hacked this
```

The print type pragma comes first and the messages follow in the next pass.

Debugging

A debugging facility had been recognized as a requirement right back in the days of, now defunct, the JuliaStudio program.

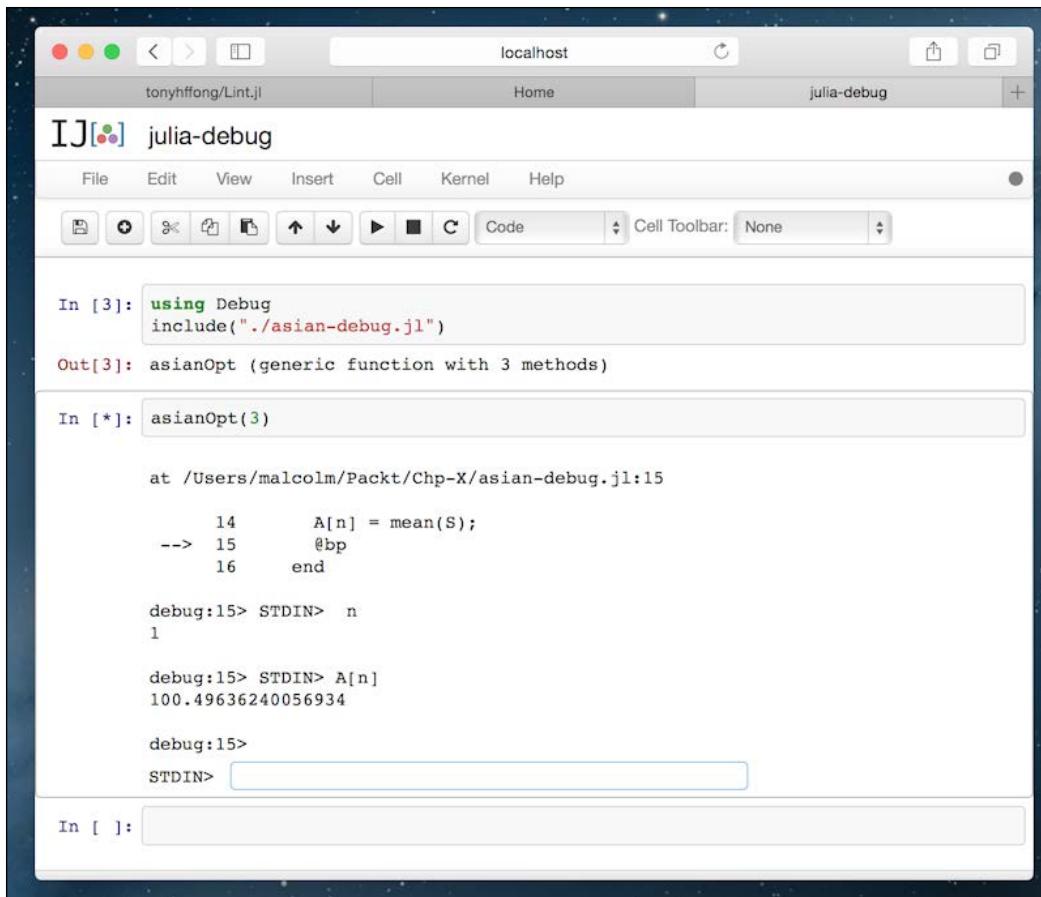
Since Julia compiles to bit code using LLVM, there is ongoing work to utilize its debugging facilities via LLDB.

At present, there is a `Debug.jl` good package that proffers the usual facilities, such as setting breakpoints, printing out values, and much more, and is convenient to use rather than peppering code with a series of the `println()` statements.

To illustrate this, I've taken the `asian-profile.jl` listing, replaced the `@iprofile begin ... end` with `@debug`, and inserted two breakpoints (`@bp`).

As with profiling, the file needs to be included in the REPL, so that `Debug` has access to the source code:

```
julia> using Debug
julia> include("asian-debug.jl")
asianOpt (generic function with 3 methods)
```



The screenshot shows a Mac OS X window titled "localhost" with the URL "tonyhffong/Lint.jl". The tab bar has "Home" and "julia-debug". The main area is titled "IJulia julia-debug". It has a toolbar with File, Edit, View, Insert, Cell, Kernel, Help, and various icons. Below the toolbar is a "Cell Toolbar: None" dropdown.

The interface is divided into sections:

- In [3]:** `using Debug`
`include("./asian-debug.jl")`
- Out[3]:** asianOpt (generic function with 3 methods)
- In [*]:** `asianOpt(3)`
- Code:** Shows the source code of the `asianOpt` function:


```
at /Users/malcolm/Packt/Chp-X/asian-debug.jl:15
  14      A[n] = mean(S);
--> 15      @bp
  16      end

debug:15> STDIN> n
1

debug:15> STDIN> A[n]
100.49636240056934

debug:15>
STDIN> 
```
- In []:** An empty input field.

One breakpoint is in the main loop following the calculation of the mean value of the stock for a single run.

Execute the loop 5 times as follows:

```
julia> asianOpt(5)
at /Users/malcolm/Packt/Chp-X/asian-debug.jl:15
  14      A[n] = mean(S);
--> 15      @bp
  16      end
```

Notice that this works quite well with Jupyter /IPython, see the preceding figure as the notebook opens up a STDIN box for interactive usage.

The debugger has a set of single character commands:

```
h: display help text
s: step into
n: step over any enclosed scope
o: step out from the current scope
c: continue to next breakpoint
l [n]: list n source lines above and below current line
p cmd: print cmd evaluated in current scope
q: quit debug session -- forces an error("interrupted")
```

Any other input is interpreted as a request to display the value of a variable.

This poses a problem in our source if we wish to examine the current value of `n`, and it conflicts with the `step over` command. To work around this, it is possible to type `<space>n`, which will display the value as required:

```
debug:15> n # => 1
debug:15> A[n] # => 105.60875364467921

debug:15> c
debug:15> n # => 2
debug:15> A[n] # => 97.41505241555782
```

Some of the debugger's internal state is available to use through the following session variables:

- `$n`: The current node
- `$s`: The current scope
- `$bp`: Set`{Node}` of enabled breakpoints
- `$nobp`: Set`{Node}` of disabled `@bp` breakpoints
- `$pre`: Dict`{Node}` of grafts

For example, in addition to setting a breakpoint with the `@bp` macro, it is possible to set one "on-the-fly" by setting to the current line and using `$push!($bp, $n)`.

Also, existing breakpoints can be removed by using `$delete!($bp, $n)` or ignored/disabled with `$push!($nobp, $n)`.

So, we can skip out of the loop and pick up a breakpoint after evaluating the Payoff matrix:

```
debug:15> $push!($nobp, $n)
debug:15> c
at /Users/malcolm/Packt/Chp-X/asian-debug.jl:19
18      [ P[n] = max(A[n] - K, 0) for n = 1:N ];
```

```
--> 19      @bp
20      price = exp(-r*tma)*mean(P);
debug:19> P # => [5.6087536446792,0.0,3.64496052229,0.0,2.415052415557]
debug:19> s (twice)
debug:21> price # => 2.3047629672105345
debug: 21> q
```

Developing a package

In this section, I'm going to look at some aspects of package development.

These comprise the various approaches you may adopt to write a package, the procedures to adopt when you want to upload and maintain it via Git as part of a registered Julia module.

I'll conclude by looking at how this strategy might be applied to the case of handling NetPBM images and sketch out the embryonic package, the full source is available in the code downloads accompanying this book.

Anatomy

A minimal package, to be saved to GitHub, should consist of a `README.md` markdown file and the `src` and `test` folders. The `src` folder contains all the code and the `test` folder has one or more test scripts that can be used to check the functionality of the package.

In Julia, the package name is normally chosen in the form of `MyMod.jl` (for example) and there should be a file in the `src` folder also called `MyMod.jl`. This is the main module and may reference other modules (that is, use these modules) or include other Julia files. In fact in many larger packages, the main module basically comprises a set of file inclusions.

Suppose we wish to bundle the code for the `hi()`, `inc()`, `fac()`, and `fib()` functions in `MyMod`, then layout of `MyMod` may be similar to:

```
module MyMod
export hi, inc, fac, fib;
hi() = "Hello"
hi(who::String) = "Hello, $who"
inc(x::Number) = x + 1
function fac(n::Integer)
    @assert n > 0
    (n == 1) ? 1 : n*fac(n-1);
end
```

```
function fib_helper(a ::Integer, b::Integer, n::Integer)
    (n > 0) ? fib_helper(b, a+b, n-1) : a
end
function fib(n::Integer)
    @assert n > 0
    fib_helper(0, 1, n)
end
function pisq(n::Integer)
    (n <= 0) ? error("zero or negative argument") : begin
        s = 1.0
        for k = 1:n
            s += (-1.0)^k/(1.0 + k)^2
        end
        return 12*s
    end
end
end
```

Conventionally, code is not indented between the module ... end, as this would result in the entire code source being indented except in the first and last lines.

Our simple module consists of a series of functions. The export statement contains the names of functions that we want to make visible when the package is referenced by using MyMod.

In our package, all functions except fib_helper() are exported. This function, as the name suggests, is used by the fib() function to provide a tail-recursive procedure to calculate the Fibonacci function, which does not experience the exponential growth in execution times when using a purely recursive algorithm.

To add these to a Julia package, we need to have a few extra files.

First, there should be a LICENSE.md markdown file, stating that the package is licensed under the MIT license and contains a copyright notice listing the package authors. The file is mainly boilerplate and any example from the package you have installed will suffice as a template.

Next, there can be a REQUIREMENTS file. This file lists the packages and Julia version ranges required by this package, and is intended as input to a (future) meta-generator to determine the requirements for the Julia registry. The file is optional and if not present, suggests that there are no dependences.

Finally, there may be a .travis.yml YAML file, which is used by the Julia automatic testing system. Later, we will see how this is constructed.

In addition, you may wish to provide a few additional files or folders. For example, you may add an `examples` folder to contain code that illustrates the use of the package.

You may have a `docs` folder, often with **structured (RST)** files to create only documentation. Also, you may need to supply datasets for the tests and examples in a `data` folder.

The test folder will comprise one (or more) scripts to be used in verifying the correct behavior of the module.

The `Base` class contains a `Test` module that is not loaded when Julia starts up and needs to be imported. It defines and exports a set of macros:

```
@test, @test_fails, @test_throws, @test_approx_eq
```

Also, there are a set of handlers for the various test outcomes: `success`, `failure`, and `error`, which can be imported and modified.

The package tests are driven by a `runtests.jl` script, for large packages this often includes other scripts. The script has to have the `using` clauses referring to both `itself` and `Base.Test`.

So, for our simple `MyMod` package, the `runtests.jl` might comprise the following:

```
using MyMod
using Base.Test
@test hi("Blue Eyes") == "Hello, Blue Eyes "
@test inc(11//7) == 18//7
@test fac(7) == 5040
@test fib(10 == 55
@test_approx_eq inc(2.3 ) 3.3
```

This should provide no output, as all the tests will succeed. When testing floating point arithmetic, it is possible to use `@test` but it is better to use `@test_approx_eq()`, which compares $\sim 1.0e-16$.

Notice this macro takes two arguments, so the `==` is not required.

With the `pisq()` function, we need to be more specific:

```
julia> @test_approx_eq pisq(100000) pi*pi
ERROR: assertion failed: |pisq(100000) - pi * pi| <= 1.77635683e-11
    pisq(100000) = 9.869604401689116
    pi * pi = 9.869604401089358
    difference = 5.997584651140642e-10 > 1.7763568394002505e-11

@test_approx_eq_eps pisq(100000) pi*pi 1.0e-6; # Test is OK
```

You may also wish to look at `FactCheck.jl`, which is a Julia testing framework.

This adds more functionality than that of `Base.Test`. and consists of a series of blocks of assertions such as:

```
facts ("Fact message") do ...
    @fact 'expr' => 'value'
end
```

Related facts can also be grouped as a context inside a `facts` block.

For `MyMod`, I've deliberately added a couple of erroneous facts:

```
facts("MyMod Arithmetic Tests") do
    @fact inc(2.3) # => 3.4
    @fact fac(5) # => 120
    @fact fib(10) # => 54
    @fact pisq(100000) # => roughly(pi*pi, 1.0E-6)
end

MyMod Arithmetic Tests
Failure   :: (line:366) :: got 3.3
           inc(2.3) => 3.4
Failure   :: (line:366) :: got 55
           fib(10) => 54
Out of 4 total facts:
Verified: 2
Failed:   2
```

Taxonomy

"All generalizations are dangerous, including this one."

— Alexander Dumas

I've divided the type of packages under four headings, in order to identify the different strategies that you may adopt when developing a module:

- **Native:** This type of package will rely on Julia coding except that there may be calls to routines in Base and API.
- **Dependent:** Here, a package is dependent on one of the existing ancillary packages, building on prior work. For example, some of the modules in JuliaWeb using libcurl or GnuTLS.

- **Wrapper:** This package does core calling mainly to routines that target a specific shared (C/Fortran) library. It may be necessary to install the library where possible -- either as a binary distro or building from source. In some cases, such as database wrappers, we can assume the library exists otherwise there is no point of this package.
- **Tasking:** Base functionality comes from a separate task. A couple of examples might make this clear, those being PyCall and Gaston. These will impose platform specificity. For example, Gaston works well with Linux, it is possible for it to work on OS X using Quartz, but is difficult on Windows.

Clearly, there may be some cases where a package adopts a mixed strategy spanning more than one class. Under such circumstances, it may be useful to create a submodule as part of the main package.

While interfacing with a specific system, you may be able to write native Julia code or make calls to an existing shared library. These may be identified as high-level and low-level interfaces and the following are the steps you may consider:

- Before deciding on implementation strategy for your package, you should briefly list the functionality intended as a first draft. This will include a set of "must have" routines to make the module practicable, together with some other functionality you consider as desirable but which you are able to consign to a TODO list.
- Next define on any types which will be useful in encapsulating the module's data structures.
- Following on this define constants and construct macros which will be of use by the package functions.
- Finally write the functions and construct the tests which form the basis for the `runtests.jl` script.

In developing the package, you will normally have developed some extensive Julia snippets. This will be of great benefit to users of your package, and should be included as part of an `examples` subdirectory, without forgetting to include any required data files, such as CSV datasets and images.

Finally, provide reasonably thorough documentation, either in terms of a comprehensive `README.md` markdown file or, what is becoming increasingly popular, by means of the Sphinx framework and RST files as part of the <https://readthedocs.org/> online system.

Using Git

While constructing a package that is to be loaded by Julia and ultimately published to the web, a number of different files are required.

Before creating these, it is necessary to create a `.gitconfig` file in your home directory by entering:

```
git config --global user.name " Malcolm Sherrington"
git config --global user.email "malcolm@amisllp.com"
git config --global github.user "sherrinm"
```

Skeleton versions of the package files can be created within the Julia environment using the package manager:

```
julia> Pkg.generate("MyMod", "MIT")
INFO: Initializing MyMod repo: /Users/malcolm/.julia/v0.3/MyMod
INFO: Origin: git://github.com/sherrinm/MyMod.jl.git
INFO: Generating LICENSE.md
INFO: Generating README.md
INFO: Generating src/MyMod.jl
INFO: Generating test/runtests.jl
INFO: Generating .travis.yml
INFO: Generating .gitignore
INFO: Committing MyMod generated files
```

This adds the embryonic package to the current repository corresponding to the version of Julia being executed (that is, v0.3, v0.4) It can be loaded, but has no functionality until your versions of `MyMod.jl` and `runtests.jl` are uploaded to replace the stubs.

The license file (`LICENSE.md`) will pick your name from the `.gitconfig` file and probably can be left as it is. On the other hand, the `README.md` file contains only a main title line consisting of the package name and should be given more attention.

The `Pkg.generate()` also creates the Travis YAML file (`.travis.yml`) and a `.git` subdirectory that contains sufficient information for use with the GitHub system:

This can be listed using `git show`:

```
$ cd /Users/malcolm/.julia/v0.3/MyMod
$ git show
commit e901cc6a3d1fbad5b1ec6dfa03d0cde324aa4b1c
Author: Malcolm Sherrington <malcolm@amisllp.com>
Date:   Fri Apr 10 13:53:44 2015 +0100
    license: MIT
    authors: Malcolm Sherrington
    years:   2015
    user:    sherrinm
```

Although optional, and hence not generated, you will probably need to create a REQUIRE file that comprises the version(s) of Julia under which the module should work correctly, together with a list of other packages to be included when the package is added.

It is possible to use the @osx and @windows constructs to indicate modules required for specific operating systems.

For example, the REQUIRE file for Cairo.jl (currently) is:

```
Compat 0.3.5
Color
BinDeps
Graphics 0.1
@osx Homebrew
@windows WinRPM
```

Publishing

At present your package is only available locally on your own computer. To make it available to more developers, you will need an account on GitHub account; if you have filled in the `github.user` config entry in the preceding code, you will already have one, if not then register for one, it's free.

First, you need to create the remote repository by pushing your code into GitHub. The Julia package manager does not do this automatically, so you need to use `git` itself. There is also a command line tool `hub` (<https://github.com/github/hub>) that simplifies the syntax markedly.

This can be used to create in the package repository and have it automatically uploaded via GitHub's API. Once done, it is possible to use the published repo using `Pkg.clone()`:

```
julia> Pkg.clone("git://github.com/sherrinm/MyMod.jl")
```

At this stage, it is a good idea to have a few people look at the package and provide some feedback. Later, you can decide to have it registered as an official Julia package.

For this, you need to add it to your "local" copy of METADATA using `Pkg.register()`:

```
julia> Pkg.register("MyMod")
INFO: Registering MyMod at
      git://github.com/sherrinm/MyMod.jl.git
INFO: Committing METADATA for MyMod
```

This creates a commit in the `~/.julia/v0.3/METADATA`. This is still locally visible and in order to make it visible globally, you need to merge your local `METADATA` upstream to the official repository.

The `Pkg.publish()` command will fork the `METADATA` repository on GitHub, push your changes to your fork, and open a pull request. If this fails with an error, it is possible to publish manually by "forking" the main `METADATA` repository and pushing your local changes using `git`; the procedure for this is discussed in detail in the Julia documentation.

If you are making an official version of your package or releasing a new version, you should give it a version number using `Pkg.tag()`:

```
julia> Pkg.tag("MyMod")
INFO: Tagging MyMod v0.0.1
INFO: Committing METADATA for MyMod
```

You can confirm the tag number using `cd /Users/malcolm/.julia/v0.3/MyMod && git tag`.

If there is a `REQUIRE` file in your package, it will be copied to the appropriate place in `METADATA` when tagging the (new) version.

Note that it is possible to specify a specific version number using the `Pkg.tag()` command by supplying a second argument such as `v"1.0.0"`.

Alternatively, you can make use one of the `:patch`, `:minor`, or `:major` symbols to increment the patch, with minor or major version number of your package accordingly.

As with `Pkg.register()`, these changes to `METADATA` are only locally available until they've been uploaded to GitHub. Again this is done using `Pkg.publish()`, which ensures the package(s) repos have been tagged and opens a pull request to `METADATA`.

Community groups

The number of packages written in Julia continues to grow at a superlinear rate. At the time of writing, there are more than 600 registered packages and many more available from programmers on GitHub but as yet unregistered with the Julialang group.

As usual, developers tended to make use of their own packages when writing new ones than those of colleagues and eventually those from the community at large.

This has led to the formation of groups of developers with shared interests, these are termed community groups and are referenced on the Julia website as part of the community section (<http://julialang.org/community/>).

Some of the larger groups such as JuliaStats and JuliaOpt maintain their own websites separate from GitHub and individual package documentation sources may also be found at <http://readthedocs.org>.

For example, the JuliaStats web pages are on <http://statsbasejl.readthedocs.org> and the documentation for the StatsBase package is to be found on <http://statsbasejl.readthedocs.org>.

Thus, you should not only look at community group packages but are encouraged to use the web search engines, in order to pick up recent work that has not been registered with Julia.

Classifications

As I commented earlier when discussing the types of packages, applying any specific taxonomy to software is not always easy and many exceptions occur to test the rule. Nevertheless, I feel there is enough to be gained in an overview of community groups to attempt the process again.

Therefore, I'm going to classify the types of groups under three main headings:

- **General purpose:** These correspond to topics that may be of use to the analyst working in his/her own field. Obvious examples of these are statistics, optimization, and database.
- **Topic specific:** These are more applicable to persons working in that particular area. At present there are embryonic groups for BioJulia and JuliaQuantum. Also, in the financial sector we have previously met one or two packages from JuliaQuant, such as TimeSeries, MarketData, and Quandl.
- **Niche:** This is harder to define, but I am looking at topics that cover the use of specific hardware or a less conventional computing paradigm such as the use of GPUs (JuliGPU) or parallel computing (JuliaParallel).

In the next couple of sections, I'm going to provide a few examples of groups for the latter two categories that appeal to me, not necessarily in my professional career, but in my wider interests as a whole. I will avoid statistics, optimization, and finance, as you will have encountered many examples of these earlier in this book.

JuliaAstro

As a member of the Baker Street Irregulars Astros, a curious set of enthusiasts who meet regularly in Regent's Park to prove that it is possible to see something in the London skies other than the moon and also as a software consultant who has worked for the European Space Agency and German Aerospace (DLR), my attention was grabbed by the JuliaAstro group and I'm going to give a couple of examples chosen from there.

JuliaAstro has its own web pages at <http://juliaastro.github.io/>, and this contains links to documentation for individual packages, plus a reference to the julia-astro mailing list.

This can be thought of as the equivalent of the Python astropy modules (<http://www.astropy.org/>) that are distributed as part of the Anaconda sources.

Cosmology models

Firstly, I'll look at `Cosmology.jl` that defines various models of the universe depending on the amount of real matter, dark matter, dark energy, and so on. This is reasonably compact, native code package that defines a set of abstract cosmologies and then goes on to establish specific instances depending on the parameterization. An instance can then be used to establish distance and time estimates for this particular model.

Cosmological models had their origin in the work of Friedmann in the 1920s, in solving Einstein's equations for general relativity. This established the principle that a "static" universe would be unstable and led to Einstein inserting a repulsion term, the cosmological constant (Λ) in his equations, which following the work of Hubble on the expanding universe, Einstein came to regret.

However, current thoughts that the universe is not only expanding, but doing so at an increasing rate, has led to the introduction of cosmic repulsion again, in terms of the postulation of dark energy.

The original Friedmann solution does not account for any repulsion term, that is, only assumes mass and gravitational attraction. It defines a critical density above which the universe is closed (elliptic) and will eventually contract, or density below which it is open (hyperbolic) and will continue to expand. A density parameter, Ω , is defined as the ratio of the matter's actual density to the critical density.

So if $\Omega = 1$, the universe is termed as flat, that is, Euclidian. For all the other cases, the curvature is non-Euclidian.

The principle function in the `Cosmology.jl` is `cosmology()` that consists only of named parameters as follows:

```
function cosmology();
    h = 0.69, # Dimensionless Hubble parameter
    Neff = 3.04, # Effective number of neutrino species
    OmegaK = 0, # Curvature density
    OmegaM = 0.29, # Matter density
    OmegaR = nothing, # Radiation density
    Tcmb = 2.7255, # Cosmic Microwave Background Temperature
    w0 = -1, # Dark energy equation of state ...
    wa = 0 ) # ... w0 + wa*(1 - a)
```

If we just run the function with the default values:

```
c = cosmology()
FlatLCDM(0.69,0.7099122024007928,0.29,8.77975992071536e-5)
```

This returns a *Flat* universe and looking at its values as follows:

```
julia> names(c)
1x4 Array{Symbol,2}:
 :h   :Ω_Λ   :Ω_m   :Ω_r
```

The Omegas are the densities for dark energy, total matter (dark + baryonic), and radiation:

```
So: julia> c.Ω_Λ + c.Ω_m + c.Ω_r; # => 0.99999..... ~ 1.0
```

`OmegaL` (Ω_Λ) is computed from the `OmegaK` curvature density, which is the Friedmann constant (ρ / ρ_{crit}) minus 1, that is, it is zero-based.

If `OmegaR` (Ω_R) is set to nothing, then it is calculated from effective number of neutron species (`Neff`) and the background cosmic radiation temperature (`Tcmb`).

By forcing `OmegaL` to be computed, we can ensure the Open (`OmegaK > 0`) or Closed (`OmegaK < 0`) cosmological models.

This package is similar to some online calculators such as Ned Wright's at UCLA (<http://www.astro.ucla.edu/~wright/CosmoCalc.html>). In this package, we input the Hubble factor as a dimensionless quantity, whereas the UCLA calculator uses the actual value.

Other routines are available to compute distances, volumes, and times for various models.

Not all are exported, but are still available by fully qualifying the function call:

```
Cosmology.hubble_dist_mpc0(c); # => 4344.8182, Hubble distance
Cosmology.hubble_time_gyr0(c); # => 14.1712, Hubble time
```

It is possible to apply a redshift factor to these calculations as follows:

```
z = 1.7; Cosmology.Z(c,z); # => 1.
Cosmology.hubble_dist_mpc(c,z); # => 1714.4094
Cosmology.hubble_time_gyr(c,z); # => 5.5918
```

Also, as with the results of the UCLA calculator, we can obtain estimates of the universe metrics, such as diameter, volume, and age. This is done as shown in the following commands:

```
comoving_transverse_dist_mpc(c, z); # => 4821.0198 Mpc
angular_diameter_dist_mpc(c,z); # => 1785.5628 Mpc
luminosity_dist_mpc(c,z); # => 13016.753 Mpc
comoving_volume_gpc3(c,z); # => 469.3592 Gpc3
age_gyr(c,z); # => 3.872 GYr
where
1 Gyr = 1,000,000,000 years.
1 Mpc = 1,000,000 parsecs = 3,261,566 light years.
```

The Flexible Image Transport System

Now, I'll discuss the use of the **Flexible Image Transport System (FITS)** that was standardized in the early 1980s and is most commonly used in digital astronomy.

A FITS file consists of one (primary) or more **header and data units (HDUs)**. It is entirely possible that an HDU may consist entirely of a header with no data blocks.

Usually a FITS file may consist of a primary header followed by a single primary data array. This is also known as Basic FITS or Single Image FITS (SIF).

Alternatively, a multiextension FITS (MEF) file contains one or more extensions following the primary HDU.

Header information may be in two formats: fixed (80 bytes) or free.

These consist of a keyword, followed by a value, and additionally a comment describing the field; = and / are used to delimit the keyword, value, and comment. There are some mandatory keywords that must be provided in fixed format.

The last field in the HDU just comprises the END word.

A basic FITS HDU for the Cygnus X-ray source (Cygnus X1) would be similar to:

```

SIMPLE = T / Conforms to the FITS standard
BITPIX = 16 / number of bits per data pixel
NAXIS = 2 / number of data axes
NAXIS1 = 250 / length of data axis 1
NAXIS2 = 300 / length of data axis 2
OBJECT = 'Cygnus X-1'
DATE = '2006-10-22'
END

```

This is a two-dimensional dataset (250x300), taken on October 22, 2006.

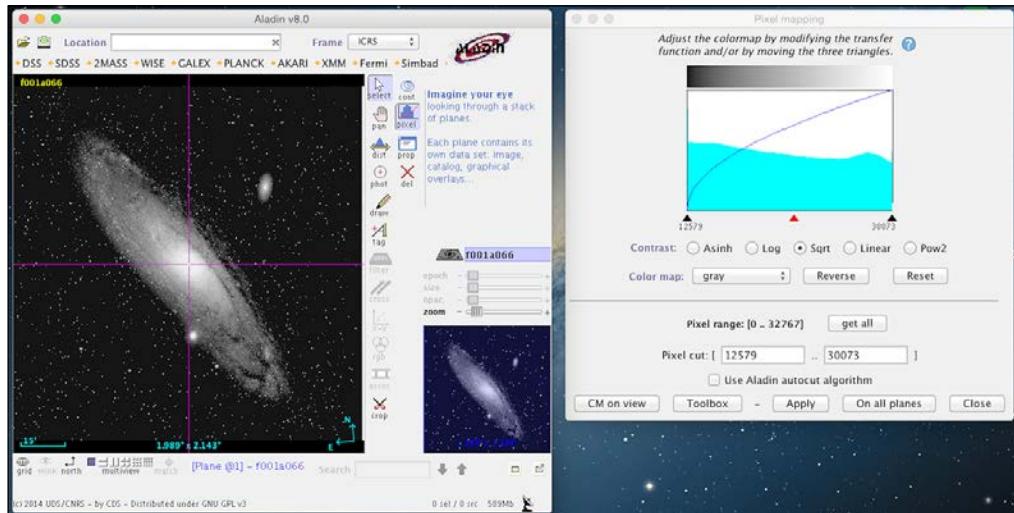
It is usual to provide more information in the header, for example, where the image was acquired.

FITS files can be viewed by a variety of software packages; on Linux/OS X, we can use GIMP and on Windows, we can use IfranView.

Also, there is a specialist program *Aladin*, which is written in Java, and runs on all platforms. Aladin is developed by the *Centre de Données astronomiques de Strasbourg* and described as a *Sky Atlas*. It provides the combination of various FITS images, enabling the possibility to combine and view object imagery acquired from IR, visible, UV, and X-ray telescopes.

I'm going to look at an image of our big sister, the Andromeda Galaxy (M31), sourced from the NASA/ESA Hubble Space Telescope, and catalogued as f001a066 using the FITSIO.jl.

A view of the file as seen via Aladin is given in the following figure:



`FITSIO.jl` has provided methods of working with FITS files, which it terms as high-level and a low-level APIs.

The low-level API is a wrapper around NASA's C-library and has a great deal of flexibility; the high-level API is a native Julia package.

Both are excellently documented at <https://julia-fitsio.readthedocs.org>.

The high-level API

The high-level API uses a FITS constructor to read a FITS dataset:

```
julia> f001a = FITS("f001a066.fits")
file: f001a066.fits
mode: r
extnum exttype      extname
image_hdu

julia> names(f001a)'
1x4 Array{Symbol,2}:
 :fitsfile  :filename  :mode   :hdus
```

The file `f001a066.fits` has a single HDU, referenced as the first element in the array:

```
julia> f001a[1]
file: f001a066.fits
extension: 1
type: IMAGE
image info:
  bitpix: 16
  size: (7055,7055)
```

We can get the number of dimensions of the imagery and its size as follows:

```
ndims(f001a[1]); # => 2
size(f001a[1]); # => (7055,7055)
```

Also, the `read()` function will return the entire image and the `readheader()` will create a parsed representation of the HDU's metadata:

```
data    = read(f001a[1]);
header = readheader(f001a[1]); # =>
length(header); # => 128

header["BITPIX"]; # => 16
header["SCANIMG"]; # => "XJ295_A066_01_00.PIM"
header["TELESCOP"]; # => "Palomar 48-in Schm"
```

Our image was taken by the Mount Palomar Samuel Orchin 48 inch telescope.

Along with the header values, it is possible to get the comment fields as shown in the following code:

```
getcomment(header, "TELESCOP")
"Telescope where plate taken"
```

The header also contains the latitude and longitude of the Mount Palomar site and the date and time when the image was acquired.

```
header["SITELAT"]; # => "+33:24:24.00      "
header["SITELONG"]; # => "-116:51:48.00      "
```

The FITS constructor can take a second argument termed as the mode that can be "r" (read-only), "r+" (read-write), or "w" (write). The default is "r" but it is possible to create a new file using a mode of "w". In this mode, any existing file with the same name will be overwritten:

```
julia> f = FITS("newfile.fits", "w");
```

When created, we write the image data and optionally the header to the file using the `write()` function:

```
data = reshape([1:120000], 300, 400);
write(f, data) # Write a new image extension with the data
write(f, data; header=my_header) # Also write a header
```

The low-level API

The low-level API is an interface to the CFITSIO library and has much more functionality. If we consider just reading meta-information, this is returned unparsed as a series of triple string ("key", "value", "comment"), one per line:

```
using FITSIO
f001 = fits_open_file("f001a066.fits")
FITSFile(Ptr{Void} @0x00007faf9c4de6d0)

n = fits_get_hdrspace(f001)[1]; # => 128
128 entries in the header

for i = 1:n
    println(fits_read_keyn(f001,i))
end
("SIMPLE", "T", "FITS header")
```

```
("BITPIX", "16", "No.Bits per pixel")
("NAXIS", "2", "No.dimensions")
("NAXIS1", "7055", "Length X axis")
("NAXIS2", "7055", "Length Y axis")
("EXTEND", "T", "")
("DATE", "'09/07/04           '", "Date of FITS file creation")
("ORIGIN", "'CASB -- STScI      '", "Origin of FITS image")
("PLTLABEL", "'SJ01467          '", "Observatory plate label")
("PLATEID", "'A066             '", "GSSS Plate ID")
("REGION", "'XJ295            '", "GSSS Region Name")
("DATE-OBS", "'1987/09/20        '", "UT date of Observation")
```

This is a series of tuples that will need to be split in to the separate consistent fields programmatically.

Alternatively, it is possible to call the `fits_hdr2str()` routine to obtain the entire header information as a single string. This consists of fixed length records of size 80 bytes with the = and / delimiters in the ninth and thirty-second positions respectively:

```
DATE-OBS= 1987/09/20 / UT date of Observation
```

JuliaGPU

One of the main themes of my professional career has been working with hardware to speed up the computing process. In my work on satellite data, I worked with the STAR-100 array processor and once back in the UK, used Silicon Graphics for 3D rendering of medical data. Currently, I am interested in using NVIDIA GPUs in financial scenarios and risk calculations.

Much of this work has been coded in C, with domain-specific languages to program the ancillary hardware. It is possible to do much of this in Julia with packages in the JuliaGPU group.

This has routines for both CUDA and OpenCL, at present covering:

- **Basic runtime:** CUDA.jl, CUDArt.jl, and OpenCL.jl
- **BLAS integration:** CUBLAS.jl, CLBLAS
- **FFT operations:** CUFFT.jl, CLFFT.jl

The CU* style routines only applies to NVIDIA cards and requires the CUDA SDK to be installed, whereas the CL* functions can be used with variety of GPUs. CLFFT and CLBLAS do require some additional libraries to be present, but we can use OpenCL as it is. The following is output from a Lenovo Z50 laptop with an i7 processor and both Intel and NVidia graphics chips:

```
julia> using OpenCL
julia> OpenCL.devices()
OpenCL.Platform(Intel(R) HDGraphics 4400)
OpenCL.Platform(Intel(R) Core(TM) i7-4510U CPU)
OpenCL.Platform( GeForce 840M on NVIDIA CUDA)
```

Here is a script that will list the platforms (GPUs) present on your computer:

```
using OpenCL
function cl_platform(pname)
    @printf "\n%s\n\n" pf
    for pf in OpenCL.platforms()
        if contains(pf[:name], pname)
            @printf "\n%s\n\n" pf
            @printf "Platform name:\t\t%s\n" pf[:name]
            if pf[:name] == "Portable Computing Language"
                warn("PCL platform is not yet supported")
                continue
            else
                @printf "Platform profile\t\t:%s\n" pf[:profile]
                @printf "Platform vendor:\t\t%s\n" pf[:vendor]
                @printf "Platform version:\t\t%s\n\n" pf[:version]

                for dv in OpenCL.available_devices(pf)
                    @printf "Device name:\t\t%s\n" dv[:name]
                    @printf "Device type:\t\t%s\n" dv[:device_type]
                    gms = dv[:global_mem_size] / (1024*1024)
                    @printf "Device memory:\t\t%i MB\n" gms
                    mma = dv[:max_mem_alloc_size] / (1024*1024)
                    @printf "Device max memory alloc:\t\t%i MB\n" mma
                    mcf = dv[:max_clock_frequency]
                    @printf "Device max clock freq:\t\t%i MHZ\n" mcf
                    mcu = dv[:max_compute_units]
                    @printf "Device max compute units:\t\t%i\n" mcu
                    mwgs = dv[:max_work_group_size]
                    @printf "Device max work group size:\t\t%i\n" mwgs
                    mwis = dv[:max_work_item_size]
                    @printf "Device max work item size:\t\t%s\n" mwis
                end
            end
        end
    end
end
```

```
    end
end
end
```

I ran this routine on a laptop with both Intel HD and NVIDIA GeForce chips that gave the following results and selected the NVIDIA GeForce device for a specific platform profile:

```
julia> cl_platform("NVIDIA")
OpenCL.Platform('Intel(R) OpenCL' @x000000012996290)
OpenCL.Platform('NVIDIA CUDA' @x00000000519ef60)
Platform name:          NVIDIA CUDA
Platform profile:       FULL PROFILE
Platform vendor:        NVIDIA Corporation
Platform version:       OpenCL 1.1 CUDA 6.0.1
Device name:            GeForce 840M
Device type:            gpu
Device memory:          4096 MB
Device max memory alloc: 1024 MB
Device max clock freq:   1124
Device max compute units: 3
Device max work group size: 1024
Device max work item size: (1024,1024,64)
```

To do some calculations, we need to define a kernel to be loaded on the GPU. The following multiplies two 1024x1024 matrices of Gaussian random numbers:

```
import OpenCL
const cl = OpenCL
const kernel_source = """
__kernel void mmul(
    const int Mdim, const int Ndim, const int Pdim,
    __global float* A, __global float* B, __global float* C) {
    int k;
    int i = get_global_id(0);
    int j = get_global_id(1);
    float tmp;
    if ((i < Ndim) && (j < Mdim)) {
        tmp = 0.0f;
        for (k = 0; k < Pdim; k++)
            tmp += A[i*Ndim + k] * B[k*Pdim + j];
        C[i*Ndim+j] = tmp;
    }
}
"""


```

The kernel is expressed as a string and the OpenCL DSL has a C-like syntax:

```
const ORDER = 1024; # Order of the square matrices A, B and C
const TOL = 0.001; # Tolerance used in floating point comps
const COUNT = 3; # Number of runs

sizeN = ORDER * ORDER;
h_A = float32(randn(ORDER)); # Fill array with random numbers
h_B = float32(randn(ORDER)); # --- ditto --
h_C = Array(Float32, ORDER); # Array to hold the results

ctx = cl.Context(cl.devices() [3]);
queue = cl.CmdQueue(ctx, :profile);

d_a = cl.Buffer(Float32, ctx, (:r,:copy), hostbuf = h_A);
d_b = cl.Buffer(Float32, ctx, (:r,:copy), hostbuf = h_B);
d_c = cl.Buffer(Float32, ctx, :w, length(h_C));
```

Now, we create the Open CL context and some data space on the GPU for the d_A, d_B, and D_C. arrays

Then we copy the data in the h_A and h_B host arrays to the device and then load the kernel onto the GPU:

```
prg = cl.Program(ctx, source=kernel_source) |> cl.build!
mmul = cl.Kernel(prg, "mmul");
```

The following loop runs the kernel for COUNT times to give an accurate estimate of the elapsed time for the operation.

This includes the cl-copy! () operation that copies the results back from the device to the host (Julia) program:

```
for i in 1:COUNT
    fill!(h_C, 0.0);
    global_range = (ORDER, ORDER);
    mmul_ocl = mmul[queue, global_range];
    evt = mmul_ocl(int32(ORDER), int32(ORDER),
                    int32(ORDER), d_a, d_b, d_c);
    run_time = evt[:profile_duration] / 1e9;
    cl.copy!(queue, h_C, d_c);
    mflops = 2.0 * Ndims^3 / (1000000.0 * run_time);
    @printf "%10.8f seconds at %9.5f MFLOPS\n" run_time mflops
end
0.59426405 seconds at 3613.686 MFLOPS
0.59078856 seconds at 3634.957 MFLOPS
0.57401651 seconds at 3741.153 MFLOPS
```

This compares with the figures to run this natively without the GPU processor:

7.060888678 seconds at 304.133 MFLOPS

That is, using the GPU gives a 12-fold increase in the performance of matrix calculation.

What's missing?

We have covered lot of ground in this book. The aim, as I stated at the outset, was to provide sufficient material for you to be able to use Julia in your work now. Out of necessity, this has been broad rather than deep; there are so many topics to cover, each of which could merit a book on its own.

I was told when starting this book that it would be harder to decide what to leave out rather than what to put in and this has been proven to be true. So I've listed some of the other facets of Julia that I researched but eventually omitted:

- OpenGL graphics
- Parallel processing
- Functional programming
- Symbolic computation
- Econometrics
- Neural nets and Bayesian statistics
- Natural language processing

There is significant work by the Julia community in all these fields and many more.

Julia is a rapidly changing language, it has not yet reached the v1.0 status. It is probably better to get over the 'pain' quickly rather than go through the machinations that Perl and Python have undergone of late and the consequences that may follow.

Doubtless, things will have changed, new packages added, and others decaying through lack of attention. However, Julia is a very useable language, even as I write this and more so when you are reading this.

The Julia website is your friend. It is well maintained and gives links to all important resources. Also, you need to be aware and participate in the various mail lists and Google groups.

If you are an academic, use it in research, or if a data scientist use it in your analysis. Superficially, it is a very easy language to learn but like the proverbial onion peel away one layer and another will appear.

Julia is strongly typed, but you can work without types. Julia uses multiple dispatch but you need not be aware of it. Above all, you can free yourself from being a user of a language to being a developer in one.

Julia gives you all that at no extra cost.

As I said at the beginning of this chapter, "it turtles all the way down", almost.

Summary

This concludes our whirlwind review of Julia. It is worth re-emphasizing to the reader that Julia is not yet v1.0, so some of the features described may be deprecated or permanently changed as you read this book. The Julia documentation and its code listings provide all the answers and hopefully you will be able to find your way around these by now.

In this chapter, we addressed a variety of topics that should be of use to you in moving from a analyst/programmer to a contributor and developer.

We began with a peek "under the hood" to find out why Julia does what it does and quickly. At present, most computer languages utilizing LLVM do it via a separate compilation stage. The magic of Julia is that it takes a conventional-style procedural code and takes this way down to the machine level in a single process.

Then we continued with thoughts on some tips and techniques that will be of help in creating enterprise-quality Julia code and covered topics such as testing, profiling, and debugging. This led to the discussion of a taxonomy of Julia modules and the considerations and steps involved in developing your own package.

Finally, we highlighted the work done by Julia Community groups, identifying (imperfectly) three types: general purpose, topic-based, and specialist. Since most of the examples in this book have been from the first type, we closed by looking at the work of a couple of examples in the second and third categories: JuliaAstro and JuliaGPU.

I believe that Julia stands out as a scripting language at present, as it is both very simple to use and extremely powerful. It allows the analyst to transcribe his/her pseudocode almost directly to Julia, and also empowers the developer to achieve some amazing things while remaining within the language.

If you have got to this point, I hope you share my opinions and that this book will have helped you in some small way in becoming a Julian.

Bibliography

This learning path has been prepared for you to help you program and create high performance code with Julia and to help you understand Julia better. It comprises of the following Packt products:

- *Getting Started with Julia, Ivo Balbaert*
- *Julia High Performance, Avik Sengupta*
- *Mastering Julia, Malcolm Sherrington*