Mastering Julia

Malcolm Sherrington

Julia is a well-constructed programming language with fast execution speed, eliminating the classic problem of performing analysis in one language and translating it for performance into a second. This book will help you develop and enhance your programming skills in Julia to solve real-world automation challenges.

This book starts off with a refresher on installing and running Julia on different platforms. Next, 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.

Who this book is written for

This hands-on guide is aimed at practitioners of data science. The book assumes some previous skills with Julia and skills in coding in a scripting language such as Python or R, or a compiled language such as C or Java.

What you will learn from this book

- Install and build Julia and configure it with your environment
- Build a data science project through the entire cycle of ETL, analytics, and data visualization
- Understand the type system and principles of multiple dispatch for a better coding experience in Julia
- Interact with data files and data frames to study simple statistics and analytics
- Display graphics and visualizations to carry out modeling and simulation in Julia
- Use Julia to interact with SQL and NoSQL databases
- Work with distributed systems on the Web and in the cloud
- Develop your own packages and contribute to the Julia Community

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'The Julia Environment'
- A synopsis of the book’s content
- More information on Mastering Julia
Malcolm Sherrington has been working in computing for over 35 years. He holds degrees in mathematics, chemistry, and engineering and has given lectures at two different universities in the UK as well as worked in the aerospace and healthcare industries. Currently, he is running his own company in the finance sector, with specific interests in High Performance Computing and applications of GPUs and parallelism.

Always hands-on, Malcolm started programming scientific problems in Fortran and C, progressing through Ada and Common Lisp, and recently became involved with data processing and analytics in Perl, Python, and R.

Malcolm is the organizer of the London Julia User Group. In addition, he is a co-organizer of the UK High Performance Computing and the financial engineers and Quant London meetup groups.
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 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 book 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 Linux, 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 book will discuss both the Jupyter and Juno IDEs.

What this book covers

Chapter 1, The Julia Environment, deals with the steps needed to get a working distribution of Julia up and running. It is important to be able to acquire the latest sources and build the system from scratch, as well as find and install appropriate packages and also to remove them when necessary.

Chapter 2, Developing in Julia, is a quick overview of some of Julia's basic syntax. Julia is a new language, but it is not unfamiliar to readers with a background in MATLAB, R, or Python, so the aim of the chapter is to briefly bring readers up to speed, using examples, with Julia and to point them to online sources. Also, it is important to be aware of the differences between working via the console in contrast to the JuliaStudio IDE.

Chapter 3, Types and Dispatch, looks at the Julia type system and shows how this exposes powerful techniques to the developer by means of its de facto functional dispatch system.

Chapter 4, Interoperability, covers the methods by which Julia can interact with the operating system and other programming languages. These methods are largely native to Julia and the chapter concludes with an introduction to parallelism that is discussed further in Chapter 9, Networking.

Chapter 5, Working with Data, begins the journey the data scientist would take from data source to analytics results. Most projects begin with data, which has to be read, cleaned up, and sampled. The chapter starts here and goes on to describe simple statistics and analytics.
Chapter 6, *Scientific Programming*, is seen as a principle reason to program in Julia. Its strength is the speed of execution combined with the ease of developing in a scripting language that makes it particularly useful in tackling compute-bound processes. The chapter looks at various techniques used in approaching mathematical and scientific problems.

Chapter 7, *Graphics*, in Julia is often compared unfavorably to other alternate languages such as MATLAB and R. While earlier versions of the language had limited graphics options, this is certainly not the case now and this chapter describes a wide variety of sophisticated approaches both to display to screen and save to disk files.

Chapter 8, *Databases*, deals with interaction with databases in Julia. Data to be analyzed may be stored in a database or it may be necessary to save the results in a database after analysis. Various approaches are considered for SQL and NoSQL datastores. These are not built in to the language, rather rely totally on contributed packages, and so may be enhanced in the near future.

Chapter 9, *Networking*, covers aspects of working with distributed data sources. Big data and cloud systems are becoming more prevalent in data science and the chapter covers network programming at the socket level and interfacing via the Web. Also, it includes a discussion on running Julia on Amazon Web Services and the Google compute server.

Chapter 10, *Working with Julia*, aims to provide information and encouragement to go on and contribute as a Julia developer. This may be as a sole author contributing to an existing package or as a member of the Julia groups.
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):

<table>
<thead>
<tr>
<th></th>
<th>Fortran</th>
<th>Julia</th>
<th>Python</th>
<th>R</th>
<th>MATLAB</th>
<th>Octave</th>
<th>Mathematica</th>
<th>Java Script</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>0.26</td>
<td>0.91</td>
<td>30.37</td>
<td>411.31</td>
<td>1992.0</td>
<td>3211.81</td>
<td>64.46</td>
<td>2.18</td>
<td>1.0</td>
</tr>
<tr>
<td>mandel</td>
<td>0.86</td>
<td>0.85</td>
<td>14.19</td>
<td>106.97</td>
<td>64.58</td>
<td>316.95</td>
<td>6.07</td>
<td>3.49</td>
<td>2.36</td>
</tr>
<tr>
<td>pi_sum</td>
<td>0.80</td>
<td>1.00</td>
<td>16.33</td>
<td>15.42</td>
<td>1.29</td>
<td>237.41</td>
<td>1.32</td>
<td>0.84</td>
<td>1.41</td>
</tr>
<tr>
<td>rand_mat_stat</td>
<td>0.64</td>
<td>1.66</td>
<td>13.52</td>
<td>10.84</td>
<td>6.61</td>
<td>14.98</td>
<td>4.52</td>
<td>3.28</td>
<td>8.12</td>
</tr>
<tr>
<td>rand_mat_mul</td>
<td>0.96</td>
<td>1.01</td>
<td>3.41</td>
<td>3.98</td>
<td>1.10</td>
<td>3.41</td>
<td>1.16</td>
<td>14.60</td>
<td>8.51</td>
</tr>
</tbody>
</table>

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.


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](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 Julia Environment

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 Julia Environment

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

Building from source

Ubuntu is part of the Debian family of Linux distributions, others being Debian itself, Linux Mint Debian Edition (LMDE), and Knoppix. All can install DEB packages and use the previous apt-get command.

The other major Linux family is based on the Red Hat distribution: Red Hat Enterprise, CentOS, Fedora, Scientific Linux, and so on. These use a different package management mechanism based on RPM files. There are also distros based on SUSE, Mandriva, and Slackware.


The link is again available from the julialang.org downloads page. Julia uses GitHub as a repository for its source distribution as well as for various Julia packages. We will look at installing on CentOS, which is the community edition of Red Hat and is widely used.

Installing on CentOS

CentOS can be downloaded as an ISO image from http://www.centos.org and written to a DVD. It can be installed as a replacement for an existing Windows system or to run alongside Windows as a dual-booted configuration.

CentOS does not come with the git command as a standard; upon installation, the first task will be to install it. For this and other installation processes, we use the yum command (Yellowdog Updater and Modified (YUM)).

You will need the root/superuser privileges, so typically, you would type su -:

    su -
    (type root password)
    yum update
    yum install git

Yum will fetch the Git sources for a Red Hat repository, list what needs to be installed, and prompt you to press y/n to continue.
Once you have installed Git, we will need to grab the Julia sources from GitHub by using the following command:

```bash
git clone git://Github.com/JuliaLang/julia.git
```

(It is also possible to use `https://` instead of `git://`, if behind a firewall).

```bash
Cloning into 'julia'...
remote: Counting objects: 97173, done.
remote: Compressing objects: 100% (24020/24020)
```

This will produce a subfolder at the current location called `julia` with all the sources and documentation.

To build, Julia requires development tools that are not normally present in a standard CentOS distribution, particularly GCC, g++, and gfortran.

These can be installed as follows:

```bash
sudo yum install gcc
sudo yum install gcc-c++
sudo yum install gcc-gfortran
```

Or, more conveniently, via a group install bundle:

```bash
sudo yum groupinstall 'Development tools'
```

Other tools (which are usually present) such as GNU Make, Perl, and patch are needed, but `groupinstall` should take care of these too if not present. We did find that an installation on Fedora 19 failed because the M4 processor was not found, but again, `yum install m4` was all that was required and the process could be resumed from where it failed.

So, to proceed with the build, we change into the cloned `julia` folder and issue the `make` command. Note that for seasoned Linux open source builders, there is no need for a configuration step. All the prerequisites are assumed to be in place (or else the `make` fails), and the executable is created in the `julia` folder, so there is no `make install` step.

The build process can take considerable time and produces a number of warnings on individual source files but when it has finished, it produces a file called `julia` in the build folder. This is a symbolic link to the actual executable in the `usr/bin` folder.
The Julia Environment

So, typically, if all the tools are in place, the process may look like this:

[malcolm@localhost] cd ~
[malcolm@localhost] mkdir Build
[malcolm@localhost] cd Build
[malcolm@localhost Build] git clone git://github.com/JuliaLang/julia.git
[malcolm@localhost julia] cd julia
[malcolm@localhost julia] make

After the build:
[malcolm@localhost julia] ls -l julia
lrwxrwxrwx 1 malcolm malcolm 39 Jun 10 09:11 julia -> /home/malcolm/Build/julia/usr/bin/julia

If you have (or create) a bin folder just under the home folder, it is worth recreating the link there as it will be automatically appended to the path.

[malcolm@localhost] cd ~/bin
[malcolm@localhost bin] ln -s /home/malcolm/Build/julia/usr/bin/julia julia

To test out the installation (assuming julia is on your path), use the following command:

[malcolm@localhost] julia -q

The -q switch on the julia command represses the print of the Julia banner.

julia> println("I've just installed Julia")
I've just installed Julia

The julia> prompt indicates the normal command mode. It is worth noting that there are a couple of other modes, which can be used at the console help (?) and shell (;).

For example:

julia> ?print
?print
Base.print(x)
   Write (to the default output stream) a canonical (un-decorated) text representation of a value if there is one, otherwise call "show". The representation used by "print" includes minimal formatting and tries to avoid Julia-specific details.

julia> ;ls
We will be looking at an example of Julia code in the next section, but if you want to be a little more adventurous, try typing in the following at the julia> prompt:

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

1. This is a simple estimate of PI by generating pairs of random numbers distributed uniformly over a unit square \([0.0:1.0, 0.0:1.0]\). If the sum of the squares of the pairs of numbers is less than 1.0, then the point defined by the two numbers lies within the unit circle. The ratio of the sum of all such points to the total number of pairs will be in the region of one quarter PI.

2. The line \(\text{sumsq}(x,y) = x^2 + y^2\) is an example of an inline function definition. Of course, multiline definitions are possible and more common but the use of one-liners is very convenient. It is possible to define anonymous functions too, as we will see later.

3. Although Julia is strictly typed, a variables type is inferred from the assignment unless explicitly defined.

4. Constructs such as `for` loops and `if` statements are terminated with `end`, and there are no curly brackets `{}` or matching `endfor` or `endif`. Printing to the standard output can be done using the `println` call, which is a function and needs the brackets. `@printf` is an example of a macro that mimics the C-like `printf` function allowing us to format outputted values.

**Mac OS X and Windows**

It is possible to download the stable build and the nightly release on Mac OS X, Windows, and Ubuntu. So, building Julia from source is less important than under distributions that do not provide a distro. However, since Julia is open source, it is possible to get detailed instructions from the Julia language website https://Github.com/JuliaLang/julia/blob/master/README.md.
On Mac OS X, you need to use a 64-bit gfortran compiler to build Julia. This can be downloaded from HPC - Mac OS X on SourceForge [http://hpc.sourceforge.net](http://hpc.sourceforge.net). In order to work correctly, HPC gfortran requires HPC GCC to be installed as well. From OS X 10.7, Clang is now used by default to build Julia, and Xcode version 5 or later should be used. The minimum version of Clang needed to build Julia is v3.1.

Building under Windows is tricky and will not be covered here. It uses the [Minimalist GNU for Windows (MinGW)](https://www.mingw.org) distribution, and there are many caveats. If you wish to try it out, there is a comprehensive guide on the Julia site.

### Exploring the source stack

Let's look at the top-level folders in the source tree that we get from GitHub:

<table>
<thead>
<tr>
<th>Folder</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>Contains the Julia sources that make up the core</td>
</tr>
<tr>
<td>contrib</td>
<td>A miscellaneous set of scripts, configuration files, and so on</td>
</tr>
<tr>
<td>deps</td>
<td>Dependencies and patches</td>
</tr>
<tr>
<td>doc</td>
<td>The reStructuredText files to build the technical documentation</td>
</tr>
<tr>
<td>etc</td>
<td>The juliarc file</td>
</tr>
<tr>
<td>examples</td>
<td>A selection of examples of Julia coding</td>
</tr>
<tr>
<td>src</td>
<td>The C/C++, Lisp, and Scheme files to build the Julia kernel</td>
</tr>
<tr>
<td>test</td>
<td>A comprehensive test suite</td>
</tr>
<tr>
<td>ui</td>
<td>The source for the console REPL</td>
</tr>
</tbody>
</table>

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 scipy python-pandas 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 $x \cdot \exp(-0.1x) \cdot \cos(2.0x)$:

\begin{verbatim}
In [15]: using PyPlot;
   x = linspace(0.0,11.75*pi,1000); y = zeros(1000);
   [y[i] = x[i]*cos(2.0*pi[i])*exp(-0.1*pi[i]) for i=1:1000]
   plot(x,y)
\end{verbatim}

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:

```julia
function run_asian(N = 100000, PutCall = 'C';)
  # European Asian option.
  # Uses geometric or arithmetic average.
```

---

[16]
Chapter 1

# 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

# Arithmetic or 'G'eometric
Averaging = 'A';

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

println("Average stock price is ");
The Julia Environment

```julia
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 `ARG` 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 = lengthARGS
 if nArgs >= 2
     run_asian(ARG[1], ARG[2])
 elseif nArgs == 1
     run_asian(ARG[1])
 else
     run_asian()
 end
```
Julia variables are case-sensitive, so we must use \texttt{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 \texttt{PutCall} option.

\textbf{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 \texttt{ASCIIPlots} package. This is not dependent on any other package and works at the console prompt.

You can find it at \url{http://docs.julialang.org} and choose the "Available Packages" link. Then, click on \texttt{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: \texttt{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:

\begin{verbatim}
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;
\end{verbatim}
\[ dW = \text{randn}(T) \sqrt{dt} \]
\[
S[t] = S[t-1] \times (1 + (r - q - 0.5 \cdot v \cdot v) \cdot dt + v \cdot dW[t] + 0.5 \cdot v \cdot v \cdot dW[t] \cdot dW[t]) \text{ for } t=2:T 
\]
x = \text{linspace}(1,T);
scatterplot(x, S, sym='*');

```julia
using ASCIIPlots
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:

```julia
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 $\tau$ elements.
2. Compute stochastic variance $\delta W$ in a single vectorized statement.
3. Compute track $s$ by using "list comprehension."
4. Create array $x$ by using linespace 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:
The Julia Environment

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.

<table>
<thead>
<tr>
<th>Language</th>
<th>Timing (c = 1)</th>
<th>Asian option</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>1.0</td>
<td>1.681</td>
</tr>
<tr>
<td>Julia</td>
<td>1.41</td>
<td>1.680</td>
</tr>
<tr>
<td>Python (v3)</td>
<td>32.67</td>
<td>1.671</td>
</tr>
<tr>
<td>R</td>
<td>154.3</td>
<td>1.646</td>
</tr>
<tr>
<td>Octave</td>
<td>789.3</td>
<td>1.632</td>
</tr>
</tbody>
</table>

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:
The Julia Environment

- BinDeps                    0.2.14
- Cairo  0.2.13
- Color 0.2.10
- HTTPClient              0.1.0
- IniFile 0.2.2
- LibCURL             0.1.3
- LibExpat0.0.4
- 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 [JuliaStats](https://github.com/) group and a Google group called [https://groups.google.com/forum/#!forum/julia-stats](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 [JuliaOpt](https://github.com/) 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:


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 mainstream 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 in 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.
The Julia Environment

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.
Where to buy this book

You can buy Mastering Julia from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.