# Table of Contents

# 1
# The Julia Environment

In this chapter, we explore all you need to get started on Julia.

Julia can also be downloaded from a couple of sources and bundled with the Juno IDE. It can be run using Jupyter and this is available on the Internet via the `juliabox.org` website. Julia is a high-level, high-performance dynamic programming language for technical computing. It runs on Linux, OSX and Windows.

In the previous edition we were using ***Julia v0.3*** which is the stable version at the time. However many breaking changes have been made since then and so all the current code against the future ***v1.0***. All the code for individual chapters is available as Jupyter notebooks and in source form, which can be run on Juno or via the Julia REPL (read-evaluate-print-loop)

In this chapter, you will learn the following topics:

- Overview of Julia
- Why use Julia?
- Getting started
- A quick look at Julia
- Package management

## Overview of Julia

Julia was first released to the world in February 2012 after a couple of years of development at the Massachusetts Institute of Technology (MIT). This followed from a couple of years of development at MIT. In later 2015 a commercial arm called Julia Computing was setup to acquire funding and provide consultancy and (some) enterprise packages. Most of Julia remains freely available and we will be concentrating on those here.

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

The language is open source, so 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 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. We will also indicate where package sources are stored; this too is a great palce for reference material

Julia is often compared with programming languages such as Python, R and Matlab. It is important to realise that Python and R have been 'around' since the mid 1990's and Matlab since 1984. Since Matlab is proprietary (*® Mathworks*) there are a few clones, in particular GNU Octave which again dates from the same era as Python and R.

Just how far the language as come is a tribute to the original developers and the many enthusiastic ones who have followed.

Julia uses Github both for a repository for its source and also 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

Julia has no simple 'built-in' graphics,, however there are several different graphic packages providing great flexibility and I will be devoting a later chapter especially directed to the most important ones to date as well as the new Graphics API.

## Philosophy

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

All needed a "fast" compiled language in the armory like C or FORTAN as the current languages listed above are in 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.*

With the introduction of LLVM (low-level virtual machine) compilation, it became 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 the other scripting languages and so should be easy to learn for anyone familiar to Python, R and Matlab. Julia code looks very similar to Matlab. However it is not a Matlab clone: Matlab code will not run in Julia nor Julia code in Matlab. Also there are many important differences between the syntax of the two languages as we will see when progressing through this book.

Also, we should not be overly fixated in considering Julia as a challenger to Python and R. In fact, we will illustrate instances where the languages are used to complement each over. Certainly, Julia was not conceived as such; 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 1970's. It was only given prominence in 2001 by William S. Cleveland in an article : "Data Science: An Action Plan for Expanding the Technical Areas of the Field of Statistics".

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

## What is data science?

It is sometimes said that there are as many definitions of Data Science and there are Data Scientists.

One definition might be:

> *Data science is the study of the generalizable extraction of knowledge from data. It incorporates varying elements and builds on techniques and theories from many fields, including signal processing, mathematics, probability models, machine learning, statistical learning, computer programming, data engineering, pattern recognition and learning, visualization, uncertainty modelling, 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 principle cause in the growth in 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?

To show you as a budding data scientist that 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 is written in Java, made it convenient to program map-reduce algorithms in it or any language which runs on the JVM. This leads to an obscene amount of bloated boiler-plated 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 in beginning was considered as an alternative to Java, but on inspection, Julia made an excellent candidate in this category too.

## Comparison with other languages

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

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

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

| | Fortran | Julia | Python | R | Matlab | Octave | Mathematica | Javascript | Go |
|---|---|---|---|---|---|---|---|---|---|
| fib | 0.26 | 0.91 | 30.37 | 411.31 | 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 |

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

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 etc., and the operating system deployed. For each test the version of the software is provided plus any external packages or libraries, for example for the rand_mat test Python is using Numpy and also C, Fortran and Julia are using OpenBlas.

Julia provide a set of webpages specially for  checking on 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 that used in C, Matlab, Python etc. Indeed extra language examples are being added and you will find benchmarks to try in Scala and Lua too.

This table is useful in another respect too, as it lists all the major comparative languages to Julia; no real surprises here, except perhaps the range of execution times:

- **Python**: It 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 at least an order of magnitude faster than Python which is why enterprise Python code need to be rewritten and compile in C/C++ or Java.
- **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 straight-forward. 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. 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 there 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.

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

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

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

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

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

## Why is Julia Fast?

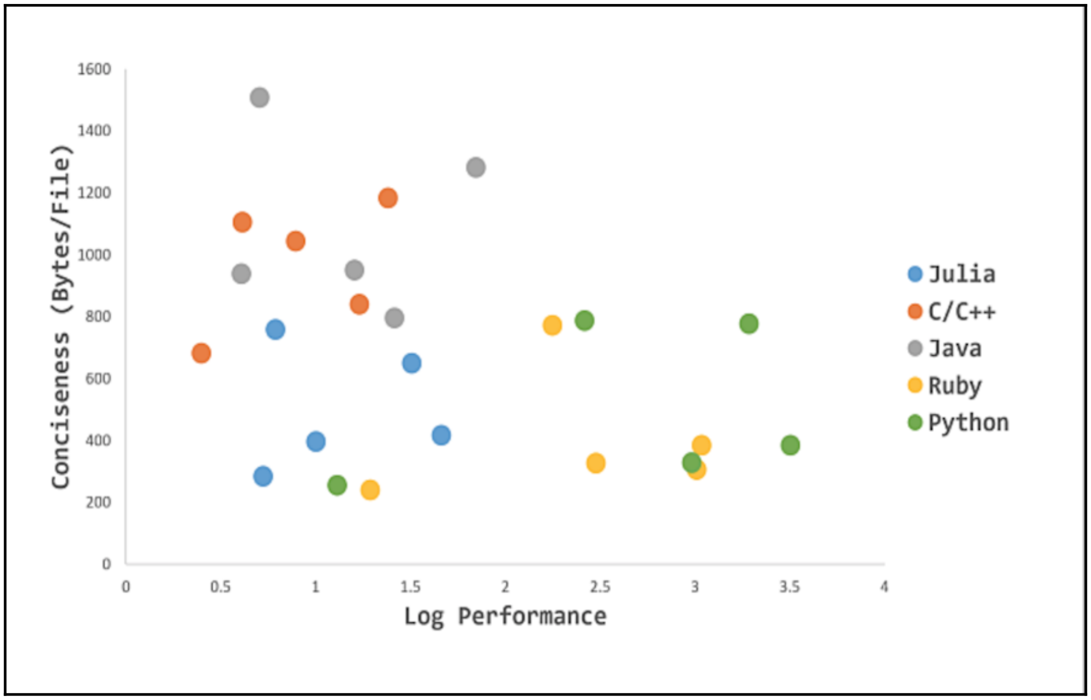
Julia's "big" idea is to compile the program right down to the machine code level.

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

Although this represents an overhead the code is cached, i.e. only compiled once and much effort has gone in to creating system images of the basic Julia system and caching individual packages. This makes execution times of the same "order" as c-code, perhaps about (x2).
C-compilers are often better optimised but LLVM is getting there quickly. So Julia provides the holy grail of compact code as well as fast execution times.

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

One of the great features of Julia is that as a scripted language it produces compact code which runs quickly, unlike Python, Ruby or R which are compact but slow or C and Java which are quick but verbose, as figure 1 shows.



*(In this figure that the performance times all are on a logarithmic scale)*

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

Since Julia was only designed in 2010 and it has been actively modified all the way up to version1.0, the advances in computing software and hardware since the 1990's, *when Python and R first were developed*, have been built in to Julia from it's design. Retrofitting these to existing language architectures has not always proved to be so easy.

## Why use Julia?

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

We have pointed out previously that Julia create executable code from scripts without a separate compilation step and this results in run times in the same order as those of C, Fortran, Java etc.; however in my opinion that is not the main reason to use Julia.

In this section we will look as the other factors with make it a *must-see* for any programmer, analyst and data scientist.

## Julia is easy to learn

Writing simple code in Julia will be almost immediate from anyone with a grounding in Python, R, C etc., as this book will show.

As mentioned previously the syntax is based on Matlab, where code blocks: for/while loops, if statements etc., are ALL terminated by "end".

There is no lining up of code (ala Python) or matching of brackets {} (ala R) and no distinction between if-endif, for-endfor, end-endwhile. The code is very close to the pseudo-code that you might write down to sketch out an algorithmic solution.

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

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

However porting from Matlab to native Julia code is usually quite straight-forward.

## Julia is written in Julia (mostly)

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

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

## Julia can interface with other languages

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

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

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

We will discuss interfacing in more detail in chapter 5.

## Julia has a novel type system

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

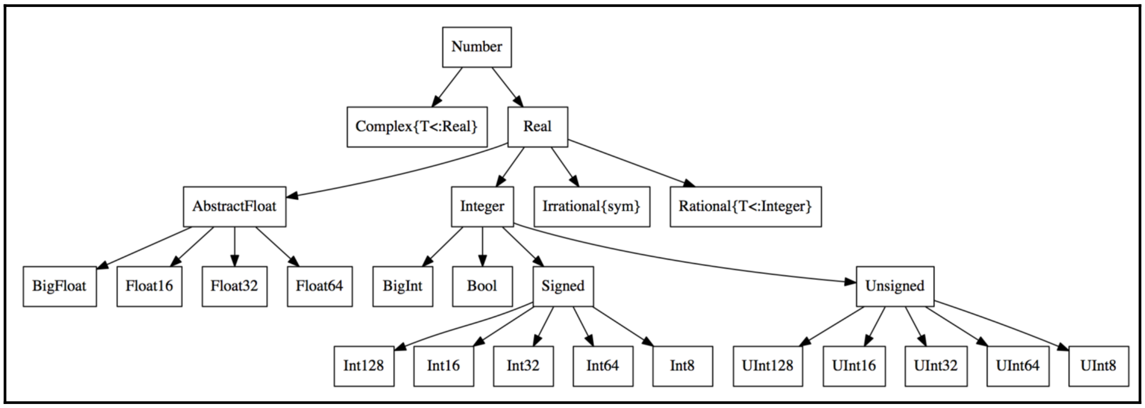


Figure 1.2 shows a subset of the type system corresponding to the hierarchy of numbers in Julia

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

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

## Julia has genuine runtime macros

Macros are defined via functions which are able to generate block code in a simple single line invocation. When a program is run, it evaluates the macro and the code produced is eventually evaluated like an ordinary expression.

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

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

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

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

We will look more closely at these in chapter 4 on *Multiple Dispatch and Metaprogramming*.

## Julia has a code level debugger

Earlier we discussed why Julia is fast. To recap, scripted code is processed (almost seamlessly) to low-level code via an intermediate representation. Recently, Julia has added a code level debugger (Gallium).

At the time of writing, Gallium is still experimental.

It can be accessed via the Julia command line (REPL) and also from the Juno IDE.

A code level debugger is pretty impressive, since it associates line by line source information with "compiled" machine code and allows step-by-step traversing through a script with the ability to inspect variable values at each step.

We will meet both Juno and Gallium in  chapter 12, *Going Further with Julia* of the book.

## Getting Started with Julia

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

We differentiate between several different sets of sources:

- Windows
- Apple OSX
- Generic Linux (x86)
- Generic Linux (ARM)
- Source (*necessary for other OS*)

The first place to look at is the main Julia community website: `http://julialang.org` and navigate to the`downloads` tab on the menu.

Windows and OSX are serviced by **exe** and **dmg** binaries respectively. In these cases, installation is as simple as downloading the binary and clicking on it, everything else is handled by the installer.

Linux systems were previously distributed for Redhat/Centos (**pkg**) and Debian/Ubuntu (**deb)** packages but now are just compiled for generic Linux systems and provide as a zipped archive; however, the overnight development system still provides pkg files.

Various binaries for ARM are available

Also a source-only archive, which can be used to build completely code.

It is also possible to get Julia and much more from the project's github site: `https://github.com/julialang`

It is worth noting that Julia has comprehensive documentation which can be found from the *docs* tab on `http://julialang.org` as well as links to the package manager, community and learning resources etc.

The main Julia site also provides a 'random' youtube presentation from previous JuliaCon meetings on its `home page`.

Once Julia is installed, it is necessary to add some additional modules using the package manager. In this book, we will introduce packages as they are needed. However, since the formation of Julia Computing, it is possible to go the site: `https://juliacomputing.com/` and download the JuliaPro product which is a bundled installer together with some 160+ of the most common packages. Because the packages are tested to work with the version of Julia bundled, the JuliaPro distribution lags somewhat behind the community source but remains free of charge and is a convenient way to get started. Some of the packages we will use may not be included in JuliaPro but these can be installed in the usual way.

Additionally, Julia Computing provide bundled versions with the Juno IDE, together with an extensive documentation and quick start guides. One-time registration is necessary, but this can be done from Google, LinkedIn of Github accounts.

Note, that when I downloaded the Windows source I came up against the Win10 security measures for unknown sources. Although, I changed the downloaded source's properties and it still refused to execute. In the end, I used the Windows Powershell to Unblock the source:

```
Windows Powershell
Copyright © 2016 Microsoft Corporation
PS C:\Users\Malcolm> cd Downloads
PS C: \Users\Malcolm\Downloads> gci julia-0.7-win64.exe | Unblock-File
–WhatIf
```

We will not be discussing build from source as this is not longer needed to get up and running . For those interested, the subsection of the Julia github project specifically dealing with Julia itself give comprehensive documentation via its markup page:

`https://github.com/JuliaLang/julia#source-download-and-compilation`

Also, this page deals with uninstalling Julia, which is as simple as deleting the source and the package specific (hidden) directory.

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

## A first Julia script

We will be looking at an example of Julia code in the next section but if you want to be a little more adventurous if you have installed Julia, start the command line version (REPL) try typing in the following at the julia> prompt:

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

Our first script computes a simple estimate of PI by generating pairs of random numbers distributed uniformly over 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 the such points to the total number of pairs will in the region of one quarter PI.

1. The line *`sumsq(x,y) = x^2 + y^2`* is an example of an inline function definition. Of course multiline definitions are possible and more common but to be able to do one-liners is very convenient. It is possible to define anonymous functions too as we will see later.
2. Although Julia is strictly typed a variables type is inferred from the assignment unless explicitly defined.
3. Constructs such as *for* loops and *if* statements are terminated with *end*, there are no curly brackets {} or matching *endfor* or *endif*.
4. Printing to standard output can be done using the `println` call which is a function and needs the brackets. *`@printf`* is an example of a macro which mimics the C-like printf function allowing us to format outputted values
5. As of v1.0 the *`@printf`* macros has been moved out of Base and into a separate package, so we need to include a `using Printf` at least once in the code.
6. In v1.0 there are new scoping rules which disbar top-level variables in the REPL

from being visible inside loops, although they are visible in begin/end and if/else/end statements; I will deal with these in the next section

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

```
@time for i = 1:N
  if sumsq(rand(), rand()) < 1.0
    global x += 1;
  end
end

0.175244 seconds (3.78 M allocations: 73.008 MiB, 5.80% gc time)
```

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

## Scoping Rules

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

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

One trick, which I'll discuss next, is to run this via v0.7; recall that v0.7 was designated as a beta for v1.0, where fatal errors in the latter would be deprecation warnings (and run successfully) - the warning being much more helpful.

```
julia> k = 0;
julia> for i = 1:10 k+= i end
┌ Warning: Deprecated syntax `implicit assignment to global variable `k``.
│ Use `global k` instead.
└ @ none:0
```

But why these rules, the syntax runs so they are more philosophical and syntactical, here is the official reasons as per the core development team:

*v0.7 correctly gives a warning that global k is needed to access k in the loop.*

*This is a slight inconvenience in the REPL, but is well worth it for programming in* *general since the scope rules are now much simpler and prefer making variables loop-local, which is better for multiple reasons.*

*One example we run into a lot is that test suites tend to use global variables. Then somebody adds a loop somewhere with a variable intended to be local, but ends up modifying global state.*

*I also think the new version makes it bit easier to explain that the global version is slower, since it's clear that you're updating a global variable*

However the elements of an top-level array are visible inside the loop, as also are field elements of a mutable structure *(see later)* ;  it seems just to be the humble scalar which has attracted this attention - Python programmers beware!

```
julia> kk = [0];
julia> for i = 1:10  kk[1] += i  end;
julia> println(kk[1])
55
```

**Making sense of v1.0 error messages**

Sometimes for veteran Julia programmers it seems that in v1.0 everything has been changed, and often purpose is not all that apparent.; to make the matter worse v1.0 is not particularly helpful in identifying the reason.

For example in v1.0

*julia> contains("Fred","/")*
*ERROR: UndefVarError: contains not defined*
*Stacktrace:*
*[1] top-level scope at none:0*

Fortunately v0.7 provides deprecation warnings and these can be very helpful in resolving the cause:

*julia> contains("Fred", "/")*
*⍰ Warning: `contains(haystack, needle)` is deprecated, use `occursin(needle, haystack)` instead.*
*⍰ caller = top-level scope at none:0*
*⍰ @ Core none:0*
*false*

Of course with Julia if you really prefer the previous syntax then you can always write:

*julia> contains(s::String,t::String) = occursin(t,s)*
*contains (generic function with 1 method)*
*julia> contains("Fred","/")*
*false*

As I'm writing this it seems that since the release of v1.0, IainNZ raised the issue #2878 of: `Global variable scope rules lead to unintuitive behavior at the REPL/notebook` and this has attracted an active discussion. Steven Johnson (of PyCall, IJulia etc.), apparently not a fan of this change, has released a package called *SoftGlobalScope.jl*, which goes someway to bypassing these rules. The fact remains that this is a measure which as IainNZ states: this leads to unintuitive behaviour in the REPL. It may well have changed when reading this book but again it may not have!

## Exploring the source stack

Before we look at some more complex examples of code, let’s look at the source as available from github or by unzipping the source distribution:

| directory | contents |
|---|---|
| base | contains the Julia sources which make up the core |
| contrib | miscelleneous set of scripts, configuration files etc. |
| deps | dependences and patches |
| doc | reStructuredText files to build the technical documentation |
| etc | juliarc file |
| examples | selection of examples of Julia coding |
| src | C/C++, Lisp and Scheme files to build the Julia kernel |
| stdlib | Standard library routines |
| test | Comprehensive test suite |
| ui | Source for the console REPL |

To gain some insight into Julia coding the best directories to look at are `base, examples` *`and test.`*

- Base contains a great portion of the standard library and the coding style exemplary.
- However in v1.0, a lot of the earlier routines in Base have been moved back in the

  Stdlib and so some of the routines which available previously now required a module to be reference. We have seen one such instance about where to use the `@printf` macro it is first necessary to have a `using Printf` statement.
- Test has some code which illustrates writing test scripts and using Base.Test system.
- Examples give Julia's take to some well-known computing old-chestnuts such as the Queens problem, Wordcounts and the Game of Life.

If you have created Julia from source, you will have all the directories available in the *git/build* directory; the build process creates a new directory tree in the directory starting with `usr` and the executable is in the *`usr/bin`* directory.

Installing on a Mac under OSX is more confusing; it creates Julia in the directory /Applications/Julia-[version].app where 'version' is the build number being installed. The executables required are in a subdirectory of this Contents/Resources/julia/bin. To find the Julia sources look into the share directory and go down one level in to the julia subdirectory.

So, the complete path will be similar to
`/Applications/julia-1.0.app/Contents/Resources/julia/share/julia`

This has the Julia files but not the C/C++, Scheme files etc., for that, you will need to view or checkout the source tree on Github, or download the source-only archive and unzip it.

For Windows the situation is the similar OSX. The installation file creates a folder called julia-[build-number] in the users AppData folder; usually this is a hidden folder so the file manager option to view hidden files needs to be set since it contains a subfolder named .julia with the Julia scripts in it.

Immediately under it are the bin, share directories (among others) and the share folder, typically as: *`C:\Users\Malcolm\AppData\Local\Julia-1.0\share\julia`*

Different operating systems have various locations for the Julia stack, so to find the location it is possible to use a builtin variable Sys.BINDIR which points to the folder containing the Julia executable; hence the actual stack is one directory above this.

```
# So to see the contents of the stdlib (in v1.0) use the following code:
julia> cd(string(Sys.BINDIR,"/../share/julia/stdlib/v1.0"));
julia> ; ls
Base64          FileWatching      LinearAlgebra    Printf    Serialization
SuiteSparse
CRC32c          Future            Logging          Profile   SharedArrays
Test
Dates           InteractiveUtils  Markdown         REPL      Sockets
```

```
UUIDs
DelimitedFiles LibGit2          Mmap           Random    SparseArrays
Unicode
Distributed    Libdl            Pkg            SHA       Statistics
```

## Interactive Development Environments (IDEs)

Julia has a few IDE alternatives, rather than working with the REPL:

- Jupyter
- Juno
- VS Code

The sources accompanying this book are provided in source format (.jl), but also as Jupyter (aka IPython) notebooks (.ipynb) and we will look at it next.

Juno is distributed with JuliaPro and can be installed into a standard distribution. We will discuss Juno in the final chapter.

VS Code is a visual studio (free) development framework which provides a Julia language extension; we will not discuss VS Code in the book but the URL currently is:

`https://marketplace.visualstudio.com/items?itemName=julialang.language-julia`

### Jupyter

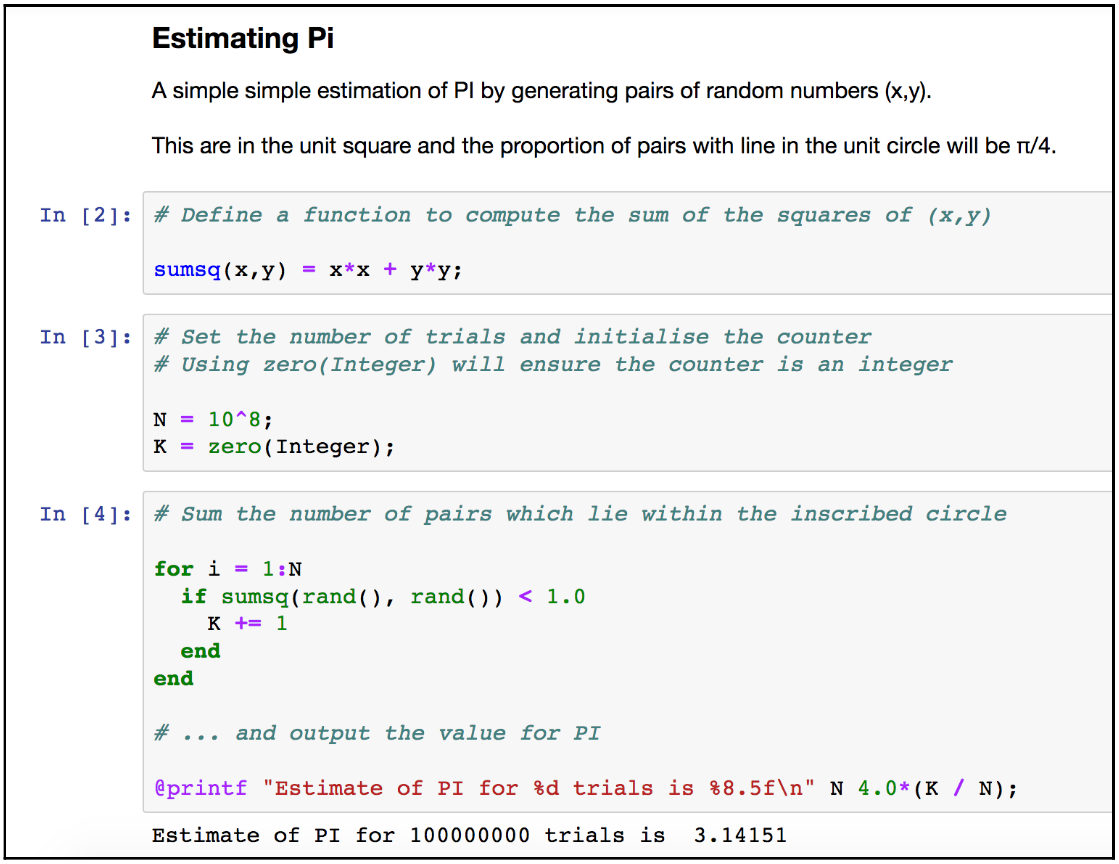
Jupyter is installed by default from the IJulia package using the Julia package manager from the REPL:

```
julia> Pkg.add(“IJulia”)
```

Note that :

- The first time `Pkg` is run, it will initialize a new repository; we will discuss the package manager in more detail at the end of this chapter.
- Adding the IJulia package will also add a number of other REQUIRED packages such as PyCall, PyPlot.

Figure 1.4 displays the earlier code to estimate PI, running in notebook for this chapter:



**Estimating Pi**

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

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

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

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

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

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

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

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

        # ... and output the value for PI

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

        Estimate of PI for 100000000 trials is  3.14151
```

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

```
julia> using IJulia; notebook()
```

The above comments will startup Jupyter in a local browser and on a well-known port (usually 8888).

If Python has been `installed from an Anaconda distribution`, then Jupyter also can be started separate from IJulia and will be aware of any Julia kernels .

In fact Jupyter will now run a large variety of kernels; in addition to Julia, Python and R, there are kernels for Perl, Lua, Clojure, Scala, Go and many more.

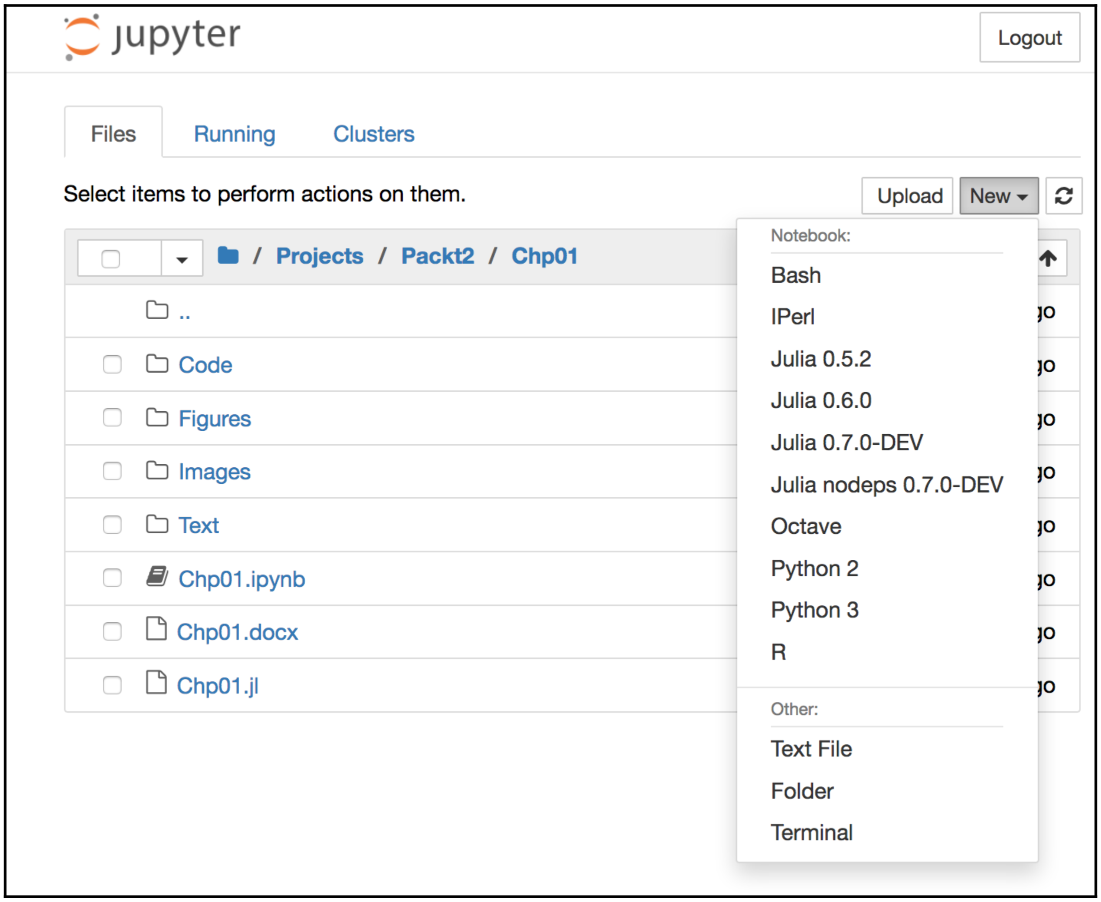
The Jupyter wiki provides the definitive list of Jupyter kernels:

`https://github.com/jupyter/jupyter/wiki/Jupyter-kernels`

The IDE starts in the default browser on `http://localhost`, normally on a port such as 8888 or 8889.

The first screen is a file directory listing and it may be necessary to traverse the folder tree to find the desired notebook. Any files with a *.ipynb* will be displayed regardless of which kernels they are running on.

It is also possible to open a new workbook and associate with any installed kernel:



For more information on IJulia or Jupyter the reader is referred to the following sources:

```
https://github.com/JuliaLang/IJulia.jl
https://github.com/jupyter
https://ipython.org/notebook.html
```

A quick look at some Julia

In the rest of this chapter, we will look at a few examples to get a feel for what Julia code looks like and how it works.

Some of the code included in the scripts may be covered in more detail later in this book.

However it should be possible to follow the listings without too much difficulty

## The Basel problem

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

To get an accurate listing, it is necessary to run this sources from the operating system, otherwise interacting with Jupyter will swamp the computation.  To this end I have included a command script in the code section accompanying this chapter (*runable in OSX and Linux*) to perform accurate timings in Julia and in addition in Python, R etc., assuming that that these have been previously installed and can be started from the execution path.

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

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

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

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

The sum of the series is approximately equal to 1.644934 …
Euler found the exact sum to be: $\pi^2/6$

He announced this discovery in 1735, but his arguments were based on manipulations that were not justified at the time, although he was later proven correct, and it was not until 1741 that he was able to produce a truly rigorous proof.

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

```
# Define the function to sum the series
function basel(N::Integer)
  @assert N > 0
  s = 0.0
  for i = 1:N
    s += 1.0/float(i)^2
  end
  return
end

basel(10^8)              # Evaluate it over 100,000,000 terms
1.644934057834575
```

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

```
Julia> /Users/Malcolm/PacktPub/Chp01/Code/basel.sh
using Python
Basel estimate is 1.64493396685
Number of terms: 10000000
Time taken was 2.83526992798 sec.
...
using R
[1] "BASEL estimate : " "1.64493396684726"
[1] "Number of terms in series: " "1e+07"
[1] "Time taken: "     "5.81213307380676"
...
using Octave
Number of terms is  10000000
Elapsed time is 30.0596 seconds.
Value of BASEL series = 1.644933
```

```
...
using Julia
0.048639 seconds (86 allocations: 6.498 KiB)
Basel estimate 1.64493397 over 10000000 terms
```

Julia takes around 50 msec compared with 2.8 sec for Python, 5.8 sec for R and 30 sec for Octave (*on my Mac Pro laptop*).

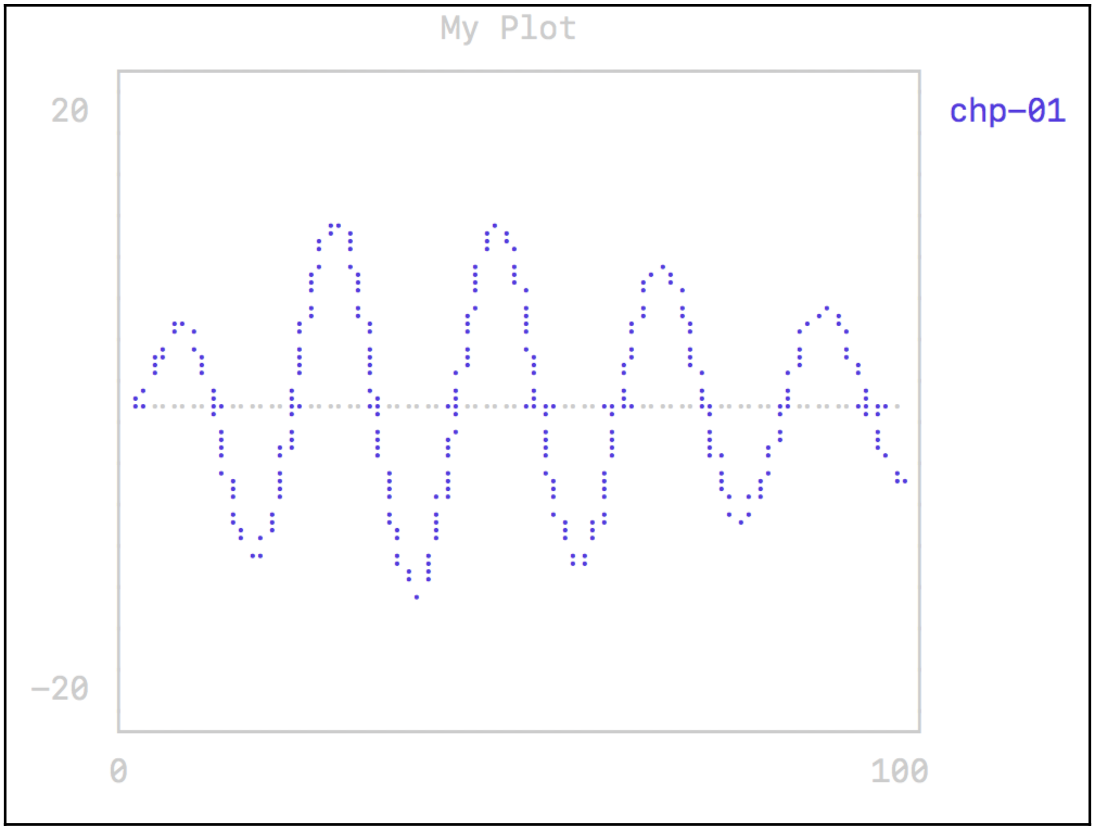
To produce  a more complete picture, it is useful to the package `BenchmarkTools` which runs a series of tests and outputs the median, mean, maximum and minimum timings.

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

## Displaying some inline graphics

```
Pkg.add("UnicodePlots"); # We need to add this on the first time.
using UnicodePlots

# Generate an array of the numbers from 1 to 100
# The ordinate value is create using a list comprehensive
x = collect(1:100);
y = [x[i]*sin(0.3*x[i])*exp(-0.03*x[i]) for i = 1:length(x)];
myPlot = lineplot(x, y, title = "My Plot", name = "chp-01")

# Alternatively this can be done using a map() construction
t = collect(0.0:0.1:10.0);
y =  map(x -> x*sin(3.0*x)*exp(-0.3*x), t);
```

The resulting graph is output to the REPL (or the notebook) as:



## Computing Geometric Brownian Trajectories

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

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

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

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

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

Here, I'm just going to look at the code required to produce some of these trajectories; t is relatively short to do this and needs no special features other than simple coding.

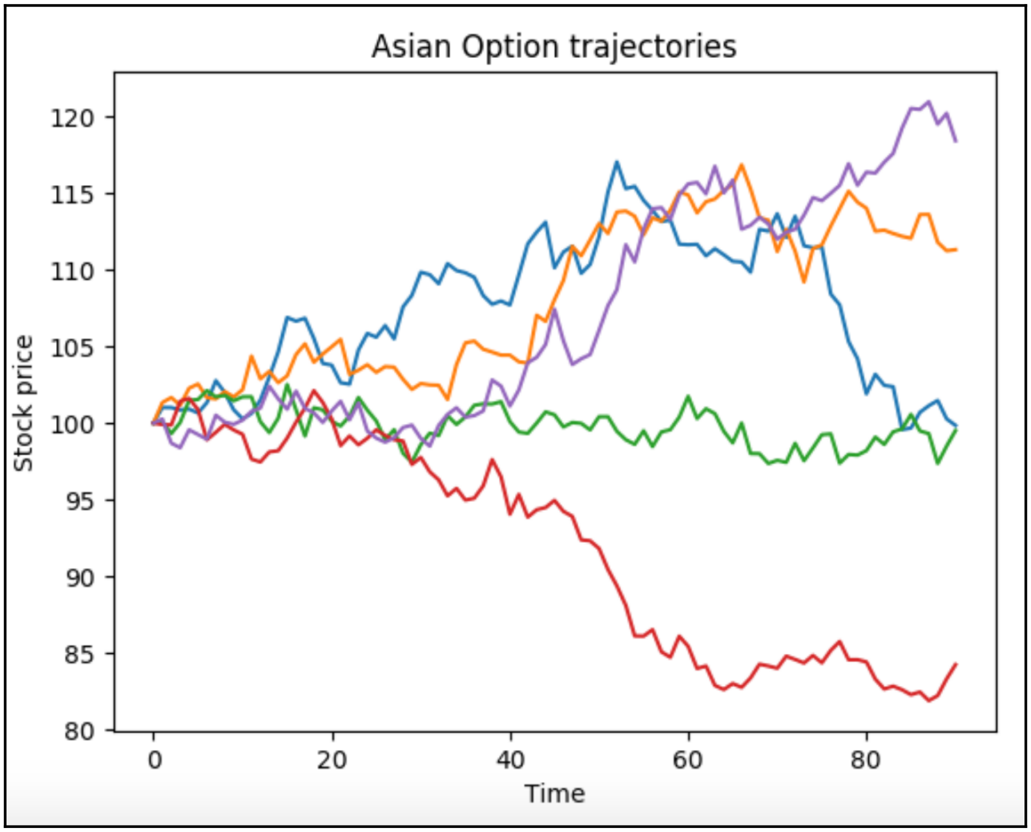
We will use PyPlot to display the graphics. This should be installed if you have previously added IJulia, otherwise add it with the package manager.

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

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

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

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

# Plot the first 5 trajectories
for k = 1:5
 S = zeros(Float64,N)
 S[1] = S0;
 dW = randn(N)*sqrt(dt);
 for t = 2:N
 z1 = (r - q - 0.5*v*v)*dt
 z2 = v*dW[t]
 z3 = 0.5*v*v*dW[t]*dW[t]
```

```
    S[t] = S[t-1] * (1 + z1 + z2 + z3)
    end
    plt.plot(x,S)
  end
```

The output is as follows:



## Package management

We have noted that Julia uses Git as a repository for itself and also for its package and that the installation has a built-in package manager, so there is no need to interface directly to Github. It is located in the Git folder of the installed system.

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

In v1.0 a new package manager has been introduced. Given the increase in registered packages, now approaching 2000, the previous version which relied on separate invocations of git became  very slow especially when using Windows

## Listing, adding and removing

Pkg3, which I'll now just term Pkg, has two modes of operating, using a command line shell and also though an API. The former is more common but some examples of the latteer can be seem in the code  accompanying each chapter.

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

This is similar to the used of `;` to drop into an operating system command line and `?` to enter the help system.

Pkg uses different means to maintain its package metadata and to track dependences and assess update requirements and utilises a number of folders in the `$HOME/.julia`

Repositories are in the *environmental/v1.o* and Pkg uses TOML (Tom's Obvious Minimal Language) format. The other main folder is *packages*, which keeps a separate user directory/folder to keep the local copies of the packages. Different from previous package managers, multiple copies of packages are maintained in a set of 5-alphanumeric hash subfolders; these are one for any top level package, i.e. added explicity, and the other when a package is dependency of another TLP.

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

- help: show this message
- status: summarize contents of and changes to environment
- add: add packages to project
- rm: remove packages from project or manifest
- up: update packages in manifest
- preview: previews a subsequent command without affecting the current state
- test: run tests for packages
- gc: garbage collect packages not used for a significant time
- init: initializes an environment in the current, or git base, directory

- build: run the build script for packages

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

Below is a typical Pkg session to install the BenchmarkTools *which have installed previously* and then remove it. The *init* command is not strictly required since an implicit initialisation will occur on the addition of the first package.

```
pkg> init
INFO: Initialized environment in /Users/malcolm by creating the file
Project.toml
pkg> status
INFO: Status "~/Project.toml"
pkg> update
INFO: Updating registry at /Users/malcolm/.julia/registries/Uncurated
INFO: Resolving package versions
INFO: Updating "~/Project.toml"
 [no changes]
INFO: Updating "~/Manifest.toml"
 [no changes]

pkg> add BenchmarkTools
INFO: Resolving package versions
INFO: Installed Nullables ───────── v0.0.3
INFO: Installed JSON ──────────────── v0.16.4
INFO: Installed BenchmarkTools ── v0.2.4
INFO: Updating "~/Project.toml"
[6e4b80f9] + BenchmarkTools v0.2.4
INFO: Updating "~/Manifest.toml"
[6e4b80f9] + BenchmarkTools v0.2.4
[34da2185] + Compat v0.49.0
[682c06a0] + JSON v0.16.4

pkg> status
INFO: Status "~/Project.toml"
[6e4b80f9] BenchmarkTools v0.2.4
[4d1e1d77] + Nullables v0.0.3

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

```
pkg> status
INFO: Status "~/Project.toml"

pkg> ^C
julia>
```

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

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

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

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

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

## Choosing and exploring packages

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

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

With the advent of Julia v1.0, it is to be hoped that great degree of package stability will be achieved. Also the commercial arm Julia Computing provide a (free) product JuliaPro which incorporates the Julia system, Juno editor and over 150 packages which are tested against the bundled product. This can be download from the Julia Computing website: `http://juliacomputing.com` along with a set of other useful material. We will meet

JuliaPro again in the last chapter when we discuss the Juno IDE and the debugger.

Naturally, releases of JuliaPro lag behind the 'latest' stable product from the community webste (`http://julialang.org`), but do offer a convenient way to get up and running without having to install a large set of modules separately. Many of the modules we use in this book are in the JuliaPro bundle, for the ones which are not it is, of course, possible to install them in the usual way.

How then should we select a package? 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. Also the principal author is likely to be delighted that someone else is finding the package useful and taking an interest in the work.

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

## 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 are adding that of SciPy and Pandas.

Statistics is well provided in Julia both on Github by the `https://github.com/JuliaStats` group, also by the groups site: `http://juliastats.github.io` and on Google groups using `https://groups.google.com/forum/#!forum/julia-stats`

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

The `Distributions.jl` package covers probability distributions and associated functions; also there is support for time series, cluster analysis, hypothesis testing, MCMC methods and more. JuliaStats is now incorporating machine learning and I am devoting a new chapter (chapter 11) to look at the work being done here.

Mathematical operations such as random number generators, exotic functions etc., 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.

## Graphics

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

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

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

An early module, and a favourite of mine is Winston. This  is a 2D graphics package which provides methods for curve plotting, 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 being shown on the screen.

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

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

We will look at all these later in the chapter 8, devoted entirely to graphics.

## Web and Networking

Distributed computing is well represented in Julia. TCP/IP sockets are implemented in the core. Additionally there is support for Curl and for SMTP and also for Websockets.

HTTP protocols and parsing are provided within a number of packages such as HTTP, HttpParser, HttpServer, JSON and Mustache.

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

The JuliaParallel group have provided a number of packages to implement support for parallel, multiprocessor and distributed processing and Julia Computing have created a product JuliaRun for batch running in enterprise applications. We will be discussing this work later in the book.

## Database packages

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

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

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

There is a package Mongo 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. However many of the NoSQL packages have received little attention recently and it may well be necessary to discuss other non-native methods using as Python libraries and REST.

## 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 OSX or the EXE file on Windows. Everything runs within this tree so there are no files installed to any 'system' directories.

In addition, we need to attend to the package directory. Recall that under Linux and OSX this is a hidden folder `.julia` in the users home directory. In Windows, it is located in the users profile typically in *`C:\Users\[my-user-name]`*. Removing this folder will erase all the packages that have been previous installed.

There is another hidden file which should be deleted *`.julia_history`* which keeps an historical track of the commands listed.

## Final thoughts

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

Julia has been maturing for nearly 6 years but with the advent of the v1.0 release, the formation of the commercial company Julia Computing and the distribution of bundled products, it has never been a better time to study and (hopefully) adopt Julia as a programming language of choice.

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

## Summary

This chapter introduced you to Julia, how to download it, install it and built 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 Julia via the command line (REPL) in order use a random walk method to evaluate the price of an Asian option. Also we discussed the use of two interactive development environments (IDEs), Juno and IJulia as an alternative to the REPL.

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

# Index