Table of Contents

1
Interoperability

Introduction

In this chapter I will discuss the cooperation between Julia and other languages, with the underlying operating system and also how Julia naturally encourages the use of parallel processing without the need for frameworks such as Hadoop or Spark.

The developers of Julia naturally focussed on calling from Julia and most of the discussions will be concerned with that, however handles were provided to go the other way and call Julia from C and hence incorporate it in foreign code, a brief overview of this is given later.

Julia has a rich and varied syntax but it was always seen that a vast wealth of code existed in object libraries covering a wide range of specialities. So it was seen that a simple mechanism to call functions from these libraries would prove to be useful and a one line native instruction call `ccall()` was added. This became a cornerstone of the way Julia operates, as we will see later.

C-style connectivity led to the development of packages which effectively wrap code around existing application programming interfaces (APIs). Some of the early successes were the interfacing to the Python executable and support packages which produced, among other things, the morphing of the Python IDE into Jupyter. Interacting with Python is described later in this chapter

Another example will be to access a database, one which exposes an API to perform function such a establishing a database connection, create and update records and to execute queries. We will see some cases of this in the chapter 8 on databases.

Calling non-native languages

By non-native languages I am referring to any programming language other than Julia.

For C and Fortran, a natural interface was created from the beginnings of Julia. For others, such as Java, Python and R, this interface has been used to create convenient means of using these languages .

What of others, such as Perl, Ruby etc.?

In fact it is possible to run these as "tasks", usually as part of a command pipeline and if necessary capture the result from the task and then post-process the output within Julia.

Spawning tasks and pipelining were introduced as concepts in Unix and operating systems such as OSX and Linux are especially well suited, whereas some of the code we encounter later may not execute on Windows.

C and Fortran

Julia as a single function **`ccall()`** which will run a routine compiled and bundled in to a shared object library. The call has the has the form:

```
ccall((fun_name,shared_lib), ret_type, (arg_type1, ...),
```

where the object code for the function **`fun_name`** is contained the shared library **`shared_lib`**

If the library name is omitted then it defaults to the default Julia library **`libjulia`** and in this case the first argument is specified as a string rather than a tuple.

If is necessary that Julia can 'find' the library, which usually needs adding it to a library on the load library path LD_LIBRARY_PATH or modifying the latter.

The second argument is the return type from the function, followed by a tuple of passed parameters, (this needs to be a tuple even if there is only a single parameters) and then these are followed by the actual values of the arguments.

As of version 1.0 ccall may have a function pointer as a first argument, such as one returned by dlsym.

Note that it is possible to create static libraries (typically with an extension .a) but most C (and Fortran) libraries are compiled into shared libraries and distributed as such.

On Unix/Linux systems these (usually) have the extension ".so", on OSX "dylib" and on Windows "dll". To assemble your own C code it is necessary to make it position independent by compiling with the –shared and –fPIC switches, which we will do later.

To clarify the ccall, we will start with the 'stock' example of accessing the system clock. The routine is part of the system library **`libc`** and has no input parameters, returning a **32-bit** integer . This is useful in setting the random number seed via a function in the package Random.

```
systime() = ccall((:clock,"libc"),Int32,());
randomize() = Random.seed!(systime());

julia> systime()
10590297
```

For a second example we will link to a *'mad'* , i.e. similar to the one we met in the previous chapter to calculate the expression (a*b + c).

This is again available form a routine in libc, but in here it is called fma, which takes 64-bit floats, and returns the same.

```
mad(a,b,c) = ccall((:fma,"libc"),Float64,
                   (Float64,Float64,Float64), a, b, c)

julia> mad(3.1,5.2,7.4)
23.520000000000003
```

The following code can be used to generate some integer random numbers, again 32-bit and so in the range 0:4294967295

```
julia> [ccall((:rand, "libc"), Int32, ()) for i=1:6]
 6-element Array{Int32,1}:
 1000393465
 963493892
 1415144464
 951615923
 1498098652
 1445766736
```

Mapping C types

Julia has corresponding types for all C types, such as Clong, Double, Cstring, refer to the Julia `online documentation` for a full list.

One caveat is that the Char type in Julia is 32-bit so C characters must be passed as bytes

(i.e. UInt8).

Since the ccall requires a tuple of argument types Julia is able to convert any passed variables implicitly rather than the programmer needing to do so explicitly Julia automatically inserts calls to the convert function to convert each argument to the specified type so it is not necessary for the programmer to explicitly make a call to library function.

Calling FORTRAN routines

FORTRAN code when compiled creates exactly equivalent object code to that written in C, so there is no difference between object libraries in either language; it is possible, but not common, to mix the two in a single library.

There is a wealth of FORTRAN routines existing, especially in the fields of scientific programming. From the beginning, Julia has utilised these, rather than implementing the same natively.

The only thing we need to be aware of in expressing the call is that FORTRAN passes scalar arguments by reference not by value, i.e. as the addresses of the parameters.

Julia passes arrays by reference, so we do not need to be careful here.

As an example we will call a FORTRAN routine from LAPACK **`ddot`** to compute the dot product between two real (double) arrays.

```
function compute_dot(DX::Vector{Float64},
                     DY::Vector{Float64} )
 @assert length(DX) == length(DY)
 n = length(DX)
 incx = incy = 1
 dotprod = ccall((:ddot, "libLAPACK"),
 Float64,
 (Ptr{Int64}, Ptr{Float64}, Ptr{Int64},
 Ptr{Float64}, Ptr{Int64}),
 Ref(n), DX, Ref(incx), DY, Ref(incy))
 return dotprod
 end
```

The ddot routine requires two arrays (DX and DY), the size of the arrays (N) and the increment to step through each array (incrx, incry), normally 1 but it is possible to step through by different increment(s), in which case N is interpreted as the number of steps and the arrays must be sufficiently large so the computation does not exceed the array bound(s).

We can see that the values of N, incrx, incry are computed and passed using Ptr(), the

arrays DX and DY are passed as is.

Below is a test for a couple of arrays of numbers of size 1000 and filled with random numbers in the range 0.0:1.0; so the dot product should be close to 1000*(0.5^2), i.e.: 250

```
# Test it out
 julia> aa = [rand() for i = 1:1000]
 julia> bb = [rand() for i = 1:1000]
 julia> compute_dot(aa,bb) # => Should be about 250
 261.5614635466589
```

Setting the LOAD PATH

Not all calls have numeric arguments in the form of scalars and arrays. We may wish to pass strings or more complex structures. For the latter it is common to emulate the structure via the Julia type system, as a mutable struct.

For a simple example we will return the user's home directory from the standard library. This is set by the operating system as the environment variable *"HOME"* and the **`getenv`** returns this.

Environment variables are so called that Julia creates a dictionary ENV at startup

So the home directory can be obtained from ENV["HOME"]

As we remarked earlier characters, (and so strings), need to be passed as a reference to an unsigned byte.

The library routine , likewise, returns a byte pointer, this is converted to a string using the routine **`unsafe_string`**, *unsafe* since we are not certain that the pointer refers to a C-String

```
julia> ptr = ccall((:getenv, "libc"), Ptr{UInt8},(Ptr{UInt8},),"HOME")
 Ptr{UInt8} @0x00007ffeec95666b
```

My modules are in the subdirectory "Julia/MyModules", relative to my home directory.

The following code is useful for put this on the LOAD_ PATH.

```
myHome = unsafe_string(ptr)
push!(LOAD_PATH, string(myHome,"/Julia/MyModules"))
4-element Array{String,1}:
 "@"
```

```
 "@v#.#"
 "@stdlib"
 "/Users/malcolm/Julia/MyModules"
```

In addition the ability to set and unset environment variables is useful while Julia is running; a routine is implemented in Base for the former but not the latter.

```
#=
The system library call has a third parameter, which when set to zero will
create a new variable but not overwrite an existing one.
=#
# Define the function to replace existing variable
evset(var::String, val::String) =
     ccall((:setenv,"libc"),Clong,
           (Cstring,Cstring,Clong),var,val,1);

# The unset routine is quite simple
evunset(evvar::String) =
     ccall((:unsetenv,"libc"),Clong,(Cstring,),evvar);
```

We can use this to set an environment variable to set to my PacktPub working directory tree, which is just below my user home directory.

```
# Set an environment variable PACKT_HOME ...
# ... and check it
packt = evset("PACKT_HOME",
            string(myHome,"/Users/malcolm/PacktPub"));

julia> ENV["PACKT_HOME"]
"/Users/malcolm/PacktPub"

# Now unset it, verify it is so.
julia> evunset("PACKT_HOME");
julia> ENV["PACKT_HOME"]
ERROR: KeyError: key "PACKT_HOME" not found
```

Basel and Horner functions in C

In previous chapters we looked a couple of examples of functions:

- **basel** : to compute the sum of $1/(x^2)$ to a given number of terms (N)
- **horner** : to evaluate a polynomial for a specific values and an array of coefficients

Below are the same functions written in C.

Although the book is about Julia and not C, the code in these cases is pretty straightforward

and the reader should be able to make sense of the procedures, even if not previously exposed to coding in C.

First the Basel function. We pass an integer from Julia, which will be a Clong (i.e. 64-bit) by default and return a 64-bit value for the sum, which will be a Cdouble.

```
//  Basel function in C
#include<stdio.h>
#include<stdlib.h>
double basel(int N) {
  double s = 0.0L;
  int i;
  double x;
  if (N < 1) return s;
  for (i = 1; i <= N; i++) {
    x = 1.0L/((double) i);
    s += x*x;
  }
  return s;
}
```

For the Horner routine, the scalar value *(x)* and the array of coefficients *(aa)* are all 64-bit, i.e. Cdouble's and the array size *(n)* Clong

```
// Link to a C routine
#include<math.h>
double horner(double x, double aa[], long n) {
  long i;
  double s = aa[n-1];
  if (n > 1) {
    for (i = n-2; i >= 0; i--) {
      s = s*x + aa[i];
    }
  }
  return s;
}
```

I have defined both routines first so that we can put them build a shared library which contains both.

Below is a command to create a library on OSX, it will be different on Linux and/or Windows. I have named the library as libmyfuns.dylib and so that it is found by the linker copied it to the standard folder to keep (local) libraries. Since this requires 'root' access it may be necessary to use a non-privileged directory and modify the LD_LIBRARY_PATH environment variable.

```
// Build a dynamic library (on OSX) as:
```

```
//
// clang -c basel.c horner.c
// libtool -dynamic basel.o horner.o
//          -o libmyfuns.dylib
//          -lSystem -macosx_version_min 10.13
//
// sudo cp libmyfuns.dylib /usr/local/lib
```

In the code accompanying this chapter there is the source for the two functions and a built library (for OSX).

The *'nm'* instruction can be used to list the entry points (globals) in the library:

```
run(`nm -g libmyfuns.dylib`)
0000000000000e90 T _basel
0000000000000f30 T _horner
U dyld_stub_binder
```

Now we can use **`ccall`** in our library, in a similar fashion as to any other library. This is quite a convenient fashion to create an interaction between Julia and C. There is no need for an complex API and the call reduces to a single native instruction, so also there is no additional overloading to the machine code.

Here is a run of the Basel function, first just for a values of 1000 terms and then to get a reasonable timing for 10^7. 100 terms is hardly sufficient as you may recall that the value of the sum is $(1.0 + \sqrt{5.0})/2.0 \Rightarrow \sim 1.618034$

```
# Run the Basel functioning 'ccall'
julia> basel = ccall((:basel,"libmyfuns"),
                     Float64, (Int64,), 1000)
 1.6349339668472596

# Time the function for 10^7 loops
 julia> @elapsed ccall((:basel,"libmyfuns"),Float64,(Int64,),10000000)
 0.191911357
```

The second function can be run to evaluate a polynomial using Horner's rule. As you might expect the same values as previously

```
# C version of Horners method (above)
# Note : use of Cdouble, Clong rather than Float64, Int64 etc.
#
julia> x = 2.1;
julia> aa = [1.0, 2.0, 3.0, 4.0, 5.0];
julia> ccall((:horner,"libmyfuns.dylib"),
```

```
        Cdouble,(Cdouble,Ptr{Cdouble},Clong),
        x,aa,length(aa))
152.71450000000002
```

Keeping it all in the family

Actually we can do a little better, using Julia's interface via the tasking interface. We will be exploring this towards the end of this chapter when I discuss using Perl, Ruby etc., to look at some examples of data munging and string processing.

However commands such as perl, ruby etc., are no different to Unix utilities such as grep, wc and (in this case) gcc. To use these in Windows, these utilities must be installed via a Posix compliant shell such as MinGW or Cygwin and the GNU compiler gcc has to be included.

To illustrate the procedure we will compute a value for PI in 'C' by the *usual* Monte Carlo method of generating pairs of random numbers and counting the times when their norm of under 1.0

In Julia we write the C code wrapped in a multiline string (delimited by """)

```
C_code = """
#include <stddef.h>
#include <stdlib.h>
double c_pi(long n) {
 long k = 0L;
 float rmax = (float) RAND_MAX;
 for (long i = 0L; i < n; i++) {
 float x = ((float) rand())/rmax;
 float y = ((float) rand())/rmax;
 if ((x*x + y*y) < 1.0) {
 k++;
 }
 }
 return 4.0*((double) k)/((double) n);
}
"""
```

Now we need a temporary name to create a library a shared library and we can spawn the compile command **`gcc`** to create it, contains the single entry point for our routine **`c_pi`**

```
const Clib = tempname() # ... make a temporary file
/var/folders/ns/9qt1mg2j7ldgnp5b1qt0fszr0000gn/T/juliala8K4T

# compile to a shared library by piping C_code to gcc
# (works only if you have gcc installed):
```

```
#
using Libdl
tmplib = string(Clib,".",dlext)
open(`gcc -fPIC -O3 -msse3 -xc -shared
                    -o $tmplib -`, "w") do
    fprint(f, C_code)
end
```

Now fro convenience we define a Julia function that calls the c_pi(), randomise the random seed with the first function we created in the chapter and run the function for 10^6 samples.

```
c_pi(N::Int64) = ccall(("c_pi",Clib),Float64,(Clong,),N)

using Random
randomize();

c_pi(1000000)
3.140868
```

Retrieving a webpage using Curl

Before we leave interfacing with C, a more complex example of retrieving a webpage using the library *libcurl* via the package LibCURL.jl, which can be installed in the usual way.

There are few other ways to do is, more contently, and we will meet then later:

1. Use the http.jl package maintained by JuliaWeb
2. Spawn the *curl* command as a task and capture its output.

```
# Specify a webpage from the London Julia User Group
# which shows the cover of this book

using LibCURL
url = "http://LondonJulia.org/mastering-julia.html";
```

We need to initialise a curl handle and set a couple of options to point to the URL and request to that it will follow webserver redirects

```
curl = curl_easy_init();
curl_easy_setopt(curl, CURLOPT_URL, url);
curl_easy_setopt(curl, CURLOPT_FOLLOWLOCATION, 1);
```

Now we need to setup a callback function to receive the data.

Note the use of the Csize_t identifier and passing the buffer addresses by reference. The actual retrieval is by use of a routine in libjulia: **`memcpy`** , because this is in the Julia library

it does not need to be specified in the call.

```
function curl_write_cb(curlbuf::Ptr{Nothing}, s::Csize_t,
                           n::Csize_t, p_ctxt::Ptr{Nothing})
  sz = s * n
  data = Array{UInt8}(undef,sz)
  ccall(:memcpy, Ptr{Nothing}, (Ptr{Nothing},
                Ptr{Nothing}, UInt64), data, curlbuf, sz)
  println(String(data))
  sz::Csize_t
end

c_curl_write_cb =
  @cfunction(curl_write_cb, Csize_t,
         (Ptr{Nothing}, Csize_t, Csize_t, Ptr{Nothing}));

 curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, c_curl_write_cb);
```

@cfunction is a macro to create a closure around the **`curl_write_cb`** function and we need to set an extra option to identify the write routine.

Running the **`curl`** function created by the first statement (and modified by all the **`curl_easy_setopt`** calls) returns an error status, success being denoted by 0; if the call fails then a `non-zero code` is returned.

```
julia> res = curl_easy_perform(curl);
<!DOCTYPE html>
<html><head><title>Mastering Julia Cover</title>
<style type="text/css">
h2 { font-family: verdana; font-size: 150%;}
p  { font-family: courier; font-size: 120%; }
</style>
</head>
<body>
<h2>Mastering Julia</h2>
<p><i>Malcolm Sherrington (2015)</i></p>
<p><img src="images/mastering-julia.jpg"
        alt="Mastering Julia" /></p>
<p> </p></body></html>
```

This is not the HTTP response status and we can get the **`curl_easy_getinfo`** call; these are standard HTTP return codes (viz: 200 => OK)

```
julia> println("curl url exec response : ", res);
curl url exec response : 0

julia> http_code = Array{Clong}(undef,1)
```

```
julia> curl_easy_getinfo(curl,
                         CURLINFO_RESPONSE_CODE,
                         http_code)
julia> println("httpcode : ", http_code)
httpcode : [200]

julia> curl_easy_cleanup(curl)
```

Embedding Julia in C

The Julia programming language is written in Julia, but not all of it. For speed and (sometimes) of necessity, the Core is written in C and also in Lisp/Scheme

The boot.jl module (in Base) comments out many of the types and functions which fall into this class and will use ccall to get them from the API, which are some of the functions defined in libjulia.

In earlier versions of Julia the API was not well covered in the documentation but this has been addressed in version 1 (https://docs.julialang.org/en/v1/manual/embedding/) and for a full discussion the reader is referred to these.

Also some of the code in Base communicate between the the Julia routines and the API, and this code can be perused to view some (well written) examples.

The following, distributed in the code snippets as jltest.c, uses the API to an expression comprising a sine function multiplied by an exponent.

If we wish to call Julia functions and pass C-variables we need convert these to the Julia type. This is termed as ***boxing*** and the generic ***jl_value_t*** is used with the appropriate **jl_box** function for the C-type to be passed.

After calling the function(s) the values returned need to be unboxed using one of the **jl_unbox** routines

```
include <julia.h>
#include <stdio.h>
#include <math.h>

// Only define this once if in an executable ...
// (i.e. not in a shared library) ...
// ... and if we want the fastest code.
JULIA_DEFINE_FAST_TLS()

int main(int argc, char *argv[])
{
```

```
 /* required: setup the Julia context */
 jl_init();

/* run Julia commands */
 jl_function_t *fnc1 = jl_get_function(jl_base_module, "exp");
 jl_function_t *fnc2 = jl_get_function(jl_base_module, "sin");
 jl_value_t* arg1 = jl_box_float64(-0.3);
 jl_value_t* arg2 = jl_box_float64(3.0);
 jl_value_t* ret1 = jl_call1(fnc1, arg1);
 jl_value_t* ret2 = jl_call1(fnc2, arg2);

/* unbox and setup final result */
 double retD1 = jl_unbox_float64(ret1);
 double retD2 = jl_unbox_float64(ret2);
 double retD3 = retD1*retD2;
 printf("sin(3.0)*exp(-0.3) from Julia API: %e\n", retD3);
 fflush(stdout);

/* Allow Julia time to cleanup pending write requests etc. */
 jl_atexit_hook(0);
 return 0;
}
```

How to build it?

- The compiler needs to know where the include file julia.h, and other related .h files, are located and then to pickup the libjulia library and include additional switches to create position independent code.
- The command to does this can be very complex and depends on the operating system of the platform on which Julia is running. In the past formulating this as quite difficult and now the Julia provides a script ***julia-config.jl*** in the ***share*** directory of the standard distribution

```
# Can use a nice command script in 'share/julia/julia-config.jl
./julia-config.jl
usage: julia-config [--cflags | --ldflags |
                     --ldlibs | --allflags]
```

We can generate the appropriate compiler switches, loader switches and/or the loader libraries and additionally with the --allflags can get them in a single shot.

Note: creating separate switches for the compilation and link-loading may be useful if writing a Makefile.

Here as a run on my MacPro, running v1.0 on OSX:

```
julia> julia julia-config.jl --allflags
-std=gnu99 -
I'/Applications/Julia-1.0.app/Contents/Resources/julia/include/julia' \
-DJULIA_ENABLE_THREADING=1 -fPIC \
-L'/Applications/Julia-
        1.0.app/Contents/Resources/julia/lib' \
-Wl,-rpath,'/Applications/Julia-
        1.0.app/Contents/Resources/julia/lib' \
-Wl,-rpath,'/Applications/Julia-
         1.0.app/Contents/Resources/julia/lib/julia' \
-ljulia
```

This is verbose since the directories are fully qualified and the initial portion is relative the the location of $JULIA_HOME; so this can simply by defining where the Julia distribution tree is installed.

```
JULIA_HOME = \
/Applications/Julia-1.0.app/Contents/Resources/julia; export JULIA_HOME

ls $JULIA_HOME
LICENSE.md bin etc include lib share
```

Armed with this we can create an executable and run it.

```
# On OSX, cc and gcc are links to the actual
# comiler clang
#
cc jltest.c -o jltest -std=gnu99 \
-I$JULIA_HOME/include/julia \
-DJULIA_ENABLE_THREADING=1 -fPIC \
-L$JULIA_HOME/lib \
-Wl,-rpath,$JULIA_HOME/lib \
-Wl,-rpath,$JULIA_HOME/lib/julia \
-ljulia

./jltest
sin(3.0)*exp(-0.3) from Julia API: 1.045443e-01
```

Python, R and Java

A Julia/Python interface has been supported since the early version of Julia but to the work of Steven G. Johnson, of MIT, on the PyCall module. Further this was used in creating the visualisation package PyPlot which is a wrapper around the Python package matplotlib.

We will seen this already in this book and will be discussing it in more details in the

chapter on Graphics. Also Steven Johnson wrote a kernel interface to the IPython IDE (viz. IJulia) and the work between the IPython and Julia teams lead to the new version of the former, Jupyter, in which the code examples accompanying this book have been distributed.

These modules are now supported by the community group JuliaPy and in addition there is a 'reverse' package, PyJulia, which permits Julia to be called from Python, I will describe that briefly below.

In addition there are some *wrapper* packages, notably those which implement the Pandas, ScikitLearn, Seaborn and SymPy modules . I will look at the last of these later in this chapter

Other languages are supported by an alternative community group JuliaInterOp. R and Java are up-to-date but others such as Matlab, Mathematica, C++ (i.e. Cxx), at the time of writing, have not been touched for a long time and are certainly not version 1.0 compliant.

JuliaInterOp also supports an interface to ZeroMQ, described as opinionated, light weight, blazing fast messaging library, which is often employed for language to language communications and is considered in chapter 11.

Python

As mentioned above the basis of the Python interface is the PyCall module. This package provides the ability to import arbitrary Python modules from Julia, call Python functions , define Python classes from Julia methods, and share large data structures between Julia and Python without copying them.

Also it provides with automatic conversion of types between Julia and Python, and the switching between the one and zero-based indexing of separate languages.

It can be called directly via the use of the **`@pyimport`** macro, which will import Python packages into Julia.

The following example creates a link to the Numpy package and uses the **`rand()`** routine, from the Python package and not a Julia one:

```
using PyCall
@pyimport numpy.random as nr
aa = nr.rand(4,5)
# aa is a Julia array generated by Numpy

4×5 Array{Float64,2}: 0.639591  0.654699  0.586075    0.235346   0.0352056
0.250739  0.793587  0.108476    0.836333   0.837293  0.469522  0.15307
```

```
0.980525    0.555424   0.569421  0.89431   0.947623  0.00407884  0.0755231
0.706829
```

The array 'aa' is now in Julia and we can create a slice, which has no overhead in Julia

```
julia> aa[2:3,2:3]
2×2 Array{Float64,2}:
0.793587   0.108476
0.15307    0.980525
```

Next lets us look at utilising another well known Python, SciPy.

SciPy is large with many sub packages for various disciplines to apply to scientific problems. As in the previous example @pyimport can be used to setup an alias to a sub-package and the routines contained in the sub-package run via the alias

The example which follows is part of the `optimize` package, and uses the routine to find the root of the function **`x*cos(x)`** over the interval [1,π]

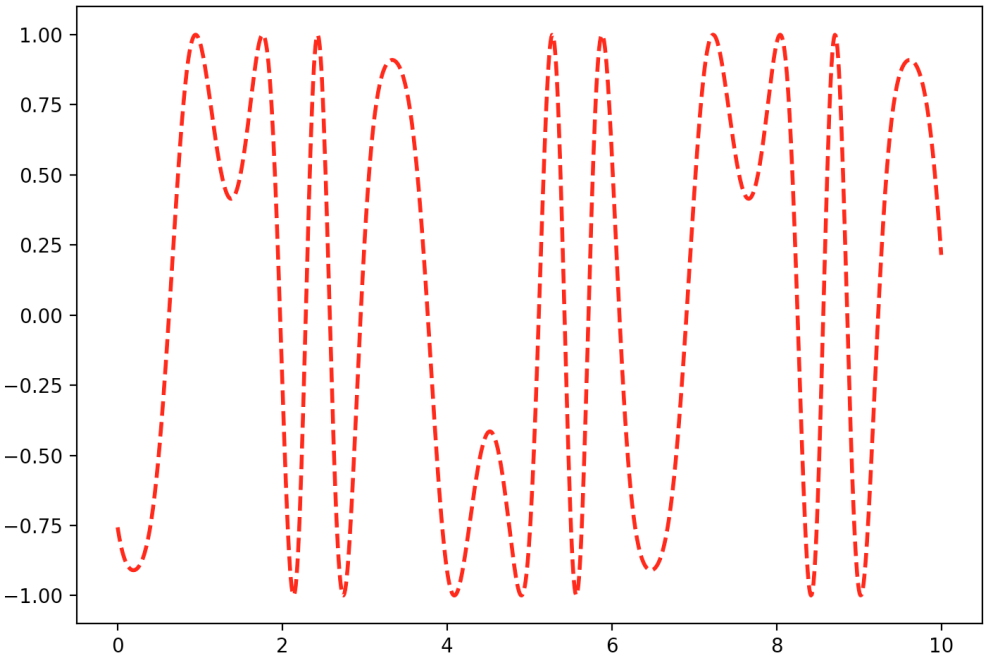
```
@pyimport scipy.optimize as so
so.ridder(x -> x*cos(x),1,π)
1.5707963267958964
```

A second example uses the integrate sub package, to find a quadrature (i.e. integral) for the function **`x*sin(x)`** over the same interval.

```
@pyimport scipy.integrate as si
si.quad(x -> x*sin(x),1,π)
(2.840423974650036, 3.153504096353772e-14)
```

Finally we note that it is possible to use *PyCall* directly to produce graphs but importing *matplotlib* rather than using the wrapper package *PyPlot*

```
@pyimport matplotlib.pyplot as plt
x = range(0,stop=10,length=1000);
y = sin.(3*x + 4*cos.(2*x));
plt.plot(x, y, color="red", linewidth=2.0, linestyle="--")
plt.show()
```



Going the other way

For the particular case of Python it is possible to access Julia. The package *pyjulia.jl* is maintained by the JuliaPy group.

A detailed discussion is outside the scope of this book but the following should be instructive for those who are interested.

1. Get the code from github
2. Check that python is installed
3. Setup the Julia module

```
$ git clone https://github.com/JuliaPy/pyjulia
$ which python
$ cd pyjulia
$ python setup.py install
```

This is all that is required, and we can now test out the Julia module.

The example I have provided is from what is known as the *low-level* interface.

```
macpro$ python
>>> from julia import Julia
>>> jl = Julia()
>>> pi = jl.pi # Pickup the Julia built constant PI
>>> jl.sin(0.25*pi)    # Now this works!
0.7071067811865475     # => 1.0/sqrt(2.0)
```

For more information and examples I direct the reader to look at the `webpage`. and in particular read about the high-level interface.

Packages with Python wrappings

As mentioned above, there are a number of packages based on PyCall which interface with Python packages rather than emulating them *natively*.

The advantage of this approach is that it is relatively easy to do and exposes a wealth of stable, powerful routines. The disadvantage is that the language (Python) and associated packages need to be present, and accessible.

As a rule it is best to use the Anaconda bistro for Python as this installs a large number of popular packages, including most of the ones needed for the Julia wrappers. Any additional packages can be added using the package manager: Conda

If you are using the Jupyter IDE to tackle the code in this book, than most likely you will have Anaconda installed already.

Apart from PyPlot, which as been mentioned there is are Pandas, Seaborn and SymPy packages. I am going to have a quick look at the last of these here.

SymPy works with a special type - the symbol (*Sym*). This is not to be confused with the Julia symbol (*Symbol*), which was introduced in the previous chapter on metaprogramming.

The following code defines a set of symbols; notice that it is possible to place restrictions on a symbol, such as only taking integer values, being positive and/or both.

```
using SymPy

u = symbols("u")
x = symbols("x", real=true)
y1, y2 = symbols("y1, y2", positive=true)
alpha = symbols("alpha", integer=true,
                         positive=true)
```

```
julia> typeof(x)
Sym
```

*Note: the type is **Sym** not **Symbol***

With one of the symbols already defined we can solve some algebraic equations, for example: $u^2 = -1$

```
solve(u^2 + 1)
[ -I
   I ]
```

When run in a Jupyter notebook, SymPy produces output in Latex and requires the Julia package ***LaTeXStrings*** to be installed, again another package due to Steven Johnson.

In the REPL SymPy does the best it can but the output is not as easy to read.

Here is a much more complex polynomial equation and its solution:

```
p = (x-3)^2*(x-2)*(x-1)*x*(x+1)*(x^2 + x + 1)
solve(p)
```

$$\begin{bmatrix} -1 \\ 0 \\ 1 \\ 2 \\ 3 \\ -\frac{1}{2} - \frac{\sqrt{3}i}{2} \\ -\frac{1}{2} + \frac{\sqrt{3}i}{2} \end{bmatrix}$$

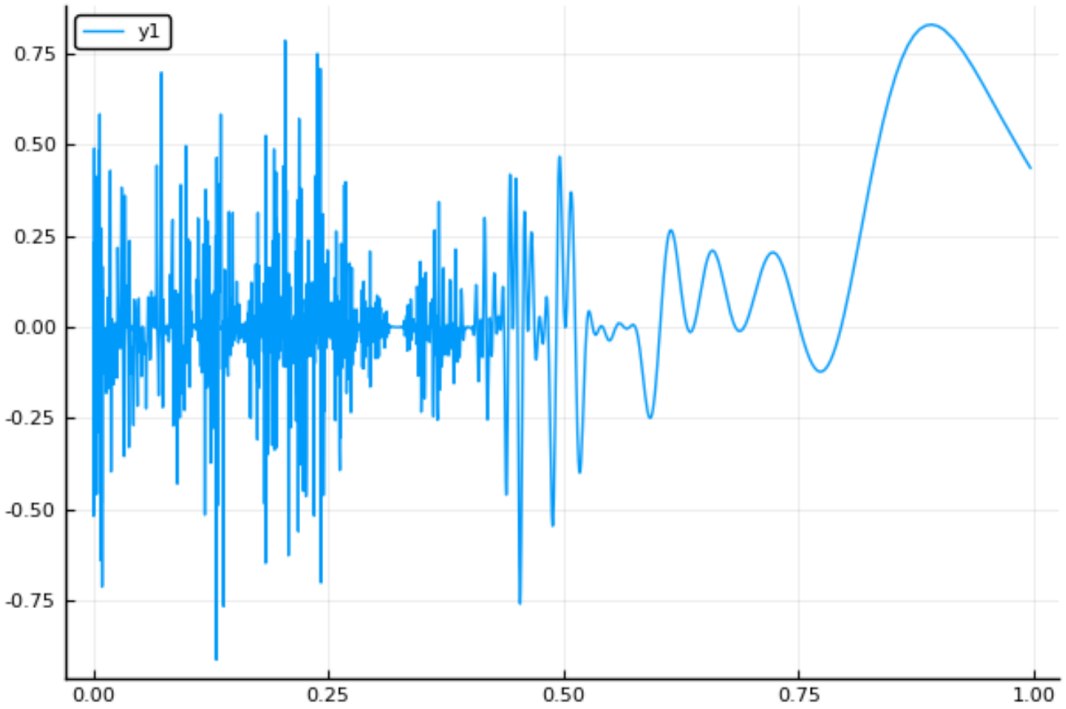
The next expression generates a product of a set of sine waves

```
x = symbols("x")
p = expand(prod([sin(x^(-i))
           for i in 1.0:1.0:5.0]))
```

```
sin(x^-5.0)sin(x^-4.0)sin(x^-3.0)sin(x^-2.0)sin(1/x)
```

. . . and this can be plotted for (say) : x ~ [0.0, 1.0]

```
using Plots
pyplot()
plot(p, 0.0, 1.0)
```



Another example which we have met previously is the sum of the so-called Basel series $(1/x^2)$ and we noted that Euler proved the value to be $\pi^2/6$

```
#=
Define Sym(bol)s for the loop index and the number of terms to computer the
summation.
SymPy identifies it as the harmonic series, with the power of 2
=#
julia> (i, n) = symbols("i, n")
julia> sn = Sum(1/i^2, (i, 1, n))
julia> doit(sn)
harmonic(n,2)

# Evaluate the limit from [1, Inf]
```

```
julia> limit(doit(sn), n, oo)
π^2/6
```

SymPy can handle calculus and solve equations containing derivatives; here is an initial valued problem:

```
y = SymFunction("y")
a, x = symbols("a,x")
eqn = y'(x) - 3*x*y(x) - 1
```

$$-3xy(x) + \frac{d}{dx}y(x) - 1$$

We will solve this equation from (x_0, y_0) in two stages, with $x_0=0$, and $y_0=a$

```
# First curry the equation with a parameter 'a' for the value of y₀
x0, y0 = 0, a
out = dsolve(eqn, x, (y, x0, y0))

# And the resolve this for a case a = 2.1
out |> subs(a,2.1)
```

$$y(x) = \left(\frac{\sqrt{6}\sqrt{\pi}}{6} \operatorname{erf}\left(\frac{\sqrt{6}x}{2} \right) + 2.1 \right) e^{\frac{3x^2}{2}}$$

R

In the previous edition interfacing with R was via a package called Rif.jl, which was quite cumbersome to use and necessitated building a versions of R as a shared executable.

Since then the introduction of RCall has greatly simplified and extended the usability of R when called from Julia

RCall provides multiple ways of interacting with R.

1. R REPL mode
2. @rput and @rget macros
3. R"" string macro
4. RCall API: reval, rcall, rcopy and robject etc.

5. @rlibrary and @rimport macros

R REPL mode

The R REPL mode allows real time switching between the Julia prompt and R prompt. After a 'using Call' statement, keying a '$' will switch to the R REPL mode and the R prompt will be shown.

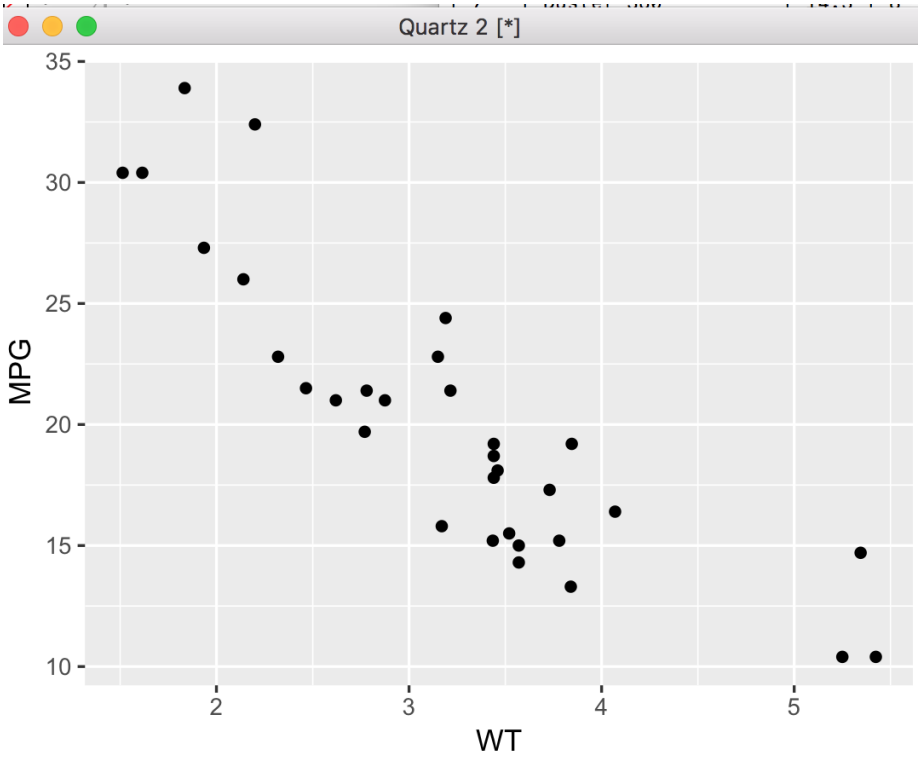
The backspace is used to leave R REPL mode, in the same way as for Pkg3

In the following snippet the *mtcars* dataset is loaded on the Julia side from RDatasets. Then switching to the R REPL, by typing a **$**, we can loading the ggplot2 library and create a graph from the dataset by referencing it as $mtcars.

The example shown is that of miles-per-gallon *vs.* car weight.

```
using RCall, RDatasets
julia> mtcars = dataset("datasets","mtcars")

R> library("ggplot2")
R> ggplot($mtcars, aes(x=WT,y=MPG)) +
                          geom_point()
```



Exchanging variables with @rget and @rput

Referencing a Julia variable on the R side can be done by using the **`@rput`** macro; this can be applied to arrays as well as scalar variables

```
julia> aa = randn(5);
julia> @rput aa;
R> aa
[1] 0.64077 0.30449 -0.67675 1.69559 0.00273
```

The same is true for other structures of @rput mtcars from the first example results in the following on the R side

```
R> mtcars
                  Model  MPG Cyl  Disp  HP DRat    WT  QSec VS AM Gear Carb
1             Mazda RX4 21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
2         Mazda RX4 Wag 21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
3            Datsun 710 22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
4        Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
5     Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
6               Valiant 18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
7            Duster 360 14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
8             Merc 240D 24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
9              Merc 230 22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
10             Merc 280 19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
11            Merc 280C 17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
12           Merc 450SE 16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
```

The alternate is possible too, using the **`@rget`** macro.

```
R> bb = c(1,2,3,4,5)
julia> @rget bb;

# Show as the transpose to save space.
# The c() list comes across as Floats not Ints
julia> bb'
1×5 RowVector{Float64,Array{Float64,1}}:
 1.0 2.0 3.0 4.0 5.0
```

Using the R"..." string macro

The R"..." string macro does not require flipping to R REPL mode, In the example below we generate a set of 1000 normally distributed variates and perform a Student's t-test. We expect the distribution to have a zero mean and unit variance, so the null hypothesis that $\mu \sim 0$ should be true.

```
julia> using RCall
julia> x = randn(1000);
julia> R"t.test($x)"

RObject{VecSxp}  One Sample t-testdata:  `#JL`$xt = -2.3245, df = 999, p-
value = 0.0203
Alternative hypothesis: true mean is not equal to 095 percent confidence
interval: -0.13405910 -0.01132606sample estimates:  mean of x -0.07269258
```

R has a number of ways of optimising functions, called from the **`optim`** routine and specifying which algorithm to use.

These are based on Nelder-Mead, quasi-Newton and conjugate-gradient algorithms the example below defines a non-linear function, applying one such the BFGS (*Broyden–Fletcher–Goldfarb–Shanno*) method.

```
julia> f(x) = 10*sin(0.3*x)*sin(1.3*x^2) +
             0.00001*x^4 + 0.2*x + 40;
julia> R"optim(0, $(x -> f(x)), method='BFGS')"
RObject{VecSxp}$par[1] -6.685958$value[1] 29.61426$countsfunction gradient
28        8 $convergence[1] 0$messageNULL
```

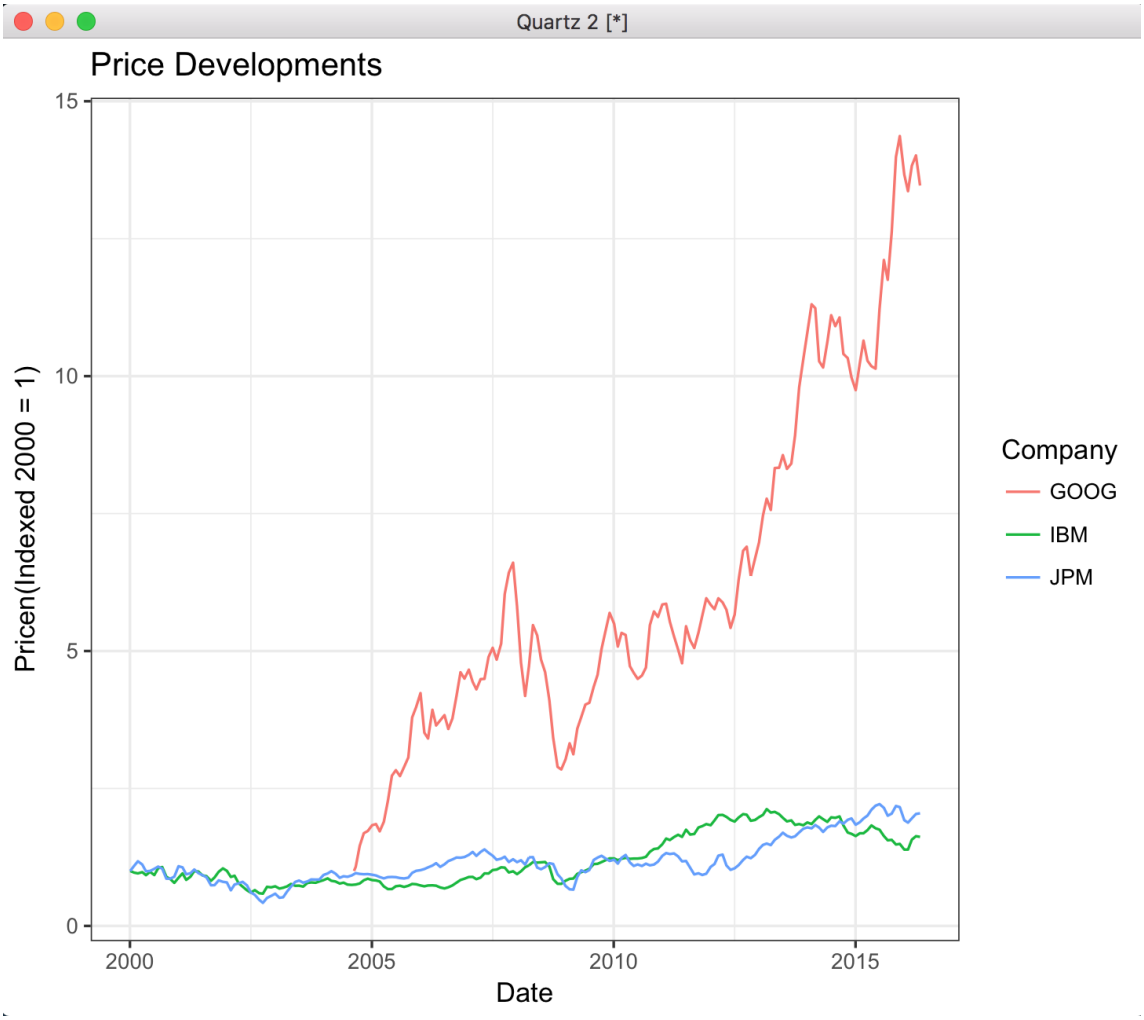
Finally in the section we will use R to read some financial data from the CSV file provided with the chapter's code, creating a data frame of price, date and ticker (i.e. stock code).

The prices for separate stocks are normalised against the first value.

```
R"""
library(data.table)
library(scales)
library(ggplot2)
link <- "fin_data.csv"
dt <- data.table(read.csv(link))
dt[, date := as.Date(date)]
# create indexed values
dt[, idx_price := price/price[1], by = ticker]
"""
```

... and display it using ggplot

```
R"""
ggplot(dt, aes(x=date, y=idx_price,
        color ticker)) + geom_line() +
           theme_bw() + xlab("Date") +
    ylab("Pricen(Indexed 2000 = 1)") +
    scale_color_discrete(name = "Company")
"""
```



The R API

First we use the *$* to get switch to R REPL mode and then generate an array of 100 random numbers in the interval [0.0, 1.0]

R> rr <- runif(100,0.0,1.0)

We could bring this over to Julia using the evaluate **`@rget`** macro.

Alternative we can use **`reval()`** the function to evaluate the array in the Julia context.

```
julia> ra = reval("rr")
RObject{RealSxp}
 [1] 0.069845643 0.807724489 0.812904286 0.372850300
 . . . . . .
 . . . . . .
 . . . . . .
 [97] 0.088333914 0.029721416 0.144311317 0.655515486
```

Notice that his is still an R-Object so have to apply R functions to it, not Julia, which we do with **`rcall()`** *passing the routine name as a symbol.*

```
julia> rcall(:mean,ra)
RObject{RealSxp}
[1] 0.4931235
```

Alternatively this can be copied to a Julia object with **`rcopy()`** , and now the Julia routine used.

```
# To use the mean in Julia we need to import the
# Statistics package
julia> using Statistics
julia> rb = rcopy(R"rr"); mean(rb)
0.49312349389307203
```

@rlibrary and @rimport macros

In this section we consider the use of the **`@rlibrary`** macro to create a reference to R packages. One popular one is MASS (`Modern Applied Statistics with S`) which contains a range of statistical routines and accompanying datasets.

The final 'S' refers to the fact the R is a clone of the commercial production S+

Here we will use **`@rlibrary`** to access MASS and then **`rcopy()`** to get some data about the *"Old Faithful"* geyser in Yellowstone Park. This is a simple dataframe with a couple of columns referring to the duration of and time between eruptions (in minutes) recorded on August 1st - August 15th, 1985.

```
@rlibrary MASS
geyser = rcopy(R"MASS::geyser")
```

	waiting	duration
	Float64	Float64
1	80.0	4.01667
2	71.0	2.15
3	57.0	4.0

	waiting	duration
	Float64	Float64
4	80.0	4.0
5	75.0	4.0
	.....	.....

This has been copied to a Julia dataframe and we can estimate the ratio of the time between eruptions to their duration (on the Julia-side)

```
round(sum(geyser[:waiting])/
      sum(geyser[:duration]),digits=5)
20.8952
```

Also we can use the R-summary routine to calculate statistics (on the R-side)

```
rcall(:summary, geyser[:waiting])
RObject{RealSxp}Min.   1st Q  Median  Mean   3rd Q   Max. 43.00  59.00
76.00   72.31  83.00   108.00

rcall(:summary, geyser[:duration])
RObject{RealSxp}
Min.   1st Q. Median Mean   3rd Q. Max.  0.833  2.000  4.000  3.461  4.383
5.450
```

*Note that BOTH datasets are were denoted as **geyser**, but are different.*

```
Import the base package as an alias
This exists in all R runtimes

julia> @rimport base as rbase
julia> ra
RObject{RealSxp}
 [1] 0.069845643 0.807724489 0.812904286 0.372850300
 . . . . . .
 . . . . . .
 . . . . . .
 [97] 0.088333914 0.029721416 0.144311317 0.655515486

julia> rbase.sum(ra)
RObject{RealSxp}
[1] 49.31235
```

R> library(help = "base")

```
julia> rbase.summary(ra)
RObject{RealSxp}
 Min.   1st Q  Median Mean   3rd Q  Max.
```

```
0.0051 0.2836 0.4761 0.4931 0.7503 0.9915
```

Java

The interface to Java is via the JavaCall package. The first call is to initialise the runtime; it is possible to passed additional switches to (say) specific the Java class path.

```
# Initialise and ask for additional memory using JavaCall
JavaCall.init(["-Xmx128M"])
```

Notes:

We can set the (say) the current directory to be the classpath in the JavaCall.init() routine by an expression such as:
["-Xmx128M -Djava.class.path=$(@__DIR__)"])

On OSX there may be a segmentation fault (-11) but this does not halt a Jupyter notebook or the REPL
See: `http://juliainterop.github.io/JavaCall.jl/faq.html`

In the REPL, Julia may need to be started with a *--handle-signals=no* option in order to disable Julia's signal handler.

NB: This may cause issues with handling ^C in Julia

- Static and instance methods with primitive or object arguments and return values are callable.
- One dimensional Array arguments and return values are also supported.
- Primitive, string, object or array arguments are converted as required.

The following snippet converts a string in Julia to a JString.

```
a = JString("Hello, Blue Eyes")
JavaObject{Symbol("java.lang.String")}
     (Ptr{Nothing} @0x00007fe4d3027c90)

# The JString as a single field (:ptr)
julia> fieldnames(typeof(a))
(:ptr,)

# The pointer can be used in a Java function
julia> b = ccall(
```

```
          JavaCall.jnifunc.GetStringUTFChars,
          Ptr{UInt8},
          (Ptr{JavaCall.JNIEnv},
          Ptr{Nothing}, Ptr{Nothing}),
          JavaCall.penv, a.ptr, C_NULL
        )

# Check that 'b' is the original string
julia> unsafe_string(b)
"Hello, Blue Eyes"
```

In a similar to fashion as with Python (PyCall), we can use the Java JVM (JavaCall) to perform some mathematics:

```
# Import the Java Math classes
jlm = @jimport "java.lang.Math"

# Calculate the function exp(π^2/6)
jcall(jlm,"exp",jdouble,(jdouble,),pi*pi/6.0)
5.180668317897116
```

.Another set of Java classes can be used to access the Internet

```
# Import the classes and set up a URL
jnu = @jimport java.net.URL
jurl = jnu((JString,), "http://LondonJulia.org/mastering-julia.html")

# Get the hostname from the full Url
julia> jcall(jurl, "getHost", JString,())
"LondonJulia.org"
```

Finally let's import the ArrayList class, define an instance of it and add some items.

```
JArrayList = @jimport(java.util.ArrayList)
a = JArrayList(())
jcall(a,"add",jboolean,(JObject,),"'Twas ")
jcall(a,"add",jboolean,(JObject,),
        "brillig, ")
jcall(a,"add",jboolean,(JObject,),"and ")
jcall(a,"add",jboolean,(JObject,),"the ")
jcall(a,"add",jboolean, (JObject,),
         "slithy ")
jcall(a,"add",jboolean,(JObject,),
        "toves,")

# Now iterate thru' the array and push
# it on a Julia array converting the bit
# type to an (unsafe) string.
```

```
t = Array{Any, 1}()
for i in JavaCall.iterator(a)
   push!(t, unsafe_string(i))
end

# Evaluate the result.
join(t)
"'Twas brillig, and the slithy toves,"
```

Working with the O/S and Pipelines

Upto now we have been discussing ways of Julia interoperating with other programming languages.

For C (and Fortran) this was relatively straight-forward as Julia was designed with a mechanism to interface directly with shared library (aka *DDLs* on Windows) . So any system which effectively operate very a shared library is immediately available to Julia.

This means that graphics frameworks, database management systems, etc., can all be made (more or less) easily accessible and a number of very successful packages have been implemented, those code depends on (and requires the installation of) third party libraries. These have been termed 'wrapper' packages, as opposed to 'native' ones, written purely in Julia. In practice many packages are often a combination of both paradigms rather than being principally one or the other.

For some other languages, notably Python, R and Java, effort was expended on creating usable interface packages and, in the case of Python, these have lead to additional packages wrapping around some of Python's better known ones.At present the list, which included Matlab, Mathematica and C++, has been reduced but perhaps that work will be taken up again in the future.

All these 'old' packages are still listed as within the province of the JuliaInterop group.

Work continues on the CxxWrap.jl package, as a wrapping for C++ types and functions, but this is outside the scope of this chapter.

For other languages such as Perl, Ruby etc., an alternative approach needs to be employed. We have seen that Julia is able to communicate with utilities in the underlying operating system. In fact languages which have the ability to execute from a command line are no different from 'standard' utilities and can be used by Julia by introducing a few additional features.

Although task spawning and pipelining are possible in Windows, they exist very much in the spirit of Unix based O/S, such as Linux and OSX.

As a first example we will use curl (or wget) to retrieve a the mastering-julia.html webpage, which we done a couple of times previouly

First check that curl (or wget) is available, otherwise you will need to install it.

```
# Notice the backpacks to demarcate the command
julia> cmd = `which curl`
julia> typeof(cmd)
Cmd
```

In early version of Julia, commands execute immediately but now a **`Cmd`** object is returned and has to be explicitly run.

- Since commands run as separate tasks, it is usually preferable to suppress the output of the run() function (*by terminating with a ;*)
- The task output will goto STDOUT (by default) but can be captured and used by the Julia program

```
julia> run(cmd)
/Users/malcolm/anaconda/bin/curl
Process(`which curl`, ProcessExited(0))
```

Since the command indicates a location, the command exists and is on the executable path. We can use curl to get the webpage and can do it in one step

```
proc = run(`curl "http://LondonJulia.org/
                   mastering-julia.html"`);
<!DOCTYPE html><html><head><title>Mastering Julia Cover</title><style
type="text/css">h2 { font-family: verdana; font-size: 150%;}p  { font-
family: courier; font-size: 120%; }</style></head><body><h2>Mastering
Julia</h2><p><i>Malcolm Sherrington (2015)</i></p><img
src="images/mastering-julia.jpg" alt="Mastering Julia"
/><p> </p></body></html>
```

Look at the file: ***process.jl*** in Base for a definition of a process in Julia. The struct is mutable since (clearly) some of the fields, viz. exit codes and terminal signals etc, need to be modified.

```
#=
This has a number of useful fields including the command to run, references
```

```
to the STDIN, STDOUT and STDERR channels and the process exit code.

mutable struct Process <: AbstractPipe
  cmd::Cmd
  handle::Ptr{Cvoid}
  in::IO
  out::IO
  err::IO
  exitcode::Int64
  termsignal::Int32
  exitnotify::Condition
  closenotify::Condition
 . . .
 . . .
end
=#

# For the previous process
julia> proc.exitcode
0                    # Zero is the normal non-error exit status
```

We can use the **`less()`** function as a conventional way to view a file.

```
# First set an environment variable to point to the Julia shared folder.
ENV["JULIA_SHARE"] =
  string(Sys.BINDIR[1:end-4],"/share/julia");

# Then use less() to view it.
less(ENV["JULIA_SHARE"]*"/base/process.jl")
```

A macro called @less which can be used to display functions in Base, which *'knows'* their location, so that of JULIA_SHARE is not needed.

The functions have to be fully qualified, i.e. with arguments specified.

As an example try:- *@less sin(2.3)*

Text Processing and Pipes.

Accompanying this book there is a folder called Alice containing poems by Lewis Carroll from the Alice books + some others.

I will use these here to demonstrate executing some code snippets; the directory in the snippets needs to be set according where the folder is located.

```
# This is where the Alice text files are stored on
# my computer
julia> cd(ENV["HOME"]*"/PacktPub/Alice"); pwd()
"/Users/malcolm/PacktPub/Alice"
```

The Unix word count program **`wc`** is common to OSX and Linux, Windows system will need to have a Posix compliant shell installed on the executable path.

Like many Unix utilities it is actually a misnomer, since the output is (by default) a count of the lines, words AND characters in a text file.

We can pass a filename using the usual '$' convention and should file check it is a 'plain' file and not a binary by defining a Julia function:

```
#=
This will produce output only if the textfile exists
The trailing ';' will suppress the output from Julia, since wc runs a a
separate task and writes it output to STDOUT
=#
wc(f) = isfile(f) && run(`wc $f`);
```

We will count the number of occurrences of 'beaver" in the Lewis Carroll's poem "The Hunting of the Snark" in the Alice folder *(* in fact not actually a poem in any of the the Alice books)* and redirecting the output to a text file as part of a **`pipeline()`** call.

```
# Assume we are in the correct folder ...
# ... otherwise the files need to be fully specified
txtfile = "hunting-the-snark.txt";
logfile = "hunting-the-snark.log";

#=
Run a pipeline using the Unix utility grep to search for
lines with beaver in them the -i switch will ignore case
=#
run(pipeline(`grep -i beaver $txtfile`,stdout=logfile));

# Apply the wc function to the log file.
julia> wc(logfile);
19 138 821 hunting-the-snark.log
```

We can repeat this for the text *bellman* but append the output from **`grep`** to the logfile.

```
run(pipeline(`grep -i bellman
             $txtfile`,stdout=logfile,append=true));
```

```
wc(logfile);
49     371    2202 hunting-the-snark.log
```

Of course this output may not be accurate. Why? - because we may have double counted any lines containing both *beaver* and *bellman*

Note in the 'grep' we ignored case (-i) but will still find punctuation and plurals, such as *Beavers.,* as the following indicates:

```
run(pipeline(`grep -bellman $txtfile`,`grep -i beaver`));
2523:He could only kill Beavers. The Bellman looked scared,

# 2523 is the line number in the textile
```

This is OK here but not great for syntactical text analysis where we will want work with a single case and first strip off any punctuation. I'll attend to this later.

Piping the above thorough *'wc'* will give us the correction needed to the number of lines containing both beaver and bellman

Finding large files

I have included below a function which I find useful to find any large files in a specific directory or attending its subdirectories. It is based on the almost incomprehensible Unix utility and I will leave it upto the reader to check out just what it is doing.

```
# Note the enclosing the * and {} in quotes is needed
# by Julia and not by 'find'
# Running the command in a try/catch block is to allow for
# the case that not files are found.

function ftop(dir=".", nf=20)
  @assert nf > 0
  ef = `find $dir -type f -iname "*" -exec du -sh "{}" + `
  try
    run(pipeline(ef,`sort -rh`,`head -$nf`))
    println("Done.")
  catch
   println("No files found.")
   end
 end

# Running this against my PAcktPub folder
```

```
julia> dd = ENV["HOME"]*"/PacktPub"
julia> ftop(dd,5)
160M      /Users/malcolm/PacktPub/Chp05/Files/access_log135M
/Users/malcolm/PacktPub/Chp06/Intro to JuliaDB.mp4 49M
/Users/malcolm/PacktPub/Chp05/Files/javaforosx.dmg3.6M
/Users/malcolm/PacktPub/.ipynb_checkpoints/AD-
3.4M        /Users/malcolm/PacktPub/Julia Documentation.pdf
```

In fact it is possible to run Julia scripts via the command lines, as well as from a notebook or in the REPL and I find it useful to incorporate the **`ftop()`** routine as a standalone script. Since this introduces some new concepts I have reproduced it below and included it with the code accompanying this chapter.

```
#=
Code to run from the command line
Source: ftop-main.jl

See if any arguments have been passed
If so process them otherwise set the defaults, if any arguments have been
passed
If so process them otherwise set the defaults
=#

# Julia passes the command line arguments in an array ARGS
# We need to check if any have been passed

  nargs = size(ARGS)[1]
  if nargs > 2
    println("usage: ftop [dir [nfiles]]")
    exit(-1)
  else
    dir = (nargs > 0) ? ARGS[1] : "."
    nf = (nargs > 1) ? ARGS[2] : 20
  end

# Main 'find' command command
  ef = `find $dir -type f -iname "*" -exec du -sh "{}" + `

# Get and print the directory to be searched
  cwd = string(pwd(),"/",dir)
  println("Directory: $cwd")

# Then run the pipeline
   try
     run(pipeline(ef,`sort -rh`,`head -$nf`))
     println("Done.")
   catch
     println("No files found.")
```

```
    end
end
```

The way this processes command lines leaves a little to be required since it is necessary to define a directory in order to change the number of files. There is a very useful package ArgParse, due to Carlo Baldassi, which provide much more flexibility, including specifying '-' style switch and accompanying values. The code I have included uses this.

The inclusion of packages can slow down running scripts from the command line as additionally they need to be interpreted.

ArgParse is in native code, is not too complex and has no additional dependences; so given the time required to recurse through large directory structures this is quite acceptable.

Perl One-Liners

Perl has falled out of fashion somewhat with the current popularity of Python but in my opinion still remains one of the best languages for data munging.

Unix distros and OSX (normally) have Perl available but on Windows it needs to installed and on the executable path.

Julia performance in handling string is not one of its greatest strenghs
so, where normal Unix utilities fall short, the processing large files can successfully done using Perl.

Julia has introduced an analytical engine (JuliaDB) to tackle the processing and analysis of large datasets, employing some clever memory management techniques.

We will be discussing this in the next chapter

Perl is well known for its (perhaps sometimes overly) compact coding style by this does lead to the ability to create some quite powerful one-liners. Indeed there are numerous webpages [google: "perl one liners"], and also a book by `Peteris Krumins`

On OSX and Linux, there is a word list in the directory /usr/share/dict and

the following command outputs an palindromes of 6 or more letters

I am not intending to teach Perl but most of the syntax is easily followed.

```
julia> cmd = `perl -nle 'print if $_ eq reverse &&
                  length > 5' /usr/share/dict/words`
julia> run(cmd)
deededdeggedhallahkakkakmurdrumredderrepaperretterreviverrotatorsooloostebb
etterret
```

To capture the output of palindrome command, we will use the stdout argument to the pipeline function to write to a temporary file

```
# Grab the name for a temporary file
tmpfile = mktemp(tempdir());

cmd = `perl -nle 'print if $_ eq reverse && length > 5'
/usr/share/dict/words`

run(pipeline(cmd; stdout=tmpfile));
dmp = read(tmpfile);
run(`rm -f $tmpfile`); # Remove the temporary file

# Dump is a byte array
julia> (eltype(dmp),length(dmp))
(UInt8, 97)
```

To process further we will need to a byte array to a string and note the carriage returns ("\n") delimits lines.

```
julia> ss = String(dmp)
deedeed\ndegged\nhallah\nkakkak\nmurdrum\nredder\nrepaper\nretter\nreviver\
nrotator\nsooloos\ntebbet\nterret\n
```

So we will have to remove the trailing carriages returns and then split into words, which returns a string array.

```
julia> sa = split(chomp(ss),"\n")
julia> typeof(sa)
Array{SubString{String},1}
```

We have remarked that PERL is good (and quick) for processing strings and works well with large files.

The following example gets the top 10 words in the Hunting of the Snark, ignoring blank lines. Again I'll leave it to the reader to reason the function of the O/S and Perl command(s) .

```
cd(ENV["HOME"]*"/PacktPub/Alice")
fl = "hunting-the-snark.txt";
c1 = `perl -pne 'tr/[A-Z]/[a-z]/' $(fl)`;
c2 = `perl -ne 'print join("\n", split(/\s+/,$_));print("\n")'`;
run(pipeline(c1,c2,`sort`,`grep -ve '^$'`,`uniq -c`,
                   `sort -rn`,`head -10`));
299 the 153 and 123 a 105 to  91 it  82 with  76 of  76 in  75 he  69 that
```

If we look at difference cases of Bellman (ignoring the case)

```
julia> run(pipeline(c1,c2,`sort`,`grep -ve '^$'`,
           `uniq -c`,`sort -rn`,`grep -i Bellman`));
25 bellman 4 bellman, 1 bellman's
```

As we suspected previously there is a problem with punctuation marks. The problem is particular more obvious when we look at the bottom (10) words.

```
julia> run(pipeline(c1,c2,`sort`,`grep -ve '^$'`,
                   `uniq -c`,`sort -rn`,`tail -10`));
   1 'fritter   1 'friends,   1 'for,   1 'dunce.')   1 'come,   1 'candle-
ends,'   1 'but,   1 'be   1 'a   1 '-jum!'
```

So we need to add an extra task in the pipeline; the command c2 in the snippet below provides this:

```
function munge(fl)
  c1 = `perl -pne 'tr/[A-Z]/[a-z]/' $(fl)`;
  c2 = `perl -pne 's/()[[:punct:]]//g'`;
  c3 = `perl -ne 'print join("\n",
                       split(/\s+/,$_));print("\n")'`
  c4 = `grep -ve '^$'`
  read(pipeline(c1,c2,c3,`sort`,c4,`uniq -c`,`sort -rn` ))
end

julia> text = munge("hunting-the-snark.txt");
julia> (eltype(text),length(text))
(UInt8, 15889)

julia> lines = split(String(text),"\n");
julia> n = length(lines)
1323
```

If we now look at the bottom 10 entries, the punctuation has been removed.

```
s2 = [split(lines[i]) for i = n-9:n]
10-element Array{Array{SubString{String},1},1}: ["1", "aided"]        ["1",
"aghast"]       ["1", "ages"]         ["1", "affectionate"] ["1", "advice"]
["1", "additional"]   ["1", "aboard"]       ["1", "able"]         ["1",
"abetted"]
```

```
[]
```

Process in/out channels

We saw previously that the Julia process structures had number of useful fields. In the next example in/out channels to capture the I/O

```
#=
We should still be in the Alice folder
The Perl script rev.pl is in the parent folder
As the name suggest its function is read its input stream, reverse the
lines and write to the output stream
=#

julia> jabber = "jabberwocky.txt";
julia> proc = open(`../rev.pl $jabber`,"r+");
julia> close(proc.in);
```

We have closed the input stream but the output is still open, so we can read the lines from it and display the reversed poem, not forgetting first to close the output stream too.

```
julia> poem = readlines(proc.out);
julia> close(proc.out);
julia> poem
34-element Array{String,1}: "sevot yhtils eht dna ,gillirb sawT'"
":ebaw eht ni elbmig dna eryg diD"          ",sevogorob eht erew ysmim llA"
".ebargtuo shtar emom eht dnA"              ""
"!nos ym ,kcowrebbaJ eht eraweB'"           "!hctac taht swalc eht ,etib
taht swaj ehT" "nuhs dna ,drib bujbuJ eht eraweB"          "'!hctansrednaB
suoimurf ehT"               ""
":dnah ni drows laprov sih koot eH"         "-- thguos eh eof emoxnam eht
emit gnoL"    ",eert mutmuT eht yb eh detser oS"          ⋮
"daeh sti htiw dna ,daed ti tfel eH"        ".kcab gnihpmulag tnew eH"
""                                          "?kcowrebbaJ eht nials uoht
tsah dnA'"      "!yob hsimaeb ym ,smra ym ot emoC"          "'!yallaC
!hoollaC !yad suojbarf hO"        ".yoj sih ni deltrohc eH"
""                                          "sevot yhtils eht dna ,gillirb
sawT'"       ":ebaw eht ni elbmig dna eryg diD"          ",sevogorob eht
erew ysmim llA"             "ebargtuo shtar emom eht dnA"
```

Other languages

We are not limited to just the Perl. Any scripting language which has a command line option and can process standard input, streaming it to standard output will do.

Below are three examples:

Perl6

Perl6 is now considered as a new and separate language from Perl (5); some see it as a reason for the decline in the latter since the syntax of the two differs markedly in some places. Perl6, unlike Perl, is not distributed as a standard.

The favour distribution for Perl6 can be obtained from https://rakudo.org

It is necessary to be able to 'find' perl6; the command which we saw earlier is a convenient method is to setup a symbolic link to the binary and it in a folder on the excuse path.

For myself (on OSX) perl6 -> /Applications/Rakudo/bin/perl6
and I put it in my '$HOME/bin' folder

```
# Check that perl6 is available
julia> run(`which perl6`);
/Users/malcolm/bin/perl6
```

Perl6 is (naturally) much more richer syntactic. Below we will use it to from the line of the greatest length in the Hunting of the Snark

run(`perl6 -e 'my $max=""; for (lines) {$max = $_ if .chars > $max.chars};END { $max.say }' hunting-the-snark.txt`);

```
julia> run(`perl6 -e 'my $max="";
           for (lines) {$max = $_ if .chars > $max.chars};
                END { $max.say }' hunting-the-snark.txt`);
'You must know ---' said the Judge: but the Snark exclaimed 'Fudge!'
```

Ruby

Ruby is not as popular as it was a decade ago, again probably due to the current interest in Python. It is a pure object oriented language and is distributed in OSX and Linux due to the fact it is used in a few system maintenance procedures.

The following example turns all characters in the Snark poem, after 40th spot to RED; you will need to run it in a notebook or the REPL to confirm it.

```
# Set the textile
snark = "hunting-the-snark.txt";

# And the Ruby command
```

```
rubycmd =`ruby -e 'w = $*.shift; $<.each {
 |l| puts "#{l}\e[31m#{
 l.chop!.slice!(w.to_i..-1)}\e[0m" }'
 40 $snark`;

# Run it, redirect to a temporary file to see ...
# ... some of the output
redout = tempname();
run(pipeline(rubycmd, stdout=redout));

# Use the head utility to show 32 lines.
run(`head -32 $redout`);
run(`rm -f $redout`);
```

Python

We could not leave this section without giving an example in Python. This time lets switch the poem to the Jabberwocky and encode it in Base 64

```
cd(ENV["HOME"]*"/PacktPub/Alice")
f1="jabberwocky.txt"
f2="jabberwocky.b64"
b64encode = `python -c 'import base64,sys;
base64.encode(open(sys.argv[1],"rb"),
              open(sys.argv[2],"wb"))' $f1 $f2`
 run(b64encode);
```

and check it:

```
julia> run('cat $f2');
J1R3YXMgYnJpbGxpZywgYW5kIHRoZSBzbGl0aHkgdG92ZXMKRGlkIGd5cmUgYW5kIGdpbWJsZSB
pbiB0aGUgd2FiZToKQWxsIG1pbXN5IHdlcmUgdGhlIGJvcm9nb3ZlcywKQW5kIHRoZSBtb21lIH
JhdGhzIG91dGdyYWJlLgoKJ0Jld2FyZSB0aGUgSmFiYmVyd29jaywgbXkgc29uIQpUaGUgamF3c
yB0aGF0IGJpdGUsIHRoZSBjbGF3cyB0aGF0IGNhdGNoIQpCZXdhcmUgdGhlIEp1Ymp1YiBiaXJk
LCBhbmQgc2h1bgpUaGUgZnJ1bWlvdXMgQmFuZGVyc25hdGNoIScKCkhlIHRvb2sgaGlzIHZvcnB
hbCBzd29yZCBpbiBoYW5kOgpMb25nIHRpbWUgdGhlIG1hbnhvbWUgZm9lIGhlIHNvdWdodCAtLQ
pTbyByZXN0ZWQgaGUgYnkgdGhlIFR1bXR1bSB0cmVlLApBbmQgc3Rvb2QgYSB3aGlsZSBpbiB0a
G91Z2h0LgoKQW5kLCBhcyBpbiB1ZmZpc2ggdGhvdWdodCBoZSBzdG9vZCwKVGhlIEphYmJlcndv
Y2ssIHdpdGggZXllcyBvZiBmbGFtZSwKQ2FtZSB3aGlmZmxpbmcgdGhyb3VnaCB0aGUgdHVsZ2V
5IHdvb2QsCkFuZCBidXJibGVkIGFzIGl0IGNhbWUhCgpPbmUgdHdvISBPbmUgdHdvISBBbmQgdG
hyb3VnaCBhbmQgdGhyb3VnaApUaGUgdm9ycGFsIGJsYWRlIHdlbnQgc25pY2tlci1zbmFjayEKS
GUgbGVmdCBpdCBkZWFkLCBhbmQgd2l0aCBpdHMgaGVhZApIZSB3ZW50IGdhbHVtcGhpbmcgYmFj
ay4KCidBbmQgaGFzdCB0aG91IHNsYWluIHRoZSBKYWJiZXJ3b2NrPwpDb21lIHRvIG15IGFybXM
sIG15IGJlYW1pc2ggYm95IQpPaCBmcmFiam91cyBkYXkhIENhbGxvb2ghIENhbGxheSEnCkhlIG
Nob3J0bGVkIGluIGhpcyBqb3kuCgonVHdhcyBicmlsbGlnLCBhbmQgdGhlIHNsaXRoeSB0b3Zlc
wpEaWQgZ3lyZSBhbmQgZ2ltYmxlIGluIHRoZSB3YWJlOgpBbGwgbWltc3kgd2VyZSB0aGUgYm9y
```

```
b2dvdmVzLApBbmQgdGhlIG1vbWUgcmF0aHMgb3V0Z3JhYmUK
```

Now reverse the process, this time using the base64 command:

```
julia> run(`base64  --decode $f2`);
Twas brillig, and the slithy tovesDid gyre and gimble in the wabe:All mimsy
were the borogoves,And the mome raths outgrabe.'Beware the Jabberwock, my
son!The jaws that bite, the claws that catch!Beware the Jubjub bird, and
shunThe frumious Bandersnatch!'He took his vorpal sword in hand:Long time
the manxome foe he sought --So rested he by the Tumtum tree,And stood a
while in thought.And, as in uffish thought he stood,The Jabberwock, with
eyes of flame,Came whiffling through the tulgey wood,And burbled as it
came!One two! One two! And through and throughThe vorpal blade went
snicker-snack!He left it dead, and with its headHe went galumphing
back.'And hast thou slain the Jabberwock?Come to my arms, my beamish boy!Oh
frabjous day! Callooh! Callay!'He chortled in his joy.'Twas brillig, and
the slithy tovesDid gyre and gimble in the wabe:All mimsy were the
borogoves,And the mome raths outgrabe
```

Working with the filesystem

Julia provides a variety of functions for reading folders and processing files natively, and we will meet some of these in the next chapter. However as a taster I have included a few examples here.

I got an error after exiting the following process
So have wrapped it in a try/catch block

```
# I got an error after exiting the following process
# I wrapped it in a try/catch block to trap the error.
cd(string(ENV["HOME"],"/PacktPub/Chp05"));
try run(`wc $(readdir())`) catch end
 3424   11555  152017 Chp05.ipynb
 978    3444   23823 Chp05.jl
wc: Code: read: Is a directory
wc: Files: read: Is a directory
 4423   15252  190180 total

# So I defined a macro to trap any run(cmd) errors.
macro traprun(c)
  quote
    if typeof($(esc(c))) == Cmd
      try
        run($(esc(c)))
      catch
```

```
      end
    end
  end
end
```

The following function filters on a regular expression, in order to get rid of the directory warnings from the above snippet

```
function filter(pat::Regex, dir=".")
  a = Any[]
  for f in readdir(dir)
    occursin(pat,f) && push!(a, f)
  end
  return a
end
```

Find all the Jupyter notebooks in the Chp05 directory

```
@traprun `find "." -name Chp\*.ipynb`;
./Code/tutorial.ipynb
./Code/SymPy.ipynb
./Chp05.ipynb
```

Analyse an Apache (Web) Access Log

One of the major uses of Perl is in analysing log files. These are typically very large, perhaps several million rows of text. In the files accompanying this chapter I have included an Apache log file, distributed by `NASA` and freely available.

Apache access logs have a well-defined structure, known as the `Common Log Format`, consisting of 8 text fields separated by whitespace.

```
julia> cd(ENV["HOME"]*"/PacktPub/Chp05"
julia> logf = "Files/access_log";

julia> run(`/usr/bin/wc -l $logf`);
1569898 Files/access_log
```

The first 5 lines look like:

```
julia> run(`head -5 $logf`)
in24.inetnebr.com - - [01/Aug/1995:00:00:01 -0400] "GET
/shuttle/missions/sts-68/news/sts-68-mcc-05.txt HTTP/1.0" 200 1839
uplherc.upl.com - - [01/Aug/1995:00:00:07 -0400] "GET / HTTP/1.0" 304 0
uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/ksclogo-
medium.gif HTTP/1.0" 304 0
uplherc.upl.com - - [01/Aug/1995:00:00:08 -0400] "GET /images/MOSAIC-
```

```
logosmall.gif HTTP/1.0" 304 0 uplherc.upl.com - - [01/Aug/1995:00:00:08
-0400] "GET /images/USA-logosmall.gif HTTP/1.0" 304 0
```

As an example of analysing the logfile in PERL, I will demonstrate how to emulate a PERL this is not a problem "tail -5" as a one-liner.

```
cmd = `perl -ne 'push @a, $_; @a = @a[@a-5..$#a];
                        END { print @a }' $logf`
julia>  run(cmd);
gatekeeper.uccu.com - - [31/Aug/1995:23:59:49 -0400] "GET
/images/ksclogosmall.gif HTTP/1.0" 304 0
gatekeeper.uccu.com - - [31/Aug/1995:23:59:49 -0400] "GET /images/lc39a-
logo.gif HTTP/1.0" 304 0
cys-cap-9.wyoming.com - - [31/Aug/1995:23:59:52 -0400] "GET
/shuttle/missions/sts-71/movies/sts-71-launch-3.mpg HTTP/1.0" 200 57344
www-c8.proxy.aol.com - - [31/Aug/1995:23:59:52 -0400] "GET
/icons/unknown.xbm HTTP/1.0" 200 515
cindy.yamato.ibm.co.jp - - [31/Aug/1995:23:59:53 -0400] "GET
/images/kscmap-small.gif HTTP/1.0" 200 39017
```

As a practical example I have include a Perl script **`hcat.pl`** which runs through the entire log file and sets up a hash with the URL as the key and the number of times it occurs as the value.

This can serve the basis for a variety of analytical calculations.

As an example the script, then outputs the top-ten hitters.

```
# To look at the source go the Perl script ...
# ... we can use the less() function
hcount_pl = "Code/hcount.pl";
less(hcount_pl)

println("Top Ten Hitters ($logf)");
@time run(`$hcout_pl $logf`);
6530   edams.ksc.nasa.gov
4846   piweba4y.prodigy.com
4791   163.206.89.4
4607   piweba5y.prodigy.com
4416   piweba3y.prodigy.com
3889   www-d1.proxy.aol.com
3534   www-b2.proxy.aol.com
3463   www-b3.proxy.aol.com
3423   www-c5.proxy.aol.com
3411   www-b5.proxy.aol.com

Number of unique URLS in the log: 75060
5.298708 seconds (28 allocations: 1.359 KiB)
```

To emphasise: The process of creating a hash from one 1.5 million lines, from over 75 thousand unique IP addresses and computing the counts of each access entry took just 5.3 sec.

Summary

In this chapter we looked at three different ways that Julia can interact with other languages.

The first, simplest and most effective is to refer to routines, residing in shared object libraries, usually created from C or FORTRAN) compilers. We noted that this is also a convenient method to create some code and call it from Julia without any formal API being necessary.

Next we considered some of popular languages such as Python, R, Java, and described how Julia packages exist to make the process simplier; also how wrapper packages have been created to utilise existing ones, with a specific example of the Python package SymPy.

Finally we discussed Julia's ability to interface with the operating system and in conjunction with pipelining and capturing the output, showing how by this mechanism it is possible to utilise other languages in a similar fashion to Unix utilities. The examples focussed mainly on Perl but demonstrated briefly how languages such as Perl6, Ruby and Python can be treated in a similar fashion.

Index