Table of Contents

Chapter 1: Working with Data 1

Your bookmark 1

Basic Input-Output (I/O) 1

Terminal I/O 2

Terminal output 2

Terminal input 3

Text files 6

Text Processing 8

Binary Files 11

Structured Data Sets 14

CSV and other DLM files 14

CSV (and TSV) files 14

DLM files 17

HDF5 and JLD files 21

Julia Data Format 23

XML files 23

Data Frames and Statistics 27

Data Frames 27

Data Arrays 27

R-Datasets 28

(Some) Statistics 33

Kernel Densities 35

Testing hypothesis 37

Linear Regression 38

Summary 40

Index 41

1
Working with Data

In many of the examples we have looked at so far, the data used was a little artificial, being generated using random numbers. It is now time to consider how Julia uses at data held in files.

In this chapter, I'm going to concentrate on simple disk-based files and leave discussions of data stored in databases and on the web for later chapters.

Also we will look at some of the statistical approaches in Julia, both those incorporated into the Julia base and in some of the growing number of package covering statistics and related topics.

This chapter will cover:

- Basic Input-Output (I/O)
- Structured Data Sets
- Data Frames and Statistics

Basic Input-Output (I/O)

Julia views its data in terms of a byte stream. This may be from/to standard input or output, which may be superseded by a disk file.

If this is coming from a network socket or a pipe, this is essentially asynchronous but the programmer need not be aware of this as the stream will block until cleared by an I/O operation.

The primitives are the **`read()`** and **`write()`** functions, the primitives deal with binary data; with formatted data, *such as text*, a number of other functions are layered on top of these.

Julia uses the **libuv**, designed to support in NodeJS.

It provides very efficient operations both locally and over a distributed system *(viz. a network)* and is central to Julia's efficient approach to I/O.

Terminal I/O

We remarked above that Julia has the concept of standard input (stdin) and standard output (stdout), inherited from an idea in Posix compliant operating systems and with the purpose of streaming data through I/O channels.

All streams in Julia have at least a read and a write routine; these take as their first argument: a filename, Julia channel/stream object or, *for read(),* a command.

- read(filename::AbstractString, args...)
- read(s::IO, nb=typemax(Int))
- read(s::IOStream, nb::Integer; all=true)
- read(command::Cmd)
- read(command::Cmd, String)
- write(io::IO, x)
- write(filename::AbstractString, x)

I/O streams needs to be opened and closed, for files, sockets etc., and we will deal with how this is done in the relevant sections.

The standard streams **`stdin`** and **`stdout`** , plus an additional *output* stream for error messages **`stderr`** are opened by Julia when it starts up and should not normally be closed.

Terminal output

Outputting to either of the **`stdout`** or **`stderr`** streams is easy achieved via the **`write()`** routine:

```
julia> write(stdout, "Help me!!!\n")
Help me!!!
11
julia> write(stderr, "Error messages should go here.\n")
Error messages should go here.
31
```

Note:

- Writing is literal, if a newline is required it must be specified; '\n' works for Linux, OSX and Windows
- The routine returns the number of characters written; due to the asynchronous nature of the I/O, the REPL outputs the string before the number of characters which were written.
- By default messages sent to stderr goto the same display device as stdout, but it is possible to redirect one or both of these to split the output streams.
- In earlier examples, using the *print* function, rather than *write*, absorbs the byte count; also the *println* function will append a carriage return.

When requiring formatted output, we saw that Julia employs the macro system for generating the boiler-plate.

In fact, the whole system is extremely complex that it warrants its own module in STDLIB: viz. Printf, and this must (now) be included via a **`using Printf`** statement.

Also in the module is a **`@sprintf`** macro which will print formatted output to a string rather than to an output stream.

That Julia does not have an **`@fprintf`** macro to write to a file, instead it uses an extended form of **`@printf`** as we will see later.

Terminal input

Reading from the terminal is more complex.

In the first chapter we looked at playing a game of "Bulls and Cows", to give a flavour of Julia and remarked that terminal input within a Jupiter notebook is different from that in the REPL ; it may be clearer now that this is due to the way that Jupyter handles the input stream.

The following may help to clarify this:-

```
# Try the following in the Jupyter and the REPL
# (CR => carriage return, ^D => control-D)

# Type a letter + CR
julia> read(stdin,Char)
A
```

```
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)

# Type a string + ^D (CR is part of the string)
julia> read(stdin,String)
Goodbye cruel world.
"Goodbye cruel world.\n"

# Type a number + CR
julia> read(stdin,Int32)
1234
875770417

# Type same number + CR
julia> read(stdin,UInt32)
1234
0x34333231
```

The behaviour when inputting the integer seems bizarre, however the latter result, from the Uint gives a clue. **0x34333221** is actually a representation of the ACSII codes **1234** but reversed.

So, the prior result (**875770417**), is the ASCII representation of 1234, reversed and converted to an integer!

When reading from ***stdin*** which out specifying an argument, we get the following:

```
julia> read(stdin)
a
bc
d
7-element Array{UInt8,1}:
 0x61
 0x0a
 0x62
 0x63
 0x0a
 0x64
 0x0a
```

The input is terminated with a control-D and creates a 7-element byte array, comprising the ASCII codes for the characters and the returns.

In it also possible to define the number of characters required and terminate this with a return:

```
julia> read(stdin,4)
abcd
4-element Array{UInt8,1}:
```

```
 0x61
 0x62
 0x63
 0x64
```

In the above additional characters will be interpreted a variable and give an defined variables error, i.e. inputting *abcdef* to the above results in:

```
julia> ef
ERROR: UndefVarError: ef not defined
```

Julia also has a **`readline()`** function which, because of what is described in the documentation, as a hack, works in Jupyter as well as the REPL.

```
julia> a = readline(stdin)
abcd
"abcd"

# Note that stdin is default, so not usually needed
```

This always returns a string and leads to methods for inputting variables such as integers and floats by converting the returns strings which bit of care.

```
julia> function getInt()
  s = chomp(readline())
  try
    parse(eltype(1),s)
  catch ex
    println(ex, "\nCan't convert $s to an integer")
  end
end
getInt (generic function with 1 method)

julia> getInt()
1234
1234

julia> function getFloat()
  s = chomp(readline())
  try
    parse(eltype(1.1),s)
  catch ex
    println(ex, "\nCan't convert $s to a float")
  end
end
getFloat (generic function with 1 method)

julia> getFloat()
```

```
12.34
12.34
```

Text files

When dealing with files on disk these need to be opened and the process establishes a channel to the file which may be for reading, writing or both.

How this is done depends on the arguments to the **`open()`** call and there are two main (equivalent) syntaxes,

- open(filename::AbstractString; keywords...)
- open(filename::AbstractString, [mode::AbstractString]

The difference is largely historic and inherent form the forms in operating system shell commands.

The keywords consists of a set of 5 boolean arguments as :

read	open for reading	*not* write
write	open for writing	truncate *or* append
create	create if does not exist	*not* read *&* write *or* truncate *or* append
truncate	truncate to zero-size	*not* read *&* write
append	seek to end of file	*false*

Not all of the combinations of the flags are logically consistent, so it is more usual to employ the other form where the mode is passed as an ASCII string

r	read
w	write, create, truncate
a	write, create, append
r+	read & write
w+	read & write, create, truncate
a+	read & write, create, append

The default when neither of the above forms is used is to open the file for reading.

To demonstrate some simple file operations, let's output the first verse of the Jabberwocky

```
# Pick up the location of the file,
# This is mine it may differ for the reader
julia> fname = String(ENV["HOME"],"/PacktPub/Alice/jabberwocky.txt");
```

```
#=
One way to check if this exists is to use isfile(). If it were but is not a
regular file this would also return false; there is an isdir() call to
check if a filespecn is a directory
=#

julia> isfile(fname)  # => true

# Open for reading and read the first 4 lines
# Allow for the Julia scoping rules
julia> fip = open(flip)
julia> k = 0;
julia> while !eof(tip)
  ln = readline(fip)
  global k = k+1
  (k > 4) ? break : println(ln)
end
'Twas brillig, and the slithy toves
Did gyre and gimble in the wabe:
All mimsy were the borogoves,
And the mome raths outgrabe.

julia> close(fip)   # Remember to close the channel
```

To save the need to monitor the number of lines and break at a certain limit, I have created a file *jabber4.txt* consisting of the first verse of the Jabberwocky.

An alternative syntax for the **`open()`** command is as below:

```
julia> flp4 =
 ENV["HOME"]*"/PacktPub/Alice/jabber4.txt";

julia> open(flp4) do pn4
  while !eof(pn4)
    println(readline(pn4))
  end
end

julia> eof(pn4)
 ERROR: UndefVarError: pn4 not defined
 Stacktrace:
 [1] top-level scope at none:0
```

Conveniently we can now read all the file in to a variable without explicitly opening it and so not having to close it . Take note that the data is returned as a byte array (*the file may be a binary*).

```
#=
```

```
So for a text file we need to convert it

jb4 = String(read(flp4))

This needs to be split of '\n's to reproduce the familiar first verse
=#
```

Instead of supplying a filename as the first argument in **`open()`**, it is permitted to precede it with a function which takes an IOStream as its argument

```
julia> capitalize(f::IOStream) =
       chomp(uppercase(String(read(f))));

julia> split(open(capitalize, flp4),"\n")
 4-element Array{SubString{String},1}:
 "'TWAS BRILLIG, AND THE SLITHY TOVES"
 "DID GYRE AND GIMBLE IN THE WABE:"
 "ALL MIMSY WERE THE BOROGOVES,"
 "AND THE MOME RATHS OUTGRABE."
```

Text Processing

In the previous chapter we ended by looking at utilising operating system utilities and scripting languages such as Perl to process (i.e. modify) text.

From the 'capitalise' example above, it is clear that this can be done purely using Julia - in this section we will delve a little further the subject.

Previously we used Perl to reverse a lines of a poem, here is a native Julia version

```
julia> open( flp4) do pn4
  while !eof(pn4)
  println(reverse(chomp(readline(pn4))))
  end
end
sevot yhtils eht dna ,gillirb sawT':ebaw eht ni elbmig dna eryg
diD,sevogorob eht erew ysmim llA.ebargtuo shtar emom eht dnA
```

Again we need to chomp the line, reverse it and then use **`println()`**; otherwise the '\n' would come at the front of each line.

Let's continue with a more tricky task, viz. emulating the 'wc' command. In the following routine only the words in a file are counted; actually this is the more difficult than counting lines and characters, which I will leave as an exercise for the reader.

```
# We will split on white space and also
```

```
# punctation marks
julia> const PUNCTS = [' ','\n','\t','-','.',',',':',';','!','?','\'','"'];

#=
The routine works by creating a hash (Dict) with the unique words found as
keys and the count as the values,
=#
julia> function wordcount(text)
   wds = split(lowercase(text),PUNCTS;
                     keepempty = false)
   d = Dict()
   for w = wds
     d[w] = get(d,w,0)+1
   end
   return d
end

#=
The get(d,w,0) call retrieves the value of the key [w] from the Dict d;
when the key does not exist an entry is created and the default value of 0
is used
=#
```

So we can read the entire poem into a string and apply this function, since we are splitting on '\n' as well as spaces.

Test it on our 4 line Jabberwocky:

```
# Test it on our 4 line Jabberwocky
julia> wordcount(String(read(flp4)))
Dict{Any,Any} with 18 entries:  "gyre"      => 1  "and"       => 3
"brillig"   => 1  "raths"     => 1  "in"        => 1  "mome"      => 1
"toves"     => 1  "mimsy"     => 1  "twas"      => 1  "did"       => 1
"the"       => 4  "borogoves" => 1  "were"      => 1  "all"       => 1
"wabe"      => 1  "outgrabe"  => 1  "slithy"    => 1  "gimble"    => 1

# Most words only occur once but 'and' &
# 'the' have a higher frequency, and agree
# with a quick scan of the file.
```

`wordcount()` returns a dictionary of the words in a file.

So if we collect the values (i.e the counts) and sum them, then this gives a total for the file.

```
#=
We will use the @print macro
Setup a constant to point to the Alice folder, Notice the convenient
trailing /
```

```
=#
julia> using Printf
julia> const ALICEDIR =
  ENV["HOME"]*"/PacktPub/Alice/";

#=
Filter to look just a the '.txt' files in the Alice directory. We can
collect all the values of the Dict in an array and sum it for the total in
the file
=#
julia> for fname in readdir(ALICEDIR)
  if match(r"\.txt$", fname) != nothing
    open(ALICEDIR*fname) do f
      n = sum(collect(values(wordcount(
                      String(read(f))))))
      @printf "%s: %d\n" fname n
    end
  end
end
aged-aged-man.txt: 512father-william.txt: 278hunting-the-snark.txt:
4524jabber4.txt: 23jabberwocky.txt: 168lobster-quadrille.txt: 231mad-
gardeners-song.txt: 348red-snark.txt: 5153voice-of-the-lobster.txt:
158walrus-and-carpenter.txt: 623
```

Let's look at some of the characters in the Hunting of the Snark.
In true Carolina fashion all the occupants on the *'hunt'* had names beginning with 'B'.

```
# Create the Dict for the Snark poem
julia> snarkDict = wordcount(String(
     read(ALICEDIR*"hunting-the-snark.txt")
))

julia> wds = ["baker","banker","barrister",
       "beaver","bellman","boots",
        "butcher"];
julia> for w in wds
  @printf "%12s => %4d\n" w snarkDict[w]
end
       baker  =>   10      banker  =>    7   barrister  =>    5      beaver
=>   18     bellman  =>   30       boots  =>    3     butcher  =>   13
```

In fact because of splitting on quote(*'*), we will treat ***s*** as a word because of entries such as Bellman's.

So our routine should take account of this and similar anomalies if counting all the words in the poem rather than just matching selected

ones.

Binary Files

Julia can handle binary files as easily as text files using **`read()`** and **`write()`** .

Earlier we created a simple grayscale image for a Julia set. In the following we will read the file and invert the image

```
cd(ENV["HOME"]*"/PacktPub/Chp06");
img = open("juliaset.pgm");
magic = chomp(readline(img))
```

We can open the file in the normal way.

The first value is the 'magic' number P5 (for a PGM file), terminated by a '\n', so we can get that using **`readline()`** .

We will be creating another PGM file so can write 'magic' number.

The next line comprises 3 integers, the wide and height of the image and the maximum pixel value (usually 255).

These are read and copied without change.

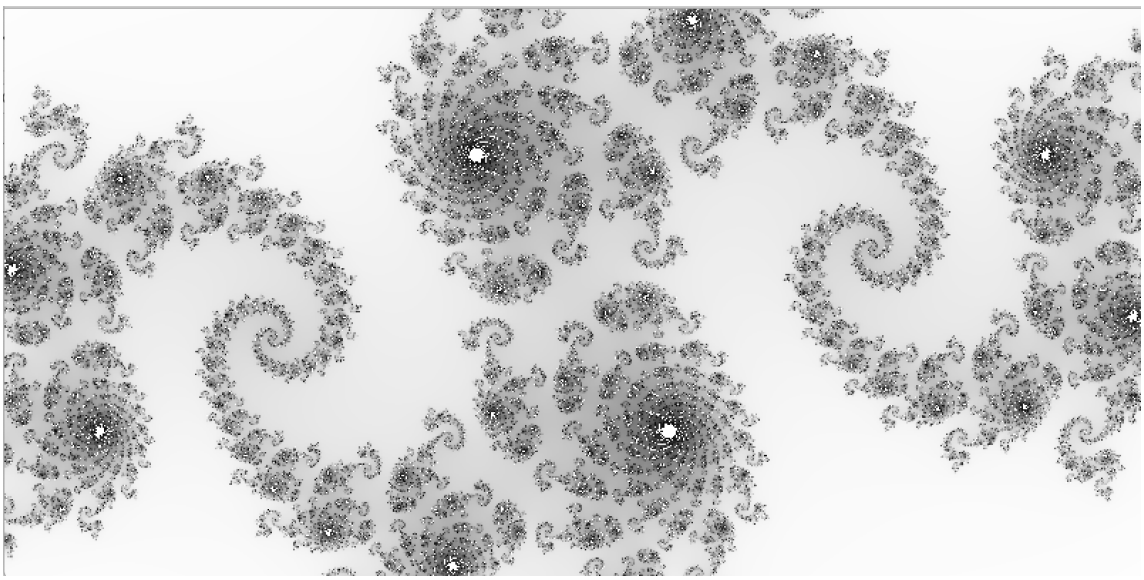
```
if magic == "P5"
   out = open("Files/jsetinvert.pgm", "w");
   println(out, magic);
   params = chomp(readline(img));
                        # => "800 400 255"
   println(out, params);
# Params splits to strings. we need integers
  (wd,ht,pmax) = parse.(Int64,split(params))

# Create a byte array and read ALL the
# image data in one call.
 np = wd*ht;
 buf = Array{UInt8,1}(undef,np)
 readbytes!(img, buf, np);
# Invert the gray scales and write it back
 bufX = [UInt8(255 - buf[i]) for i = 1:np]
 write(out,bufX))
 close(out);
else
 error("Not a NetPBM grayscale file")
end
```

```
close(img)
```

Notes:

1. The routine reads all the (remaining) bytes into a single byte buffer of size *(width x height)*
2. For large images it might be necessary to process the image row-by-row in a loop, the logic will be virtually the same.
3. **`readbytes!(img,buf,n)`** reads upto *'n'* bytes in the the byte array *'buf'*
4. The array will be extended to size *'n'* if it is too small, so we define a 0-size array to begin.
5. If there are insufficient bytes remaining in the file, the rest of the buffer is filled with nulls (0x00).

Here is the resulting image:



It is also possible to add some colour using the following algorithm for each pixel value.

This is called pseudo-colour because there is still a single pixel value in the range [0,255], true colour would have 3 values, for each of red, green and blue.

```
julia> function pseudocolor(pix)
```

```
  if pix < 64
    pr = UInt8(0)
    pg = UInt8(0)
    pb = UInt8(4*pix)
  elseif pix < 128
    pr = UInt8(0)
    pg = UInt8(min(4*(pix - 64),255))
    pb = UInt8(255)
  elseif pix < 192
    pr = UInt8(0)
    pg = UInt8(255)
    pb = UInt8(min(4*(192 - pix),255))
  else
    pr = UInt8(min(4*(pix - 192),255))
    pg = UInt8(min(4*(256 - pix),255))
    pb = UInt8(0)
  end
  return (pr, pg, pb)
end
```

Because we are changing a PGM file to a PPM one, the magic number needs changing from P5 to P6. The rest of the code is similar than above except writing the [r,g,b] values from the **`pseudocolour()`** routine in a loop

```
if magic == "P5"
  out = open("jsetcolor.ppm", "w");
  println(out, "P6");
  params = chomp(readline(img));
              # => "800 400 255"
  println(out, params);
  (wd,ht,pmax) = parse.(Int64,split(params))
   np = wd*ht;
   buf = Array{UInt8,1}(undef,np)
   readbytes!(img, buf, np);
   for j = 1:np
    (r,g,b) = pseudocolor(buf[j]);
    write(out,UInt8(r))
    write(out,UInt8(g))
    write(out,UInt8(b))
 end
 close(out);
else
 error("Not a NetPBM grayscale file")
end
close(img)
```

The colour version created by this code is in the notebook and also is provided with the files accompanying this chapter.

Structured Data Sets

In this section we will look at files which contain *metadata* to indicate the way that data is arranged, as well as the values themselves.

This include simple delimited files, such as the familiar comma-separated-values (CSV) and also files with incorporating more descriptive metadata such as XML and HDF5.

Finally we will discuss the important topic of Julia's dataframes ; familiar to all R users and also implemented in Python by the *pandas* module.

CSV and other DLM files

Data is often presented in table form as a series of rows representing individual records and fields corresponding to a data value for that particular record, rather than the relatively unstructured forms we have seen in the previous files.

Columns in the tableau are consistent, in the sense that they may all be integers, floats, dates etc, and are to be considered as the same 'class' of data.

This will be familiar to most readers, as it maps directly to the way data is held in a spreadsheet.

CSV (and TSV) files

One of the 'oldest' forms of such representing such data is the Comma-Separated-Value (CSV) file. This is essentially an ASCII file in which fields are separated by commas `,` and records *(rows)* by a newline `'\n'` .

There is an obvious problem if some of the fields are strings containing commas and so CSV files use quoted text *(normally using a double quote `"`)* to overcome this, however this gives rise to the new question of how to deal with text fields which contain the `"` character.

In fact, the CSV file was never defined as a standard and a variety of implementations exist. However, the principle is clear that we require a method to represented involving a field separator and a record separator together with a way to identify any cases where the field and record separators are to be interpreted as regular characters. These types of file are often name delimited and Julia supports DLM in addition to more specific CSV ones.

We will start by looking at CSV files, supported by the CSV.jl package. In the files section of this chapter I have included some data on the stock prices of Apple

```
# Apple stock has the abbreviation AAPL
# In addition to the CSV package we will need
# the STDLIB modules: Statistics and Printf.

julia> using CSV, Statistics, Printf
julia> cd(ENV["HOME"]*"/PacktPub/Chp06/")
julia> aaplcsv = "Files/aapl.csv"; isfile(aaplcsv)
```

The **`CSV.File()`** routine returns the schema; notice the *unions of Missing and Floats64 :* as the data may have missing values. The first 5 values are the most common, correspondent to the Open/High/Low/Close prices and the Volume traded on specific dates. These are sometimes referred to as OHLC[V] values.

```
julia> aapl = CSV.File(aaplcsv)
CSV.File("Files/aapl.csv",rows=8336):Tables.Schema: :Date
Union{Missing,Date}    :Open                Union{Missing,Float64} :High
Union{Missing,Float64} :Low                 Union{Missing,Float64} :Close
Union{Missing,Float64} :Volume              Union{Missing,Float64}
Symbol("Ex-Dividend")
                     Union{Missing,Float64} Symbol("Split Ratio")
Union{Missing,Float64} Symbol("Adj.Open")   Union{Missing,Float64}
Symbol("Adj.High")   Union{Missing,Float64} Symbol("Adj.Low")
Union{Missing,Float64} Symbol("Adj.Close")  Union{Missing,Float64}
Symbol("Adj.Volume") Union{Missing,Float64}

# First look at the field(names)s comprising
# the data structure.
julia> fieldnames(AAPL)
(:names, :types, :name, :io, :parsinglayers, :positions,
:originalpositions, :currentrow, :lastparsedcol, :lastparsedcode, :kwargs,
:pools, :strict, :silencewarnings)

# We are currently at the first position
julia> aapl.currentrow
Base.RefValue{Int64}(1)

# Print the first 5 rows
julia> k = 0;
```

```
julia> for r in aapl
   rChange = r.Close - r.Open
   rSpread = r.High - r.Low
   @printf "%s : %.2f\n"  r.Date  rChange
   global k = k + 1
   if k > 5 break end
end
2013-12-31 : 6.852013-12-30 : -2.942013-12-27 : -3.732013-12-26 :
-4.202013-12-24 : -2.222013-12-23 : 2.09

# Check that we are now at the 6th row.
julia> aapl.currentrow
Base.RefValue{Int64}(6)
```

To look at all the values the most convenient method is to 'pipe' the *CSV.File* structure to a dataframe; we will discuss these in more detail later.

```
# Useful to sort the dataframe in place
# Otherwise the latest values are first.

julia> using DataFrames
julia> df = aapl |> DataFrame
julia> sort!(df)
 Date          Open  High  Low   Close . . .
1  1980-12-12  28.75 28.88 28.75 28.75 . . .
2  1980-12-15  27.38 27.38 27.25 27.25 . . .
3  1980-12-16  25.38 25.38 25.25 25.25 . . .
4  1980-12-17  25.88 26.0  25.88 25.88 . . .
```

Data frames can be queried with the "Queryverse", of which more later in Chapter 9, when we return to the subject of data sources and databases.

```
#= The following returns the OHLC values vs Date from the previousdata
frame df, starting with values beginning at 20-12-2013;
these are the final 7 values in the dataset.
=#

julia> using Query, Dates

julia> x = @from i in df begin
  @where i.Date >= Date(2013,12,20)
  @select {i.Date, i.Open, i.High,
                   i.Low, i.Close}
  @collect DataFrame
end
```

Output:

```
1 2013-12-20 545.43 551.61 544.82 549.02
2 2013-12-23 568.0  570.72 562.76 570.09
3 2013-12-24 569.89 571.88 566.03 567.67
4 2013-12-26 568.1  569.5  563.38 563.9
5 2013-12-27 563.82 564.41 559.5  560.09
6 2013-12-30 557.46 560.09 552.32 554.52
7 2013-12-31 554.17 561.28 554.0  561.02
```

DLM files

As was remarked earlier CSV files are a particular instance of delimited files (DLM). The problem of using comma as a field separator is exacerbated by enclosing strings in quotes, but this then has the effect of making the `"` as a specific markup character.

A second common type of files is one which uses **`TAB`** as a field separator, largely making the need for `"` redundant.

In the case of these, so-called Tab-Separated-Value files, the DLM module is useful.

In the Files folder is a file UKH-Prcs.tsv; this is in TSV format, providing the house prices in various UK regions, monthly for the 20 years from 1996 to 2017.

```
# Pickup the file UKH-Prcs.tsv files
julia> using DelimitedFiles
julia> cd(ENV["HOME"]*"/PacktPub/Chp06/")
julia> ukhptsv = "Files/UKH-Prcs.tsv"
julia> isfile(ukhptsv)
true
```

To input the data we can use the function :
readdlm(source, delim::AbstractChar, eol::AbstractChar; options...)

If all data is numeric, the result will be a numeric array; if some elements cannot be parsed as numbers, a heterogeneous array of numbers and strings is returned.

One of the options is 'header', which is a boolean - if set to true the routine returns the data and header in separate arrays, the header being read as the first line in the dataset.

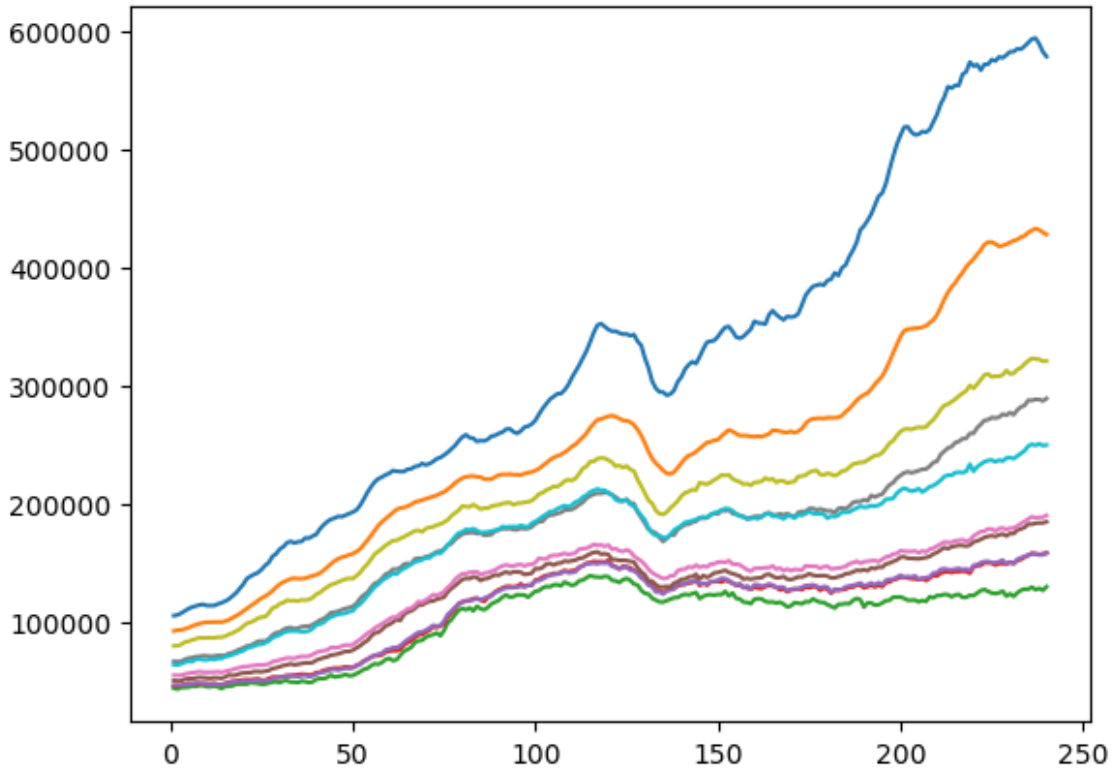
```
# The file has a header line
julia> (ukhpData,ukhpHead) =
        readdlm(ukhptsv, '\t'; header=true);

# Which is a 1x10 array.
julia> ukhpHead
1×10 Array{AbstractString,2}: "Inner London" "Outer London" "North East" …
```

```
"South East" "South West"

# The data portion is a 240x10 matrix ...
julia> (ukd1, ukd2) = size(ukhpData)
(240, 10)

# ... all of Floats
julia> typeof(ukhpData)
Array{Float64,2}
```

We can visualise this data using PyPlot:

```
julia> using PyPlot

julia> t = collect(1:ukd1)
julia> for i in 1:10
   plot(t,ukhpData[:,i])
end
```

Output:



As a further example lets use the DLM routine to read the CSV file of Apple (AAPL) stock prices.

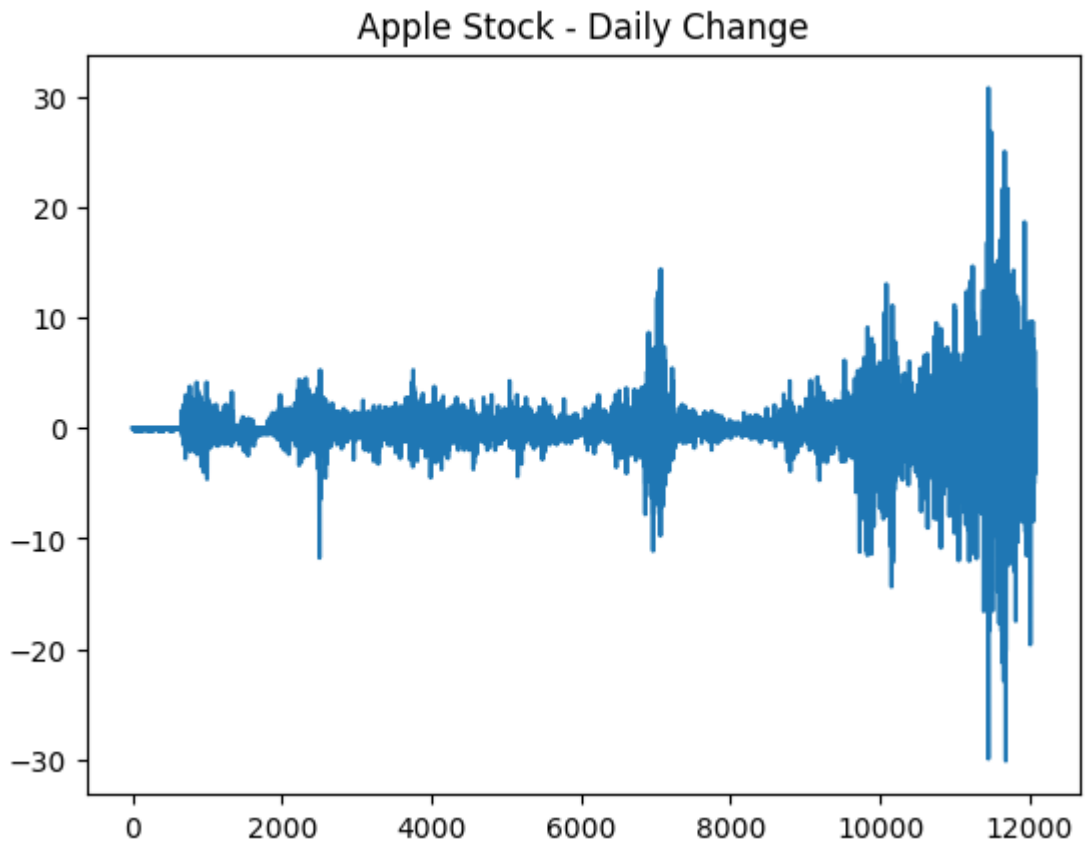
```
julia> aaplcsv = "Files/aapl.csv";
julia> isfile(aaplcsv)
julia> (aaplData,aaplHead) =
        readdlm(aaplcsv, ','; header=true);

julia> aaplHead
1×13 Array{AbstractString,2}: "Date"  "Open"  "High"  "Low"  "Close"  …
                "Adj. Close"  "Adj. Volume"

#=
Use a slice to display the first 10 lines and 6 columns. We have not sorted
the array so the data is in descending date order
=#
julia> aaplData[1:10, 1:6]
10×6 Array{Any,2}: "2013-12-31"  554.17  561.28  554.0   561.02  7.9673e6
"2013-12-30"  557.46  560.09  552.32  554.52  9.0582e6  "2013-12-27"
```

```
563.82  564.41  559.5   560.09  8.0673e6  "2013-12-26"  568.1   569.5
563.38  563.9   7.286e6   "2013-12-24"  569.89  571.88  566.03  567.67
5.9841e6  "2013-12-23"  568.0   570.72  562.76  570.09  1.79038e7
"2013-12-20"  545.43  551.61  544.82  549.02  1.55862e7 "2013-12-19"  549.5
550.0   543.73  544.46  1.14396e7 "2013-12-18"  549.7   551.45  538.8
550.77  2.02094e7 "2013-12-17"  555.81  559.44  553.38  554.99  8.2108e6
```

To plot this we can take each column and apply **`reverse()`** to each:

```
julia> using Dates, PyPlot

julia> d0 = Date("2000-01-01")
julia> aaplDate = reverse(Date.(aaplData[:,1]))
julia> aaplOpen = reverse(Float64.(aaplData[:,2]))
julia> aaplClose = reverse(Float64.(aaplData[:,5]))

julia> const NAAPL = length(aaplDate)
julia> aapl_days = zeros(Int64, NAAPL)

julia> d0 = aaplDate[1]
julia> dt = [(aaplDate[i]-d0).value for i = 1:NAAPL]

julia> aaplDifs = [(aaplClose[i] - aaplOpen[i]) for i = 1:NAAPL]

# Plot the daily difference between Open and Close
julia> plot(dt,aaplDifs)
julia> title("Apple Stock - Daily Change")
```

Output:



HDF5 and JLD files

HDF5 stands for Hierarchical Data Format v5. It was originally developed by the NCSA in the USA and is now supported by the HDF Group. It was developed to store large amounts of scientific data which are exposed to the user as groups and datasets which are akin to directories and files in a convention file system. Version 5 was developed to overcome some of the limitations of the previous version (v4) and Julia, in common with languages such as Python, R, Matlab/Octave, has extensions to be able to access files in HDF5 format.HDF5 also uses "attributes" to associate metadata with a particular group or dataset and ASCII names for these different objects. Objects can be accessed by UNIX-like pathnames, such as e.g., "/projects/juno/tests/first.h5" where "projects" is a top-level group, "juno" and "tests" are subgroups and first.h5 is a dataset.

You also need to have the HDF5 library installed on your system (version 1.8 or higher is required).

For most users no additional steps should be required; the HDF5 library should be installed automatically when packages is added.

Language wrappers for HDF5 may be viewed as either "low level" or "high level." The Julia package hdf5.jl contains both levels. At the low level, it directly wraps HDF5's functions, copying their API and making them available from within Julia. At the high level, it provides a set of functions which are built on the low-level wrapper in order to simplify the usage of the library. For simple types (scalars, strings, and arrays), HDF5 provides sufficient metadata to know how each item is to be interpreted while representing the data in a way that is agnostic of computing architectures.

Plain HDF5 files are created and/or opened in Julia with the h5open command:

fid = h5open(filename, mode)

where mode can be any one of the following:

- "r" : read-only
- "r+" : read-write, preserving any existing contents
- "w" : read-write, destroying any existing contents

This returns an object of type PlainHDF5File, a subtype of the abstract type HDF5File. "Plain" files have no elements (groups, datasets, or attributes) that are not explicitly created by the user.

Here is a quick snippet which uses HDF5 :

```
julia> h5file = "Files/mydata.h5"
julia> aa = [u + v*rand()
         for u = 0.5:0.5:10.0, v = 0.5:0.5:6.0]

julia> h5open(h5file, "w") do f
 write(f, "aa", aa)
end
```

Alternatively, we can say either of the following :

h5write(h5file,"aa",aa) or else *@write h5file aa*

This can be read back without the **`h5open()`** statement, similar to **`h5write()`** syntax in the TIP above.
Also we can create a slice of the file on-the-fly, without reading all the dataset into memory

```
julia> bb = h5read(h5file, "aa", (2:3:14, 4:3:10)
5×3 Array{Float64,2}: 2.49545  3.73017  3.55605 3.02282  3.69234  4.83512
4.57292  6.13872  5.82451 6.63335  7.49806  6.99261 8.67941  9.10232
8.12539
```

Julia Data Format

HDF5.jl package also provides the basis for a specific JLD (J*ulia Data format*) to accurately store and retrieve Julia variables.

While it is possible to use "plain" HDF5 for this purpose but the advantage of the JLD module is that it preserves meta-information such as the exact type of each variable.

At the end of the previous section we read in the Apple stock dataset and computed the differences between opening and closing stock. The following snippet will save this to a JLD file

```
julia> aapljld = "Files/aapldifs.jld";
julia> rm(aapljld, force=true)
#=
Removing the existing file is not strictly necessary, since the open for
"w" will achieve this
=#
julia> jldopen(aapljld, "w") do fid
  write(fid,"aaplDate",aaplDate)
  write(fid,"aaplClose",aaplClose)
  write(fid,"aaplDifs",aaplDifs)
end
julia> isfile(aapljld)
true
```

We can open the JLD file and read back ***SOME*** of the data:

```
julia> fid = jldopen(aapljld, "r")
Julia data file version 0.1.2:
Files/aapl.jld

julia> aaDate = read(fid, "aaplDate")
julia> aaDifs = read(fid, "aaplDifs")
julia> close(fid)
```

XML files

Alternative data representations are provided using XML and JSON.

We will consider the latter (JSON) in a subsequent chapter when when discussing networking, web and REST services.

In this section, we will look at XML file handling and the functionality available in the LightXML package. To assist in this, we will use the file books.xml, which contains a list of ten books.

The first portion of the file is :

```
<?xml version="1.0" encoding="UTF-8" ?>
<catalog>
  <book id="bk101">
    <author>Gambardella, Matthew</author>
    <title genre='Computing'>XML Developer's Guide</title>
     <price currency='GBP'>44.95</price>
     <publish_date>2000-10-01</publish_date>
     <description>An in-depth look at creating applications with
XML.</description>
 </book>
. . . . .
. . . . .
</catalog>
```

LightXML is a wrapper of libxml2, which provides a reasonably comprehensive high-level interface covering the most functionalities:

- Parse a XML file or string into a tree
- Access XML tree structure
- Create an XML tree
- Export an XML tree to a string or an XML file

I will cover parsing an XML file here, as it is probably the most common procedure.

```
#=
Both HDF5 and LightXML use routines called root and name; so to avoid a
clash, the calls below are fully qualified
=#
julia> using LightXML
julia> xdoc = parse_file("Files/books.xml");
julia> xtop = LightXML.root(xdoc);
julia> println(LightXML.name(xtop));
julia> catalog
```

```
# xdoc contains all the XML dataset
julia> xdoc
<?xml version="1.0" encoding="utf-8"?><catalog>   <book id="bk101">
<author>Gambardella, Matthew</author>      <title genre="Computing">XML
Developer's Guide</title>      <price currency="GBP">44.95</price>
<publish_date>2000-10-01</publish_date>      <description>An in-depth look
at creating applications with XML.</description>   </book>   <book
id="bk102">      <author>Ralls, Kim</author>      <title
genre="Fantasy">Midnight Rain</title>      <price
currency="GBP">5.95</price>      <publish_date>2000-12-16</publish_date>
<description>A former architect battles corporate zombies, an evil
sorceress, and her own childhood to become queen of the
world.</description>   </book>
. . . . .
. . . . .
</catalog>
```

The following code navigates through all the *child nodes* of *xtop* outputting the the titles and the genre into which they have been classified.

```
julia> using Printf
julia> for c in child_nodes(xtop)
  if is_elementnode(c)
    e = XMLElement(c)
    t = find_element(e, "title")
    title = content(t)
    genre = attribute(t, "genre")
    @printf "\n%28s -:- %s" title genre
   end
end

       XML Developer's Guide -:- Computing               Midnight Rain -:-
Fantasy             Maeve Ascendant -:- Fantasy             Oberon's Legacy
-:- Fantasy          The Sundered Grail -:- Fantasy                 Lover
Birds -:- Romance               Splish Splash -:- Romance
Creepy Crawlies -:- Horror               Paradox Lost -:- SciFi .NET: The
Programming Bible -:- Computing
```

Next, we look for all the computing books and print out the full details.

The publication date is in the format YYYY-MM-DD so I've used the Dates module (in STDLIB) to create a more readable string.

```
julia> using Dates
julia> for c in child_nodes(xtop)
   if is_elementnode(c)
      e = XMLElement(c)
      t = find_element(e, "title")
```

```
      genre = attribute(t, "genre")
      if genre == "Computing"
         a = find_element(e,"author")
         p = find_element(e,"price")
         curr = attribute(p, "currency")
         d = find_element(e,"publish_date")
         dc = DateTime(content(d))
         ds = string(day(dc)," ",monthname(dc)," ",year(dc))
         desc = find_element(e,"description")
         println("Title: ", content(t))
         println("Author: " ,content(a))
         println("Date: " ,ds)
         println("Price: " ,p ," (", curr, ")")
         println(content(desc),"\n");
      end
   end
end
```

Finally let's using the genre to select all the books which have been categorised as *"Computing"*.

```
julia> using Dates
julia> for c in child_nodes(xtop)
   if is_elementnode(c)
      e = XMLElement(c)
      t = find_element(e, "title")
      genre = attribute(t, "genre")
      if genre == "Computing"
         a = find_element(e,"author")
         p = find_element(e,"price")
         curr = attribute(p, "currency")
         d = find_element(e,"publish_date")
         dc = DateTime(content(d))
         ds = string(day(dc),
            " ",monthname(dc)," ",year(dc))
         desc = find_element(e,"description")
         println("\nTitle: ", content(t))
         println("Author: " ,content(a))
         println("Date: " ,ds)
         println("Price:",p," (", curr, ")")
         println(content(desc),"\n");
      end
   end
end

Title:      XML Developer's GuideAuthor:     Gambardella, MatthewDate:
1 October 2000Price:      44.95 (GBP)An in-depth look at creating
```

```
applications with XML.Title:      .NET: The Programming BibleAuthor:
O'Brien, TimDate:       9 December 2000Price:      36.95 (GBP)Microsoft's
.NET initiative is explored in detail in this deep programmer's reference.
```

Data Frames and Statistics

We were introduced to Julia's implementation of data frames in the previous section and also used the availability of a series of datasets, first made available by the Comprehensive R Archive network (CRAN), hence the epithet R-Datasets.

A full listing can be obtained from the `R-Datasets page` and also from the package maintainer, Vincent Arel-Bundock's `github page`.

The equivalent package in Python is Pandas, of which there is also a Julia package (Pandas.jl), which a wrapper around the Python one, available via the JuliaPy github page.

When dealing with tabulated datasets there are occasions when some of the values are missing and it is one of the features of statistical languages is that they can handle such situations

Support for this has been changed in version 1.0 by the introduction of the *Missings.jl* package (via the *JuliaData* group).

Data Frames

The Data Frame (DF) is one of the cornerstones of Julia. Implementations go back to the very early days of Julia but the current version is a rewrite, totally in native code and many packages handle data in a data frame as easily as if in a plain array.

In essence a DF is a matrix where the columns are all of the same type but may be different from each other, and which may be referenced by name. The analogy would be a sheet in an Excel (or similar) workbook.

In this chapter we will look at datasets which return a data frame and packages which can be used to process them. In chapter 9, I will discuss how to get datasources from the web and transform these into a DF.

Data Arrays

In the past the basis of the data frame as the data array, provided by the package

DataArrays.jl. However in version 0.7, this package was depreciated and hence is removed in version 1.0

The package provided a type (DataArray) which would work efficiently with data containing missing values encapsulated via the Missings package; however now the advice is to use the construct Array{Union{T.Missing}} instead.

in Julia. missing is actually very similar to its predecessor NA, but it with significant improvements:

- Missing values are safe by default: when passed to most functions, they either propagate or throw an error.
- The missing object can be used in combination with any type, be it defined in Base, in a package or in user code.
- Standard Julia code working with missing values is efficient, without special rules.

For a full discussion the reader is referred to the following `blog post`.

R-Datasets

As mentions above the R Datasets is maintained by Vincent Arel-Bundock, and the RDatasets.jl is a package which provides (most) of these.

Datasets are 'lumped' together of, some 33, packages and a list can be created by using the **`packages()`** routine

```
julia> using RDatasets
julia> RDatasets.packages()
33×2 DataFrame
| Row | Package | Title
|     | String  | Union{Missing, String}
|-----|---------|-----------------------------------------|
| 1   | COUNT   | Functions, data and code for count data.
| 2   | Ecda    | Data sets for econometrics
| 3   | HSAUR   | A Handbook of Statistical Analyses Using R (1st Edition)
| 4   | HistData | Data sets from the history of statistics and data
visualization
| 5   | ISLR    | An Introduction to Statistical Learning with
Applications in R
| 6   | KMsurv  | Data sets from Klein and Moeschberger (1997), Survival
Analysis
| 7   | MASS    | Support Functions and Datasets for Venables and Ripley's
MASS
```

```
. . . . . .
. . . . . .
```

Passing a package name to the datasets function as **`RDatasets.datasets("MASS")`** will output a listing of the individual datasets in the package, including a description (title) and the number of rows and columns.

Note that the routine is not exported by RDatasets, so the call must be fully qualified *(see the notebook for an example).*

We will start by picking up some data on the frequency and severity of `earthquakes around Fiji` from the R-Datasets package usin the following:

```
julia> using DataFrames, RDatasets
julia> quakes = dataset("datasets", "quakes");

# Just display the first 5 rows.
julia> quakes[1:5,:]
```

Output:

	Lat	Long	Depth	Mag	Stations
	Float64⍰	Float64⍰	Int64⍰	Float64⍰	Int64⍰
1	-20.42	181.62	562	4.8	41
2	-20.62	181.03	650	4.2	15
3	-26.0	184.1	42	5.4	43
4	-17.97	181.66	626	4.1	19
5	-20.42	181.96	649	4.0	11

To do some meaningful statistics on this data we will need to access to some additional routines; strangely some of the basic ones are in the STDLIB package Statistics, while others (although relatively common) are in StatsBase, the latter which will need to be added via the package manager and both *'used'*.

```
#=
As a reminder, here is how to add a package using the new Pkg3 API; it
needs to be done only once and the REPL interactive shell can be used as an
altenative
=#

julia> using Pkg
julia> pkg.add("StatsBase")
```

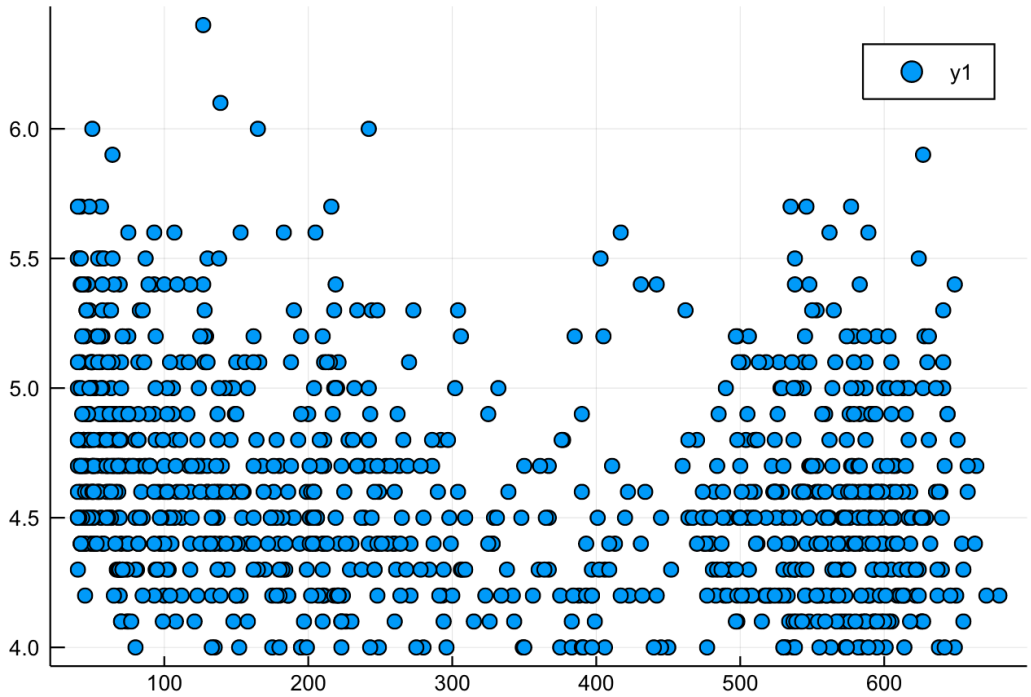
```
#=
At the same time you may also wish to add a series of other statistical
packages:
 • Distributions
 • KernelDensity
 • HypothesisTests
 • GLM (aka General Linear Models)
which we will need later in the chapter.
=#

julia> using Statistics, StatsBase

# Pickup the magnitude and depth of the quakes
mags  = Float64.(quakes[:Mag]);
depth = Float64.(quakes[:Depth]);

# See is there is any correlation between them.

julia> cor(mags, depths)
-0.23063769768765782
```

So a weak negative one, which is a little counter-intuitive; personally I would have expected deeper quakes to be more powerful, possibly the power is absorbed by the surrounding rock.

Visualising the data may be useful:

```
# Use the Plots API and the GR backend
# Other packages will do as well.

using Plots
gr()            # Set the GR backend

scatter(depth,mags)
```

Output:



So no obvious trends, there seem to be two clusters, a deep water and a shallow water one, and for the latter the largest quakes have occurred.

We can use the **`describe()`** routine to look at some summary statistics on quake magnitudes, (the granularity of the data is only to 1 dec. place)

```
describe(mags)
Summary Stats:
Mean:           4.620400
Minimum:        4.000000
1st Quartile:   4.300000
Median:         4.600000
3rd Quartile:   4.900000
Maximum:        6.400000
Length:         1000
Type:           Float64
```

The summary statistics (above) includes the mean, but not the variance, skew, etc. For some of these we require using the StatsBase package:

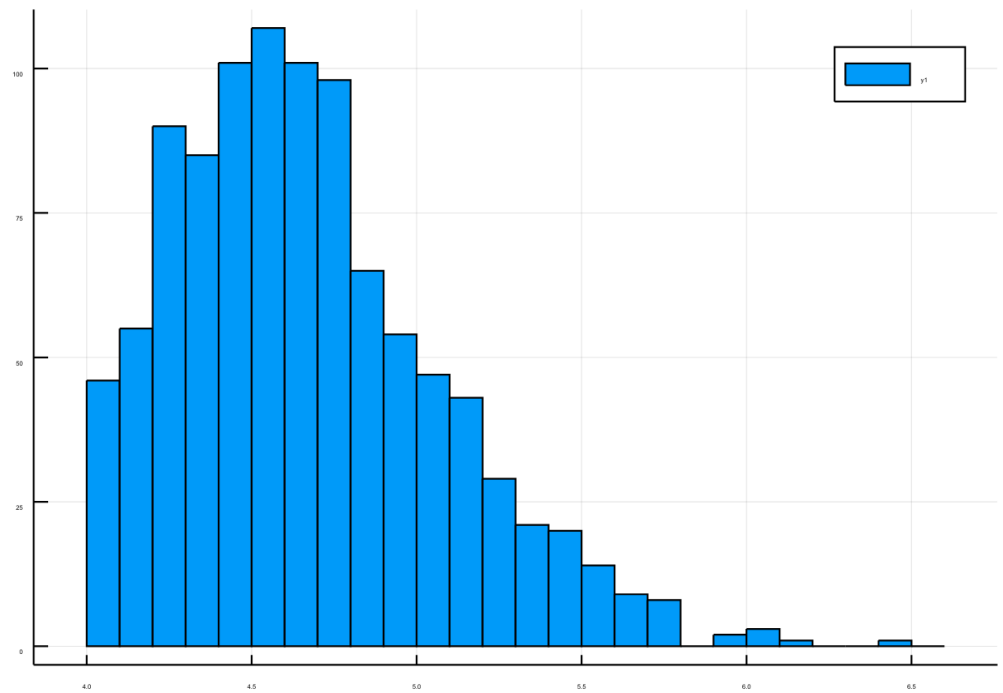
```
julia> mags = Float64.(quakes[:Mag]);
```

```
julia> (m1, m2, m3, m4) = map(x -> round(x,digits=4),
 [mean(mags), std(mags), skewness(mags), kurtosis(mags)]);

julia> using Printf
julia> @printf "\nMean: %.4f\nStdV: %.4f\nSkew: %.4f\nKurt: %.4f\n" m1 m2
m3 m4

Mean: 4.6204StdV: 0.4028Skew: 0.7686Kurt: 0.5103
```

And we can easily display a histogram of the quake magnitudes:

```
julia> histmag = fit(Histogram, mags,
                     4.0:0.1:6.6, closed=:left)
Histogram{Int64,1,Tuple{StepRangeLen{Float64,Base.TwicePrecision{Float64},B
ase.TwicePrecision{Float64}}}}edges: 4.0:0.1:6.6weights: [46, 55, 90, 85,
101, 107, 101, 98, 65, 54  …  9, 8, 0, 2, 3, 1, 0, 0, 1, 0]closed:
leftisdensity: false

julia> plot(histmag)
```

Output:



(*Some*) Statistics

Data frames are especially useful in the new compendium discipline commonly termed data science. Both Python and R are frequently seen as it's cornerstones but with the new application of Julia's DataFrames modules, extensive plotting options (chapter 8) and the addition of the parallel analytical engine JuliaDB (chapter 9), Julia presents a really exciting (and fast) alternative.

In this current section we will look at application of some simple statistics involving datasources from the R-Datasets package.

```
julia> mlmf = dataset("mlmRev","Gcsemv"); size(mlmf)
(1905, 5)
```

We will use data from *mlmRev* which is a group of datasets from the *Multilevel Software Review*, the dataset *Gcsemv* refers to UK's GSCE exam scores.

This covers the results from 73 schools both in examination and course work, the data is not split by subject (only school and pupil) but the gender of the student is provided. Schools are listed via a categorical variable.

```
# Display the first 5 rows
julia> mlmf[1:5,:]
```

	School	Student	Gender	Written	Course
	Categorical…	Categorical…	Categorical…	Float64⍰	Float64⍰
1	20920	16	M	23.0	missing
2	20920	25	F	missing	71.2
3	20920	27	F	39.0	76.8
4	20920	31	F	36.0	87.9
5	20920	42	M	16.0	44.4

```
# We can use describe to output the summary statistics
julia> describe(mlmf)
```

variable	mean	min	median	max	nunique	nmissing	eltype
Symbol	Union…	Any	Union…	Any	Union…	Union…	DataType
1 School		20920		84772	73		CategoricalString{UInt8}

	variable	mean	min	median	max	nunique	nmissing	eltype
	Symbol	Union…	Any	Union…	Any	Union…	Union…	DataType
2	Student		1		5521	649		CategoricalString{UInt16}
3	Gender		F		M	2		CategoricalString{UInt8}
4	Written	46.3652	0.6	46.0	90.0		202	Float64
5	Course	73.3874	9.25	75.9	100.0		180	Float64

We see that there are 73 schools and 649 students; because of the fact that these are categorical values, statistics such as min/max, median and mean have not relevance.

The written and course work values refers to both genders (M and F) and both sets contain missing values, so we need to collect all the records with a value and split these by gender.

We will concentrate just on the written (exam) scores, those for the course work will be left for the reader.

```
julia> writtenF =
    collect(skipmissing(mlmf[mlmf.Gender .== "F", :Written]));
julia> writtenM =
    collect(skipmissing(mlmf[mlmf.Gender .== "M", :Written]));
```

First let us calculate the mean and standard deviation of the two groups

```
julia> (μWM, μWF) =
    round.((mean(writtenM), mean(writtenF)), digits=3)
(48.286, 45.005)

julia> (σWM, σWF) =
    round.((std(writtenM), std(writtenF)),digits=3)
(12.905, 13.535)
```

We can use these to apply a Students t-test between the means, with the (NULL) hypothesis that they are drawn from the same population

```
# We will need the numbers in each subset
julia> (nWM, nWF) = (length(writtenM), length(writtenF))
(706, 997)

# And evaluate a completed standard deviation
julia> σW = sqrt((σWM*σWM)/(nWM - 1) + (σWF*σWF)/(nWF - 1));

# The compute the t-statistic
julia> tt = round(abs(σWM - σWF)/σW , digits=4)
0.9726

#=
Looking at t-tables p ~ 0.33; 95% ~ 0.06, 90% ~ 0.13
```

```
On the basis of these we reject the null hypothesis and  assert that there
is a statistical difference between the means

=#
```

Kernel Densities

The summary statistics above there is clearly a significant difference between marks for coursework and examination and also compute the kernel densities of the two groups.

Kernel density is a technique to create a smooth curve given a set of data.

This can be useful if you want to visualize just the "shape" of the dataset, as an analog of a discrete histogram for continuous data.

To analyse these we will need to extract all the records of students and have scores in ***BOTH*** categories; this we do by using the routine **`completecases()`**

```
using RDatasets, KernelDensity
mlmf = dataset("mlmRev", "Gcsemv");
df = mlmf[completecases(mlmf[[:Written, :Course]]), :]
```

	School Categorical…	Student Categorical…	Gender Categorical…	Written Float64?	Course Float64?
1	20920	27	F	39.0	76.8
2	20920	31	F	36.0	87.9
3	20920	42	M	16.0	44.4
4	20920	101	F	49.0	89.8
5	20920	113	M	25.0	17.5
6	22520	1	F	48.0	84.2

We need to extract the value from the DataFrame, as the element type is (still) the union of a Float64 and Missing; for convenience let define a macro to operate on the array

```
macro F64(sym)
  quote
    Float64.(skipmissing(Array($sym)))
   end
end
```

Look at the coursework; calculate the kernel density and output the summary statistics

```
julia> dc = @F64 df[:Course];
julia> kdc = kde(dc);
```

```
julia> summarystats(dc)
Summary Stats:Mean:           73.381385Minimum:        9.2500001st
Quartile:   62.900000Median:         75.9000003rd Quartile:
86.100000Maximum:        100.000000
```

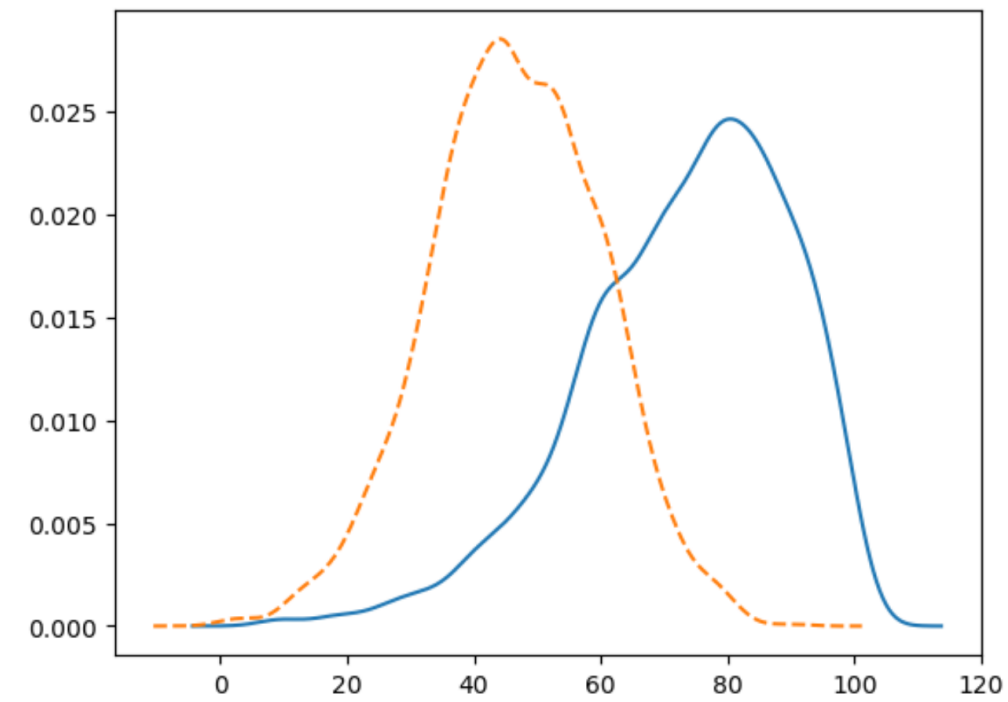
Now repeat the same for the written marks.

```
julia> dW = @F64 df[:Written];
julia> kdw = kde(dw);

julia> summarystats(dw)Summary Stats:Mean:           46.502298Minimum:
0.6000001st Quartile:   38.000000Median:         46.0000003rd Quartile:
56.000000Maximum:        90.000000
```

And finally display the kernel densities for the two groups, i.e. coursework and written

```
julia> using PyPlot
julia> PyPlot.plot(kdc.x, kdc.density)
julia> PyPlot.plot(kdw.x, kdw.density, linestyle="--")
```

Output:



Testing hypothesis

So far we have looked at differences between written and coursework but not differentiated between various schools. Recall that a categorical variable is provided which references the school and this can be used create subset of the dataframe.

To reduce the length of the results, we will restrict the analysis to schools with more than 40 students.

```
julia> for subdf in groupby(df, :School)
  (size(subdf)[1] > 40) &&
  let
    sch = subdf[:School][1]
    msw = mean(subdf[:Written])
    msc = mean(subdf[:Course])
    nsz = size(subdf)[1]
    @printf "%10s : %8.4f %8.4f %3d\n" sch msw msc nsz
  end
end

 22520 :  35.4482  57.4580  56 60457 :  53.4773  85.9568  44 68107 :
44.9107  74.6750  56 68125 :  47.1556  77.5322  45 68137 :  28.2807
62.5373  83 68411 :  40.4615  59.4369  65 68809 :  42.7705  71.1115  61
```

Lets take two of these schools (#68107 and #68411) and investigate the difference in scores from each. Again we will apply a T-test but will using a package from JuliaStats

```
julia> using HypothesisTests
julia> df68107 = mlmf[mlmf[:School] .== "68107", :];
julia> df68107cc =
    df68107[completecases(df68107[[:Written, :Course]]), :];

julia> df68411 = mlmf[mlmf[:School] .== "68411", :];
julia> df68411cc =
     df68411[completecases(df68411[[:Written, :Course]]), :];
```

First apply the test to written marks.

```
julia> df68107wri = @F64 df68107cc[:Written];
julia> df68411wri = @F64 df68411cc[:Written];
julia> UnequalVarianceTTest(df68107wri, df68411wri)

Two sample t-test (unequal variance)-----------------------------------
Population details:    parameter of interest:   Mean difference    value
under h_0:         0    point estimate:          4.449175824175825    95%
confidence interval: (-0.1837, 9.082)Test summary:    outcome with 95%
confidence: fail to reject h_0    two-sided p-value:
0.0596Details:    number of observations:   [56,65]    t-statistic:
```

```
1.9032531870995715    degrees of freedom:       109.74148002018097
empirical standard error: 2.337668920946911
```

And then repeat for the coursework

```
julia> df68107cou = @F64 df68107cc[:Course];
julia> df68411cou = @F64 df68411cc[:Course];
julia> UnequalVarianceTTest(df68107cou, df68411cou)

Two sample t-test (unequal variance)-----------------------------------
Population details:    parameter of interest:   Mean difference    value
under h_0:         0    point estimate:          15.238076923076903    95%
confidence interval: (10.6255, 19.8506)Test summary:    outcome with 95%
confidence: reject h_0    two-sided p-value:           <1e-8Details:
number of observations:   [56,65]    t-statistic:
6.541977424916656    degrees of freedom:       118.13175559744462
empirical standard error: 2.329276904111456
```

We can see that we reject the null hypothesis for coursework but not for written; which would seem to imply that there is significant difference in marking between these two schools in the former case.

Linear Regression

To close this section let us look at applying a regression analysis some data from these two schools. Linear regression, using least squares is well supported, but JuliaStats provide a package GLM, to implement a set of generalised methods.

To proceed we will need to have equal sizes of dataset and to rank (i.e. sort) these. Using the two schools we examined above, we see that both have student numbers above 50. So we can sample, randomly, at that value and then sort the resultant data.

```
julia> using GLM
julia> dw68411ss = sort(sample(df68411wri,50));
julia> dw68107ss = sort(sample(df68107wri,50));
julia> dc68411ss = sort(sample(df68411cou,50));
julia> dc68107ss = sort(sample(df68107cou,50));
```

The 'ss' suffices correspond to the sorted-samples data and the first thing to look at is if there is any correlation between results from each school

```
julia> macro rdup(val, dgt)
  quote
    round.($val, digits=$dgt)
  end
end
```

```
julia> @rdup cor(dw68107ss, dw68411ss) 3
0.968

julia> @rdup cor(dc68107ss, dc68411ss) 3
0.967
```

Apparently there is a high degree of correlation between these *sorted* data.

To fit a linear model we will need to combine the values from the two schools into a single datagrams using the **`hcat()`** routine.

```
julia> dwf = convert(DataFrame, hcat(dw68107ss, dw68411ss))
julia> names!(dwf, [:s68107, :s68411])
dwf[1:5,:]
```

	s68107 Float64	s68411 Float64
1	18.0	16.0
2	22.0	23.0
3	22.0	28.0
4	22.0	29.0
5	22.0	29.0

```
julia> dcf =
        convert(DataFrame, hcat(dc68107ss, dc68411ss))
julia> names!(dcf, [:s68107, :s68411])
julia> dcf[1:5, :]
```

	s68107 Float64	s68411 Float64
1	47.2	32.4
2	47.2	32.4
3	50.0	32.4
4	55.5	32.4
5	55.5	37.0

So now we can apply the **`fit()`** from the GLM package

```
julia> lm1 = fit(LinearModel, @formula(s68107 ~ s68411), dwf)

StatsModels.DataFrameRegressionModel{LinearModel{LmResp{Array{Float64,1}},D
ensePredChol{Float64,LinearAlgebra.Cholesky{Float64,Array{Float64,2}}}},Arr
ay{Float64,2}}Formula: s68107 ~ 1 + s68411Coefficients:
Estimate Std.Error  t value Pr(>|t|)(Intercept)  -4.79368   1.81815
-2.63657   0.0112s68411        1.17445 0.0439365  26.7307   <1e-29
```

```
julia> lm2 = fit(LinearModel, @formula(s68107 ~ s68411),
dcf)StatsModels.DataFrameRegressionModel{LinearModel{LmResp{Array{Float64,1
}},DensePredChol{Float64,LinearAlgebra.Cholesky{Float64,Array{Float64,2}}}}
,Array{Float64,2}}Formula: s68107 ~ 1 + s68411Coefficients:
Estimate Std.Error t value Pr(>|t|)(Intercept)   19.7128   2.10054 9.38461
<1e-11s68411       0.945694 0.0359333 26.3181   <1e-29
```

Summary

In this chapter we looked at how to handle data, primarily in the form of (text-based) disk files or from Julia modules such as the RDatasets module.

We saw how Julia implements the "R" style dataframe, *aka Pandas in Python,* and its sibling the timearray; how to visualise the datasets and further to analyse the values by the application of a set of simple statistical routines.

In a subsequent chapter we will discuss dealing with data contained within databases and other sources available over the internet.

Index