Table of Contents

1 Developing in Julia

Julia is a feature-rich language. It was designed to appeal to the novice programmer and purist alike. Indeed for those whose interest lie in data science, statistics and mathematical modeling, Julia is well equipped to meet all their needs.

Our aim is to furnish the reader with the necessary knowledge to begin programming in Julia almost immediately. So rather than begin with an overview of the language's syntax, control structures and the like, we will introduce Julia's facets gradually over the rest of this book. Over the next four chapters we will look at some of the basic and advanced features of the Julia core. Many of the features such as graphics and database access, which are implemented via the package system will be left until later when discussing more specific aspects of programming Julia.

If you are familiar with programming in Python, R, MATLAB and so on, you will not find the journey terribly arduous, in fact we believe it will be a particularly pleasant one.

This chapter will cover:

Simple data types

Lead ins

Integers, bits, bytes, and bools

Julia is a strongly typed language allowing the programmer to specify a variable's type precisely. However in common with most interpreted languages it does not require the type to be declared when a variable is declared, rather it infers it from the form of the declaration.

A variable in Julia is any combination of upper or lowercase letters, digits and the underscore (_) and exclamation (!) characters. It must start with a letter or an underscore.

Conventionally variable names consist of lowercase letters with long names separated by underscores rather than using camel case.

To determine a variable type we can use the **`typeof()`** function.

So typically:

```
julia>  x = 2;    typeof(x)  #  =>  gives Int64
julia>  x = 2.0;  typeof(x) #  =>  gives Float64
```

Notice that the type (see the preceding code) starts with a capital letter and ends with a number which indicates the number of bit length of the variable. The bit length defaults to the word length of the operating system and this can be determined by examining the built-in constant `WORD_SIZE`.

```
julia> WORD_SIZE  # => 64  (on my MacPro computer)
```

In this section we will be dealing with integer and boolean types.

Integers

The integer type can be any of **`Int8, Int16, Int32, Int64 and Int128`**, so the maximum integer can occupy 16 bytes of storage and be anywhere within in the range from**`-2^127 to (+2^127 - 1)`**.

If we need more precision than this Julia core implements the *BigInt* type:

```
julia> x = BigInt(2^32)
6277101735386680763835789423207666416102355444464034512896
```

There are a few more things to say about integers:

As well as the integer type, Julia provides the unsigned integer type **`UInt`**; again **`UInt`** ranges from 8 to 128 bytes, so the maximum**`UInt`** is *(2^128 - 1)*.

We can use the **`typemax()`** and **`typemax()`** functions to output the ranges of the **`Int`** and **`UInt`** types.

```
julia> for T =
Any[Int8,Int16,Int32,Int64,Int128,UInt8,UInt16,UInt32,UInt64,UInt128]
  println("$(lpad(T,7)): [$(typemin(T)),$(typemax(T))]")
end
     Int8: [-128,127]
     Int16: [-32768,32767]
     Int32: [-2147483648,2147483647]
```

```
Int64: [-9223372036854775808,9223372036854775807]
Int128: [-170141183460469231731687303715884105728,
          170141183460469231731687303715884105727]
UInt8: [0,255]
UInt16: [0,65535]
UInt32: [0,4294967295]
UInt64: [0,18446744073709551615]
UInt128: [0,340282366920938463463374607431768211455]
```

Particularly notice the use of the form of the *for* statement which we will discuss when we deal with arrays and matrices later in this chapter.

Suppose we type:

```
julia> x = 2^32; x*x   # => the answer 0
```

The reason is that integer overflow 'wraps' around, so squaring 2^32 gives 0 not 2^64 since my *WORD_SIZE* is 64.

```
julia> x = int128(2^32); x*x  # => the answer we would expect
18446744073709551616
```

We can use the `typeof()` function on a type such as `Int64` in order to see what its parent type is.

```
# So typeof(Int64) gives DataType and typeof(UInt128) also gives DataType.
```

The definition of *DataType* is 'hinted' at in the core file *boot.jl*; hinted at because the actual definition is implemented in C and the Julia equivalent is commented out.

The definitions of the integer types can also be found in *boot.jl*, this time not commented out.

In the next chapter we will discuss the Julia type system in some detail. here it is worth noting that we distinguish between two kinds of datatypes: abstratc and primative (concrete).

The general syntaxt for declating an abstract type is

```
abstract type «name» end
abstract type «name» <: «supertype» end
```

Typically:

```
abstract type Number end
abstract type Real <: Number end
abstract type AbstractFloat <: Real end
abstract type Integer <: Real end
abstract type Signed <: Integer end
abstract type Unsigned <: Integer end
```

Where the *<:* operator corresponds to a subclass of the parent .

If we type:

```
julia> x = 7; y = 5; x/y #  => this gives 1.4
```

So division of two integers produces a real result. In interactive mode we can use the symbol *ans* to correspond to the last answer, that is, *typeof(ans)* gives *Float64*.

To get the integer divisor we use the function *div(x,y)* which gives 1 as expected and *typeof(ans)* is *Int64*. The remainder is obtained either by *rem(x,y)* or by using the % operator.

Julia has one curious operator the backslash, syntactically *x\y* is equivalent to *y/x*. So with x and y as before *x\y* gives 0.71428 (to 5 decimal places).

Primitive types

A primitive type is a concrete type whose data consists of a series of bits. Examples of primitive types are the (well-known) integers and floating-point values which we have met previously.

The general syntaxes for declaring a primitive type is similar to that of an abstract type but with the addition of the number of bits to be allocated:

```
primitive type «name» «bits» end
primitive type «name» <: «supertype» «bits» end
```

Since Julia is written (mostly) in Julia, a corollory is that Julia lets you declare your own primitive types, rather than providing only a fixed set of built-in ones.

That is all the standard primitive types are all defined in *Base* itself:

```
primitive type Float16 <: AbstractFloat 16 end
primitive type Float32 <: AbstractFloat 32 end
primitive type Float64 <: AbstractFloat 64 end

primitive type Bool <: Integer 8 end
primitive type Char 32 end

primitive type Int8 <: Signed 8 end
primitive type UInt8 <: Unsigned 8 end
primitive type Int16 <: Signed 16 end
primitive type UInt16 <: Unsigned 16 end
primitive type Int32 <: Signed 32 end
primitive type UInt32 <: Unsigned 32 end
primitive type Int64 <: Signed 64 end
primitive type UInt64 <: Unsigned 64 end
primitive type Int128 <: Signed 128 end
primitive type UInt128 <: Unsigned 128 end
```

Note that only sizes that are multiples of 8 bits are supported, so boolean values, although they really need just a single bit, cannot be declared to be any smaller than eight bits.

Logical and arithmetic operators

As well as decimal arguments it is possible to assign binary, octal, and hexadecimal ones using the prefixes *0b, 0o* and *0x*.

So *x = 0b110101* creates the hexadecimal number 0x35 (i.e., decimal 53) and *typeof(ans)* is *UInt8*, since 53 will 'fit' into a single byte.

For larger values the type is correspondingly higher, i.e. *x = 0b1000010110101* gives *x = 0x10b5* and *typeof(ans)* is *UInt16*.

When operating on bits Julia provides the following: ~ *(not)*, | *(or)*, *& (and)* and *$ (xor)*.

```
julia> x = 0xbb31; y = 0xaa5f; x$y    # => 0x116e
```

Also we can perform arithmetic shifts using**`<< (LEFT)`** and**`>>(RIGHT)`** operators. Note because x is of type**`UInt16`** the shift operator retains that size, so:

```
julia> x = 0xbb31;  x<<8
```

This gives *0x3100* (the top two nibbles being discarded) and *typeof(ans)* is *UInt16*.

Arithmetic shifts preserve the sign bit. This is not relevant when dealing with unsigned integers but bitwise operators can be applied to signed integers too. Logical and arithmetic left shifts produce the same result but right shifts do not. In this case Julia provides the >>> operator to apply a right logical shift.

```
julia> x = 0xbb31; y = int16(x)  # => 17615
```

```
y >> 4 gives -1101 and y >>> 4 gives 2995.
```

Booleans

Julia has the logical type *Bool*. Dynamically a variable is assigned a type *Bool* by equating it to the constant true or false (both lowercase) or to a logical expression such as:

```
julia> p = (2 < 3) # => true
julia> typeof(p)   # => Bool
```

Many languages treat 0, empty strings, NULLS as representing false and anything else as true. This is NOT the case in Julia however, there are cases where a Bool value may be promoted to an integer in which case true corresponds to unity.

That is, an expression such as $x + p$ (where x is of type *Int* and p of type *Bool*) will:

```
julia> x = 0xbb31; p = (2 < 3); x + p     # => 0x000000000000bb32
julia> typeof(ans) # => UInt64
```

Big integers

If we consider the factorial function defined by the usual recursive relation:

```
# n! = n*(n-1)!  for integer values of n (> 0)

function fac(n::Integer)
  @assert n > 0
  (n == 1) ? 1 : n*fac(n-1)
end
```

However note that since normally integers in Julia *overflow* (a feature of LLVM) an then the above definition can lead to problems with large values of ***'n'***

```
using Printf
for i = 20:30
    @printf "%3d : %d\n" i fac(i)
end
 20 : 2432902008176640000
 21 : -4249290049419214848
 22 : -1250660718674968576
 23 : 8128291617894825984
 24 : -7835185981329244160
 25 : 7034535277573963776
 26 : -1569523520172457984
 27 : -5483646897237262336
 28 : -5968160532966932480
 29 : -7055958792655077376
 30 : -8764578968847253504
```

```
# Since a BigInt <: Integer,
# if we pass a BigInt the routine returns the correct value
julia> fac(big(30))
265252859812191058636308480000000

# See can check this since integer values: Γ(n+1)  ===  n!
julia> gamma(31)
2.6525285981219107e32
```

Arrays

An array is an indexable collection of (normally) heterogeneous values such as integers, floats, booleans. In Julia unlike many programming languages the index starts at 1 not 0.

```
A = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610];
```

These are the first 15 numbers of the well-known Fibonacci sequence defined for positive integers by the recurrence relation:

fib(1) = fib(2)= 1
fib(n) = fib(n-1) + fib(n-2)

In conjunction of loops in the Asian option example in the previous chapter we meet the definition of a range as: start:[step]:end

```
julia> A = 1:10; typeof(A)
UnitRange{Int64}

julia> B = 1:3:15; typeof(B)
StepRange{Int64,Int64}

julia> C = 0.0:0.2:1.0; typeof(C)
StepRangeLen{Float64,Base.TwicePrecision{Float64},Base.TwicePrecision{Float
64}}
```

In Julia the above definition return a range type.
To convert a range to an array we can use the **`collect()`** function:

```
julia> C = 0.0:0.2:1.0; collect(C)
6-element Array{Float64,1}:
0.0
0.2
0.4
0.6
0.8
```

```
1.0
```

Julia also provides functions such as zeros(), ones() and rand() which provide array results. Normally these are returned a float-point values so a little bit of work is required to provide integer results.

```
julia> A = int(zeros(15));
julia> B = int(ones(15));
julia> C = rand(1:100,15);
```

Another method of creating and populating an array is by using a list comprehension. If we recall the first example of the Fibonacci series, we can get the same result but creating an uninitialised array of 15 integers by using Array(Int64,15); then assigning the first couple of values and use the definition of the Fibonacci series (above) to create the remaining values.

```
# Create integer array of 15 (undefined) elements
julia> A = Array{Int64,1}(undef,15);

# Add some values
julia  A[1] = 0; A[2] = 1;
julia> [A[i] = A[i-1] + A[i-2] for i = 3:length(A)]
```

Finally it is possible to create a completely empty array by using Array{Int64,1}(undef,0). Since arrays are immutable this would see a little futile but certain functions can be used to alter status of the array. In particular the push!() function can add a value to the array and increase its length by one.

```
julia> A = Array{Int64,1}(undef,0);
push!(A,1); push!(A,2); push!(A,3); etc. => results in A = [1,2,3]
```

The corresponding **`pop!(A)`** function will return the value 3, and result in A = [1,2].

Note: the use of the tailing **`!`** borrows the syntax form lisp-like conventions and is purely arbitrary.

Since functions are first class variables this is the reason that **`!`** is an accepted character in variable names but its a good idea to respect the convention and not use **`!`** in reference to common variables.

Consider the following two array definitions:

```
julia> A = [1,2,3];
3-element Array{Int64,1}
julia> A = [1 2 3];
1x3 Array{Int64,2}
```

The first, with values separated by commas, produces the usual 1-dimensional data structure; the second, where there are no commas, produces a matrix or 1 row and 3 columns, hence the definition: 1x3 Array{Int64,2}

To define more rows we separate the values with semi-colons as :

```
julia> A = [1 2 3; 4 5 6]
2x3 Array{Int64,2}
 1 2 3
 4 5 6
```

If we type:

```
for i in (1:length(A))
  @printf("%d \t%d\n", i, A[i]);
end
1    1
2    4
3    2
4    5
5    3
6    6
```

In Julia indexing in is in column order and the array/matrix can be indexed as 1-dimensional or 2-dimensional

```
julia> # A[1,2] is 2 and A[2] # is 4 and A[5] is 3
```

In fact it is possible to reshape the array to change it from a 2 x 3 matrix to a 3 x 2 one

```
julia> B = reshape(A,3,2)
3x2 Array{Int64,2}:
 1  5
 4  3
 2  6
```

Fibonacci sequences

We saw above that the Fibonacci sequence can be defined by the recurrence relation:

```
A = Array{Int64}(undef,15);
A[1]=1; A[2]=1;
[A[i] = A[i-1] + A[i-2] for i = 3:length(A)];
```

This presents a similar problem in as much as eventually the value of the Fibonacci sequence will overflow. However it is much slower since the relationship invloves addition rather than multiplication and so increase much more slowly.

A more immediate problem is with the recurrence relation itself which involves *two* previous terms and the execution speed will get rapidly (*as 2^n*) longer.

A better version is to store all the intermediate values (*upto n*) in an array

```
function fib(n::Integer)
  @assert n > 0
  a = Array{typeof(n),1}(undef,n)
  a[1] = 1
  a[2] = 1
  for i = 3:n
    a[i] = a[i-1] + a[i-2]
  end
  return a[n]
end

# Using the big() function avoids overflow problems
@time(fib(big(101)))
  0.053447 seconds (115.25 k allocations: 2.241 MiB)
573147844013817084101
```

A still better version is to scrap the array itself, which reduces a little on the storage requirements although there is little difference in execute times

```
function fib(n::Integer)
  @assert n > 0
  (a, b) = (big(0), big(1))
  while n > 0
    (a, b) = (b, a+b)
    n -= 1
  end
  return a
end
@time(fib(big(101)))
  0.011516 seconds (31.83 k allocations: 760.443 KiB)
573147844013817084101
```

We can check on the function since fib(n+1)/fib(n) converges to the *Golden* ratio as *n* gets large

```
@printf "%.15f" fib(101)/fib(100)
1.618033988749895

# Golden ratio is equivalent to (1 + sqrt(5))/2
julia> const phi= (1 + sqrt(5))/2;
julia> abs(phi - fib(101)/fib(100))
```

```
5.4321152e-17
```

Simple matrix operations

We will be meeting matrices and matrix operations thorough this book but let us look at the simplest of operations

Taking A and B as defined previously, the normal matrix rules apply.
We'll define C as the transpose of B so:

```
julia> C = transpose(B)
2x3 LinearAlgebra.Transpose{Int64,Array{Int64,2}}:
 1 4 2
 5 3 6
julia> A + C
2x3 Array{Int64,2}:
 2 6 5
 9 8 12
julia> A*B
2x2 Array{Int64,2}:
 15 29
 36 71
```

Matrix division makes more sense with square matrices by it is possible to define the operations for non-square matries too. Note the the / and \ operations produce results of different sizes.

```
julia> A / C
2x2 Array{Float64,2}
 0.332273 0.27663
 0.732909 0.710652

julia> A \ C
3x3 Array{Float64,2}:
 1.27778 -2.44444 0.777778
 0.444444 -0.111111 0.444444
 -0.388889 2.22222 0.111111
```

We will discuss matrix decomposition in more detail later when looking at linear algebra.

Although A * C is not allowed because number of columns of A is not equal to number of rows of C, following are all valid:

```
julia> A .* C
2x3 Array{Int64,2}:
  1   8   6
```

```
 20  15  36

julia> A ./ C
2x3 Array{Float64,2}:
 1.0  0.5      1.5
 0.8  1.66667  1.0

julia> A .== C
2x3 BitArray{2}:
 true   false  false
 false  false   true
```

Simple Markov chain: Cat and Mouse

Suppose there is a row of five adjacent boxes, with a cat in the first box and a mouse in the fifth box. At each 'tick' the cat and the mouse both jump to a random box next to them. On the first tick the cat must jump to box 2 and the mouse to box 4 but on the next ticks they may to the box they started in or to box 3.

When the cat and mouse are in the same box the cat catches the mouse of the chain terminates. Because there are odd number of boxes between the cat and mouse its easy to see that they will not jump past e0ach other.

```
So Markov chain that corresponding to this contains the only five possible
combinations of (Cat,Mouse)
 State 1: (1,3)
 State 2: (1,5)
 State 3: (2,4)
 State 4: (3,5)
 State 5: (2,2), (3,3) & (4,4) # => cat catches the mouse
```

The matrix P = [0 0 .5 0 .5; 0 0 1 0 0; .25 .25 0, .25 .25; 0 0 .5 0 .5; 0 0 0 0 1] represents the probabilities of the transition from one state to the next and the question is how long as the mouse got before its caught. Its best chance is starting in state 2 = (1,5)

The matrix P is a stochastic matrix where all the probabilities along any row add up to 1. This is actually an easy problem to solve using some matrix algebra in a few line in Julia and for a full discussion of the problem look at the Wikipedia discussion

https://en.wikipedia.org/wiki/Stochastic_matrix#Example:_the_cat_and_mouse

```
I = Diagonal(ones(4));
P = [0 0 .5 0; 0 0 1 0; .25 .25 0 .25; 0 0 .5 0];
ep = [0 1 0 0]*inv(I - P)*[1,1,1,1];
println("Expected lifetime for the mouse is $(ep[1]) ticks") # => ep = 4.5
(ticks).
```

- The Diagonal construct returns a square(real) matrix with leading diagonal unity and other values zero.
- The matrix P can be reduced to 4x4 since when in state 5 the Markov chain terminates.
- The inv(I - P)*[1,1,1,1] returns the expected lifetime (no disrespect) of the mouse in all states so multiplying with [0 1 0 0] gives the expectation when starting in state 2.0

Char & Strings

Characters

Julia has a built-in type Char to represent a character. A character occupies 32 bits not 8, so

```
# All the following represent the ASCII character capital-A
julia> c = 'A';
julia> c = Char(65);
julia> c = '\U0041'
'A': ASCII/Unicode U+0041 (category Lu: Letter, uppercase)
```

Since Julia supports unicode c

```
julia> c = '\Uc041'
'□': Unicode U+c041 (category Lo: Letter, other)
```

It is possible to specify a character code of '\Uffff' but char conversion does not check that every value is valid. However Julia provides an **`isvalid()`** function which can be applied to characters

```
julia> c = '\Udff3'; isvalid(c)
false.
```

Julia uses the special C-like syntax for certain ASCII control characters such as '\b', '\t', '\n', '\r', \'f' for backspace, tab, newline, carriage-return and form-feed.

Otherwise the backslash acts as a escape character, so Int('\s') => 115 whereas Int('\t') => 9.

Strings

The type of string we are most familiar with comprises a list of ASCII characters which, in Julia, are normally delimited with double quotes, i.e.

```
julia> s = "Hello there, Blue Eyes"; typeof(s)
String
```

The following points are worth noting

1. The built-in concrete type used for strings (and string literals) is **`String`**
2. This supports the full range of Unicode characters via the UTF-8 encoding
3. A **`transcode()`** function is provided to convert to/from other Unicode encodings.
4. All string types are subtypes of the abstract type **`AbstractString`** so when defining a function expecting a string argument, you should declare the type as **`AbstractString`** in order to accept any string type.

In Julia (as in Java), strings are immutable: i.e. the value of a String object cannot be changed. To construct a different string value, you construct a new string from parts of other strings.

1. ASCII strings are indexable so from s as defined above: s[14:17] # => "Blue".
2. The values in the range are inclusive and if we wish we can change the increment as s[14:2:17] => "Bu" or reverse the slice as s[17:-1:14] => "eulB".
3. Omitting the end of the range is equivalent to running to the end of the string: s[14:] => "Blue Eyes".
4. However s[:14] is somewhat unexpected and gives the character 'B' not the string upto and including the B. This is because the ':' defines a 'symbol', and for a literal :14 is equivalent to 14, so s[:14] is the same as s[14] and not s[1:14]

Strings allow for the special characters such a \n, \t etc.

If we wish to include the double quote we can escape it but Julia provides a """ delimiter.

So s = "This is the double quote \" character" and s = """This is the double quote \" character""" are equivalent.

```
julia> s = "This is a double quote \" character."; println(s);
This is a double quote " character.
```

Strings also provide the *'$' convention* when displaying the value of variable.

```
julia> age = 21; s = "I've been $age for many years now!"
 I've been 21 for many years now!
```

Concatenation of strings can be done using the $-convention but also Julia uses the '*' operator (rather that '+' or some other symbol)

```
julia> s = "Who are you?";
julia> t = " said the Caterpillar."
julia> s*t or "$s$t" # => "Who are you? said the Caterpillar."
```

Regex expressions

Regular expressions came to prominence with their inclusion in Perl programming.
There is an old Perl programmer's adage: *"I had a problem and decided to solve it using regular expressions, now I have two problems".*
Regular expressions are used for pattern matching, numerous books have been written on them and support is available in a variety of our programming languages post-Perl, notably Java and Python. Julia supports regular expressions via a special form of string prefixed with an 'r'.

Suppose we define the pattern empat as:
empat = r"^[_a-z0-9-]+(\.[_a-z0-9-]+)*@[a-z0-9-]+(\.[a-z0-9-]+)*(\.[a-z]{2,4})$"

The follow example will give a clue to what the pattern is associated with.

```
julia> occursin(empat, "fred.flintstone@bedrock.net") ; # => true
julia> occursin(empat, "Fredrick Flintstone@bedrock.net"); # => false
```

The pattern is for a ***valid*** email address and in the second case the space in "Fredrick Flintstone" is not valid (because it contains a space!) so the match fails.

Since we may wish to know not only whether a string matches a certain pattern but also how it is matched, Julia has a function *match()*

```
julia> m = match(r"@bedrock","barney,rubble@bedrock.net")
RegexMatch("@bedrock")
```

If this matches the function returns a RegexMatch object, otherwise it returns 'Nothing'

```
julia> m.match        # => "@bedrock"
julia> m.offset      # => 14
julia> m.captures    # =>
0-element Array{Union{Nothing,SubString{String}},1}
```

Byte Array Literals

Another special form is the byte array literal: `b"..."` which permits string notation express arrays of Uint8 values.
The rules for byte array literals are the following:

1. ASCII characters and ASCII escapes produce a single byte.
2. \x and octal escape sequences produce the byte corresponding to the escape value.
3. Unicode escape sequences produce a sequence of bytes encoding that code point in UTF-8.

```
# Consider the following two examples:
julia> A = b"HEX:\xefcc" # => 7-element Base.CodeUnits{UInt8,String}:
[0x48,0x45,0x58,0x3a,0xef,0x63,0x63]
julia> B = b"\u2200 x \u2203 y" #=> 11-element
Base.CodeUnits{UInt8,String}:
[0xe2,0x88,0x80,0x20,0x78,0x20,0xe2,0x88,0x83,0x20,0x79]

(Note for space I'm showing the output inline, i.e. as the transpose of the
array, the REPL will output in columnwise.
```

Version literals

Version numbers can be expressed with non-standard string literals as `v"..."`.
These literals create VersionNumber objects which follow the specifications of `"semantic versioning"` and therefore are composed of major, minor and patch numeric values, followed by pre-release and build alpha-numeric annotations.

So a full specification typically would be: `v"0.7.1-rc1"` where the major version is "0", minor version "7", patch level "1" and release candidate 1.;

Currently only the major version need to provided and the others assume default values but this make change in the future to allow for more rigorous package management.

So v"1" is equivalent to v"1.0.0"

An Example : Bulls and Cows

Let us look at some code to play the game "Bulls and Cows". A computer program "moo", written in 1970 at MIT in the PL/I, was amongst the first Bulls and Cows computer implementation. It is proven that any number could be solved for up to seven turns and the minimal average game length is 5.21 turns.
The computer enumerates a four digit random number from the digits 1 to 9, without duplication. The player inputs his/her guess and the program should validate the player's guess, reject guesses that are malformed, then print the 'score' in terms of numbes of bulls and cows

- One bull is accumulated for each digit in the guess that equals the corresponding digit in the randomly chosen initial number.
- One cow is accumulated for each digit in the guess that also appears in the randomly chosen number, but in the wrong position.
- The player wins if the guess is the same as the randomly chosen number, and the program ends.
- Otherwise the program accepts a new guess, incrementing the number of 'tries'

```
# Coding this up in Julia
using Random # stdlib module is now needed for srand() => Random.seed!()

tm = round(time());
seed = convert(Int64,tm);
Random.seed!(seed);

# Run this in the REPL, not in the Jupyter notebook

function bacs()
  bulls = cows = turns = 0
  a = Any[]
  while length(unique(a)) < 4
    push!(a,rand('0':'9'))
  end
  my_guess = unique(a)
  println("Bulls and Cows")
  while (bulls != 4)
    print("Guess? > ")
    s = chomp(readline(stdin))
    if (s == "q")
      print("My guess was "); [print(my_guess[i]) for i=1:4]
      return
    end
    guess = collect(s)
    if !(length(unique(guess)) == length(guess) == 4 && all(isdigit,guess))
```

```
        print("\nEnter four distinct digits or q to quit: ")
        continue
      end
      bulls = sum(map(==, guess, my_guess))
      cows = length(intersect(guess,my_guess)) - bulls
      println("$bulls bulls and $cows cows!")
      turns += 1
    end
    println("\nYou guessed my number in $turns turns.")
  end

  # Now run the function
  bacs()
```

Here is a some sample output:

```
BULLS and COWS
===============
Enter four distinct digits or <return> to quit
Guess> 1234
0 bulls and 1 cows!
Guess> 5678
0 bulls and 1 cows!
Guess> 1590
2 bulls and 0 cows!
Guess> 2690
2 bulls and 0 cows!
Guess> 3790
2 bulls and 0 cows!
Guess> 4890
2 bulls and 2 cows!
Guess> 8490
4 bulls and 0 cows!

You guessed my number in 7 turns.
```

We define an array A as Any []; this is because although arrays we described as homogenoeus collections, Julia provides a type 'Any' which can, as the name suggests, stored any form of variable - this is similar to the Microsoft variant datatype.

```
julia> A = Any["There are ",10, " green bottles", " hanging on the
wall.\n"];
julia> [print(A[i]) for i = 1:length(A)]
There are 10 green bottles hanging on the wall.
```

1. Integers are created as characters using the rand() function and pushed onto A with push!()
2. The array A may consist of more than 4 entries so a unique() function is applied

which reduces it to 4 by eliminating duplicates and this is stored in bacs_number.

3. User input is via readline() and this will be a string including the trailing return (\n), so a chomp() function is a applied to remove it and the input is compared with 'q' to allow an escape before the number is guessed.
4. A collect() function applied is applied to return a 4-element array of type Char and it is checked that there are 4 elements and that these are all digits.
5. The number of 'bulls' is determined by comparing each entry in 'guess' and 'bacs_number'; this is achieved by using a map() function to applying '==', 4 bulls and we are done. Otherwise its possible to construct a new array as the intersection of 'guess' and 'bacs_number' which will contain all the elements which match. So subtracting the number of 'bulls' leaves the number of 'cows'

Real, Complex and Rational numbers

Reals

We have met real numbers a few times already; the generic type is FloatingPoint which is sub-classed from Real

A float can be defined as x = 100.0 or x = 1e2 or x = 1f2; all represent the number 100.
The first will be of the type equivalent to the WORD_SIZE, the second of type Float64 and the third (using 'f' rather than 'e' notation) of type Float32
There is also a 'p' notation which can be used with hexadecimals, i.e. x = 0x10p2 corresponds to 64.0

Operators and Built-in Functions

Julia provides comprehensive operator and function support for real numbers. There is a wealth of mathematical functions built-in. In addition to the 'usual' ones such as exp(), log(), sin(), cos() etc., there is support for gamma, bessel, zeta and hankel functions and many others, although the latter set of functions are now in a package in stdlib - SpecialFunctions and this need to be included in the normal way

```
julia> using SpecialFunctions
julia> x = zeta(1.1)
10.584448464950798
```

It is not a bad idea to place the using SpecialFunctions in your julia startup file so that this will be always referenced.

This was previously .juliarc but now has been replaced with .julia/config/startup.jl

One feature to note is that the multiplication operator '*' can be omitted in places where there is no ambiguity. If x is a variable then 2.0x and 2.0*x are both valid. This is useful in cases when dealing with pre-defined constants such as pi, where 2pi => 6.2831

2.4.1.2 Special values

In dealing with real numbers Julia defines three special values Inf, -Inf and Nan. Inf and -Inf refer to values greater (or less) than all finite floating-point values and NaN is "not a number" which is a value not equal to any floating-point value (including itself).

So 1.0/0.0 is Inf and -1.0/0.0 is -Inf, wheras 0.0/0.0 is Nan, as is 0.0 * Inf
Note that typemin(Float64) and typemax(Float64) are defined as -Inf and Inf respectively rather than the minmum/maximum representation

BigFloats

Earlier, in regard to integers, we met BigInts; unsuprisingly there are also BigFloats which can be used for arbitrary precision arithmetic

```
julia> h_atoms_in_universe = 1.0*10.0^82 #=> 1.0e82
julia> x = BigFloat(h_atoms_in_universe) # or big(h_atoms_in_universe)
9.999999999999999634067965630886574211027143225273567793680363843427086501542887e+81
```

Rationals

Julia has a rational number type to represent 'exact' ratios of integers. A rational is defined by use of the // operator, e.g. 5//7. If the numerator and denominator has factor common factor then the number is reduced to its simplest form, 21//35 reduces to 5//7.

```
# Operations on rationals or on mixed rationals and integers return a
rational result:
 julia> x = 3; y = 5//7;
 julia> x*y    # => 15//7;
 julia> y^2    # => 25/49;
```

```
    julia> y/x    # => 5//21;
```

The functions n `numerator()` and `denominator()` return the numerator and denominator of a rational and `float()` can be used to convert a rational to a float.

```
   julia> x = 17//100; numerator(x) # => 17;
   julia> denominator(x) # => 100;
   julia> float(x) => 0.17
```

Complex Numbers

There are two ways to define a complex number in Julia. First using the type definition Complex as its associated constructor Complex().

```
   # Note the difference in these two definitions
   julia> c = Complex(1, 2); typeof(c)
   Complex{Int64}

   julia> c = Complex(1, 2.0); typeof(c)
   Complex{Float64}

   julia> c= ComplexF32(1,2.0); typeof(c)
   Complex{Float32}
```

- Because in the second example, the complex number consists of an ordered pair of two reals, its size is 128 bits whereas the ComplexF32 has 2xFloat32 arguments and ComplexF16 will have 2xFloat16 arguments.
- The number Complex(0.0,1.0) corresponds to the imaginary number 'i', that is sqrt(-1.0), but Julia uses the symbol 'im' rather the 'i' to avoid confusion with a variable i, frequently used as an index, iterator.
- Hence Complex(1, 2) is exactly equivalent to 1 + 2*im, but normally the '*' operator is omitted and this would be expressed as: 1 + 2im.

The complex number supports all the normal arithmetic operations:

```
   julia> c = 1 + 2im;
   julia> d = 3 + 4im;
   julia> c*d
   -5 + 10im
   julia> c/d
   0.44 + 0.08im
   julia> c\d
   2.2 - 0.4im
```

The division c/d and c\d produce real arguments even when the components are integer.

This similar to Julia's behaviour with simple division of integers

Also defined complex functions real(), imag(), conj(), abs(), and angle().
Abs and angle can be used to convert the complex arguments to polar form.

```
julia> c = 1.0 + 2im;
julia> abs(c)2.23606797749979
julia> angle(c)
1.1071487177940904  # (in radians).
```

Complex versions of many mathematical functions can be applied:

```
 julia> c = 1 + 2im;
 julia> sin(c) = 3.1657 + 1.9596im;
 julia> log(c) # => 0.8047 + 1.10715im;
 Julia> sqrt(c) # => 1.272 + 0.78615im
```

Example : Juliasets

Julia documentation provides the example of generating a Mandelbrot set, so we instead we will provide code to create a Julia set instead. This is named after Gaston Julia and is a a generalisation of the Mandlebrot set. Computing a Julia set requires the use of complex numbers.
Both the Mandelbrot set and Julia set (for a given constant z0) are the sets of all z (complex number) for which the iteration z = z*z + z0 does not diverge to infinity. The Mandelbrot set is those z0 for which the Julia set is connected.
We create a file jset.jl and its contents defines the function to generate a Julia set.

```
function juliaset(z, z0, nmax::Int64)
  for n = 1:nmax
    if abs(z) > 2 (return n-1) end
    z = z^2 + z0
  end
  return nmax
end
```

Here z and z0 are complex values and nmax is the number of trials to make before returning. If the modulus of the complex number z gets above 2 then it can be shown that it will increase without limit.

The function returns the number of iterations until the modulus test succeeds or else nmax.

Also we will write a second file pgmfile.jl to handling displaying the Julia set.

```
function create_pgmfile(img, outf::String)
  s = open(outf, "w")
  write(s, "P5\n")
  n, m = size(img)
  write(s, "$m $n 255\n")
  for i=1:n, j=1:m
    p = img[i,j]
    write(s, uint8(p))
  end
  close(s)
end
```

Although we will not be looking in any depth at graphics later in the book, it is quite easy to create a simple disk file using the portable bitmap (netpbm) format. This consists of a "magic" number P1 - P6, followed on the next line the image height, width and a maximum color value which must be greater than 0 and less than 65536; all of these are ASCII values not binary.
Then follows the image values (height x width) which make be ASCII for P1,P2,P3 or binary for P4,P5,P6. There are three different types of portable bitmap; B/W (P1/P4), Grayscale (P2/P5) and Colour (P3/P6)
The function create_pgm() creates a binary grayscale file (magic number = P5) from an image matrix where the values are written as Uint8. Notice that the for loop defines the indices i, j in a single statement with correspondingly only one 'end' statement. The image matrix is output in column order which matches the way it is stored in Julia.

So the main program looks like:

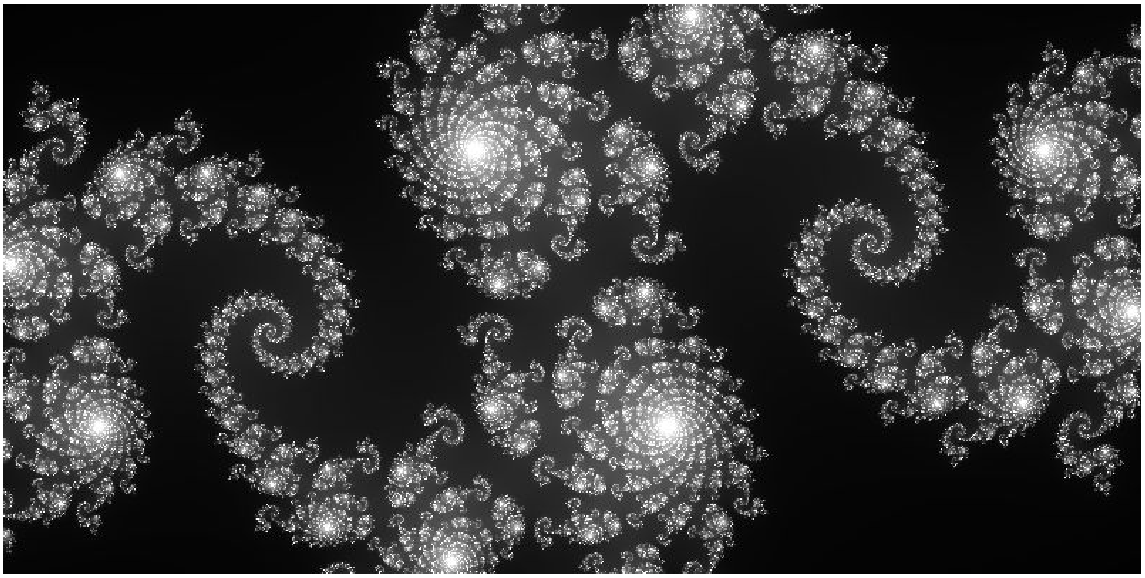
```
include("pgmfile.jl")
include("jset.jl")
pgn_name = "jset.pgm"

function jmain(h::Integer, w::Integer, pgm::String)
   M = Array{Int64,2}(undef,h,w)
   c0 = -0.8 + 0.16im;
   for y=1:h, x=1:w
     c = Complex((x-w/2)/(w/2), (y-h/2)/(w/2))
     M[y,x] = juliaset(c, c0, 256)
   end
   create_pgmfile(M, pgm)
end

eps = @elapsed jmain(400,800,"jset.pgm")
print("Written $pgm_name\nFinished in $eps seconds.\n")
```

1. We define an matrix N of type Int64 to hold the return values from the juliaset

function.

2. The constant c0 is arbitrary, different values of c0 will produce different Julia sets. c0 = 0.0+0.0im corresponds to the Mandelbrotset.
3. The starting complex number is constructed from the (x,y) coordinates and scaled to the half width.
4. We 'cheat' a little by defining the maximum number of iterations as 256.

Because we are writing byte values (UInt8) and value which remains bounded will be 256 and since overflow values wrap around will be output as 0 (black).



Composite Types

A composition type is a collection of named fields, grouped together and treated as a single entity; these are been termed records and structures in some programming languages. (We will be discussing composite types in great detail in the next chapter).
If a type can also have functions (methods) associated with them the resulting collection is termed an object and the languages which support them (Java, C++, Python, Ruby etc.) called object-oriented.
In Julia, functions are not bundled up with the data structures they operate on.

The choice of the method a function uses is termed dispatch.

When the types of ALL of a function's arguments are considered when determining the method employed is termed multiple dispatch and Julia uses this rather than the single dispatch we associated with object methods.

We will be considering the implication of multiple dispatch in detail in the next chapter.

Composite type details are defined with the `struct` keyword, followed by a list of field names, optionally annotated with the :: operator and terminated with `end`. If the type of the field is not specified 'Any' is assumed.
As an example consider a simple type definition for membership of a meetup group:

```
struct Member
  fullname::String
  email::String
  meetup::String
  age::Int
  organiser::Bool
  mobile::String
end
me = Member("Malcolm Sherrington", "malcolm@ljuug.org", "London Julia User
Group", 55, true, "07777 555555")

julia> fieldnames(typeof(me))
(:fullname,:email,:group,:mobile,:organiser,:mobile)

julia> me.fullname
"Malcolm Sherrington"
julia> me.mobile
"07777 555555" # (-- not really, so don't call!
```

Types defined using ***struct*** are immutable, i.e. once created the fields can not be changed

```
julia> me.age = 22
ERROR: type Member is immutable
```

We will see in the next chapter how to create mutable types

A little bit more about Matrices

Lead ins

Vectorised and devectorised code

Consider the following code to add two vectors:

```
function vecadd1(a,b,c,N)
  for i = 1:N
    c = a + b
  end
  return
end

function vecadd2(a,b,c,N)
  for i = 1:N, j = 1:length(c)
    c[j] = a[j] + b[j]
  end
  return
end

julia> A = rand(2); B = rand(2); C = zeros(2);
julia> @elapsed vecadd1(A,B,C,100000000)
6.418755286

julia> @elapsed vecadd2(A,B,C,100000000)
@elapsed vecadd2(A,B,C,100000000)
0.284002398
```

Why the difference in timings? The function vecadd1() uses the array plus operation to perform the calculation whereas vecadd2() explicitly loops through the arrays ad performs a series of scalar additions. The former is an example of vectorized coding and the latter devectorised, the current situation in Julia is that devectorized code is much quicker than vectorised.
With languages such as R, Matlab and Python (using NumPy) vectorized code is faster than devectorized but the reverse is the case in Julia. The reason is that in R (say) vectorization is actually a thin-wrapper around native-C code and since Julia performed is similar to C, calculations which are essentially concerned JUST with array operations will be comparable with those in Julia.
There is little doubt that coding with vector operations is neater and more readable and the designers of Julia are aware of the benefit on improving on timings for vector operations. That it has not been done is tantamount to the difficulty in optimizing code under all circumstances.

Multi-dimensional Arrays

So far we have encounters arrays in one dimension (vectors) and in two (matrices). In fact Julia views all arrays as a single stream of values and applies size and reshape parameters as a means to compute the appropriate indexing.

So arrays with the number of dimensions greater than 2 (i.e. airy > 2) can be defined in a straight-forward method:

```
julia> A = rand(4,4,4)
 4x4x4 Array{Float64,3}:
 [:, :, 1] =
 0.522564 0.852847 0.452363 0.444234
 0.992522 0.450827 0.885484 0.0693068
 0.378972 0.365945 0.757072 0.807745
 0.383636 0.383711 0.304271 0.389717
[:, :, 2] =
 0.570806 0.912306 0.358262 0.494621
 0.810382 0.235757 0.926146 0.915814
 0.634989 0.196174 0.773742 0.158593
 0.700649 0.843975 0.321075 0.306428
[:, :, 3] =
 0.638391 0.606747 0.15706 0.241825
 0.492206 0.798426 0.86354 0.715799
 0.971428 0.200663 0.00568161 0.0868379
 0.936388 0.183021 0.0476718 0.917008
[:, :, 4] =
 0.252962 0.432026 0.817504 0.274034
 0.164883 0.209135 0.925754 0.876917
 0.125772 0.998318 0.593097 0.614772
 0.865795 0.204839 0.315774 0.520044

 Note:
 1. Use of slice ':' to display the 3-D matrix
 2. We can reshape this into a 8x8 2-D matrix.
 3. Values are ordered by the 3rd index, then the second and finally the
first.
```

It is possible to convert this 3-D array into a standard matrix containing the same number of values

```
julia> B = reshape(A,8,8)
 8x8 Array{Float64,2}:
 0.522564 0.452363 0.570806 ... 0.15706 0.252962 0.817504
 0.992522 0.885484 0.810382 ... 0.86354 0.164883 0.925754
 0.378972 0.757072 0.634989 ... 0.005681 0.125772 0.593097
 0.383636 0.304271 0.700649 ... 0.0476718 0.865795 0.315774
```

```
    0.852847 0.444234 0.912306 ... 0.241825 0.432026 0.274034
    0.450827 0.0693068 0.235757 ... 0.715799 0.209135 0.876917
    0.365945 0.807745 0.196174 ... 0.086838 0.998318 0.614772
    0.383711 0.389717 0.843975 ... 0.917008 0.204839 0.520044
```

Or as a simple vector

```
julia> C = reshape(A,64); typeof(C) # => Array{Float64,1}
 julia> transpose(C)
 1x64 LinearAlgebra.Transpose{Float64,Array{Float64,1}}:
 0.522564 0.992522 0.378972 0.383636 ... 0.876917 0.614772 0.520044
```

Sparse Matrices

Normal matrices are sometimes referred to as 'dense', which means that there is an entry for cell[i,j]. In cases where most cell values are 0 (say) this is inefficient ans it is better to implement a scheme of tuples: (i,j,x) where x is the value referenced by i and J.
These are termed sparse matrices and we can create a sparse matrix by:

```
using SparseArrays
S1 = SparseArrays.sparse(I, J, X[, m, n, combine])
S2 = SparseArrays.sparsevec(I, X[, m, combine])
S3 = SparseArrays.sparsevec(D::Dict[, m])
```

where S1 of will dimensions m by n and S[I[k], J[k]] = X[k].

If m and n are given they default to max(I) and max(J). The combine function is used to combine duplicates and if not provide, duplicates are added by default.

S2 is a special case where a sparse vector is created and S3 uses an associative array (dictionary) to provide the same thing. The sparse vector is actually an m by 1 size matrix and in the case of S3 row values are keys from the dictionary and the nonzero values are the values from the dictionary. (see section 2.9.1 for more information on associative array)
Sparse matrices support much of the same set of operations as dense matrices but there are a few special functions which can be applied. For example spzeros(), spones, speye() are the counterparts of zeros(), ones() and eye() and random number arrays can be generated by sprand() and sprandn().

```
# The 0.1 means only ~10% for the numbers generated will be deemed as
nonzero
# This will produce different arrays each time it is run
julia>  A = sprand(5,5,0.1)
5×5 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  0.611724
  [4, 1]  =  0.325444
  [4, 2]  =  0.722912
# So sqauring the matrix produces another sparse matrix
julia> A * A
5×5 SparseMatrixCSC{Float64,Int64} with 2 stored entries:
  [1, 1]  =  0.374207
  [4, 1]  =  0.199082
```

Using Matrix() converts the sparse matrix to a dense one as:

```
julia> B = full(A);
julia> typeof(B)
5x5 Array{Float64,2}
 0.611724  0.0       0.0  0.0  0.0
0.0        0.0       0.0  0.0  0.0
0.0        0.0       0.0  0.0  0.0
 0.325444  0.722912  0.0  0.0  0.0
0.0        0.0       0.0  0.0  0.0
```

Data Arrays and Data Frames

Users of R will be aware of the success of data frames when employed in analysing dataset, a success which has been mirrored by Python with the "pandas" package. Julia too adds data frame support thorugh use a package DataFrames, which is available on Github, in the usual way.

The package extends Julia's base Julia by introducing three basic types:

1. Missing.missing: An indicator that a data value is missing
2. DataArray: An extension to the Array type that can contain missing values
3. DataFrame: A data structure for representing tabular data sets

It is such a large topic that we will be looking at data frames in some depth when we consider statistical computing in chapter 6.
However to get a flavour of processing data with these packages:

```
julia> using DataFrames
julia> df1 = DataFrame(ID = 1:4,
                      Cost = [10.1,7.9,missing,4.5])
4×2 DataFrame
│ Row │ ID │ Cost    │
```

```
├────────┼───────┼──────────────┤
│ 1    │ 1   │ 10.1     │
│ 2    │ 2   │ 7.9      │
│ 3    │ 3   │ missing  │
│ 4    │ 4   │ 4.5      │
```

Common operations such as computing mean(d) or var(d)of the Cost because of the missing value in row 3

```
julia> using Statistics
julia> mean(df1[:Cost])
missing
```

We can create a new data frame by dropping ALL the rows with missing values and now statistical functions can be applied as normal:

```
julia> df2 = dropmissing(df1)
3×2 DataFrames.DataFrame
│ Row │ ID │ Cost │
├────────┼───────┼──────────┤
│ 1    │ 1   │ 10.1 │
│ 2    │ 2   │ 7.9  │
│ 3    │ 4   │ 4.5  │
julia> (μ,σ) = (mean(df2[:Cost]),std(df2[:Cost]))
(7.5, 2.8213471959331766)
```

Dictionaries, Sets and Others

In addition to arrays, Julia supports associative arrays, sets and many other data structures In this section we will introduce a few..

Dictionaries

Associative arrays consist of collections of (key,values) pairs. In Julia associative array are called dictionaries (Dicts).
Let us look at a simple data type to hold a user credentials: ID, password, email etc. We will not include a username as this will be the key to a credential data type. In practice this would not be a great idea that users often forget their username as well as their password!

To implement this we use a simple module (We will be looking at modules in more detail in chapter 4). This includes a type (struct) and some functions which operate on that type. Note the inclusion of the 'export' statement which makes the type UserCreds and the functions visible.

```
module Auth
using Base64
struct UserCreds
   uid::Int
   password::String
   fullname::String
   email::String
   admin::Bool
end
function matchPwds(_mc::Dict{String,UserCreds},
                   _name::String,
                   _pwd::String)
   return (_mc[_name].password == base64(_pwd) ? true : false)
end
isAdmin(_mc::Dict{String,UserCreds},
        _name::String) = _mc[_name].admin;
export UserCreds, matchPwds, isAdmin;
end
```

We can use this to create an empty authentication array (AA) and add an entry for myself. We will be discussing security and encryption later, so at present we'll just use the base64() function to scramble the password.

```
julia> using Auth
julia> using Base64
julia> AA = Dict{String,UserCreds}();
julia> AA["malcolm"] = UserCreds(101,base64encode("Pa55word"),"Malcolm
Sherrington","malcolm@myemail.org",true);

julia> println(matchPwds(AA, "malcolm", "Pa55word") ? "OK" : "No, sorry")
 OK
```

Adding the user requires the scrambling of the password by the user, otherwise matchPwds will fail.
To overcome this we can override the default constructor UserCreds() by adding an internal constructor inside the type definition - this is an exception to the rule that type definitions can?t contain functions, since clearly it does not conflict with the requirement for multiple dispatch.
The *"using Auth"* statement looks for auth.jl in directories on the LOAD_PATH but will also include the current directory. On a Linux system where v"0.7" is installed on /opt typically would be:

```
julia> println(LOAD_PATH)
[@"@, "@v#.#", "@stdlib"]
```

We can add to the LOAD_PATH with push!:

If we add this statement to the startup file it will happen whenever Julia starts up.

push!(LOAD_PATH, "/home/malcolm/jlmodules) ;

An alternatively way to define the dictionary is adding some initial values

```
julia> BB = ["malcolm" => UserCreds(101,base64("Pa55word"),
             "Malcolm Sherrington","malcolm@myemail.org",true)];
```

So the values can be reference via the key.

```
julia> me = BB["malcolm"]
UserCreds(101,"UGE1NXdvcmQ=",
          "Malcolm Sherrington","malcolm@myemail.org",true)
```

The '.' notation is used to reference the fields

julia> me.fullname
"Malcolm Sherrington"

Alternatively is is possible to iterate over all the keys

```
for who in keys(BB)
  println( AA[:who].fullname)
end
"Malcolm Sherrington"
```

Attempting to retrieve a value with a key does not exist, such as AA["james"], will produce and error.

We need to trap this in the module routines such as matchPwds and isAdmin using try/catch/finally syntax.

```
# i.e. isAdmin function in auth.jk could be rewritten as:
function isAdmin2(_mc::Dict{String,UserCreds},
                  _name::ASCIIString)
  check_admin::Bool = false;
  try
    check_admin = _mc[_name].admin
  catch
    check_admin = false
  finally
```

```
      return check_admin
    end
  end
```

Sets

A set is a collection of distinct objects and the "Bulls and Cows" example earlier could have been implemented using sets rather than strings. Julia implements its support for sets in Base.Set (file: set.jl) and the underlying datastructure is an associative array.

The basic constructor creates a set with elements of type Any, supplying arguments will determine (restrict) the set type

```
julia> S0 = Set()
Set(Any[])
```

Alternative we can create a set of specific type of elements

```
julia> S1 = Set([1,2,3])
Set([2, 3, 1])
julia> typeof(S1)
Set{Int64}
julia> S2 = Set([2,4,6])
Set([4, 2, 6])
```

The 'usual' functions of union and intersection can be applied to s1 and S2

```
julia> S3 = union(S1,S2)
Set([4, 2, 3, 6, 1])
julia> S4 = intersect(S1,S2)
Set([2])
```

Also we can check whether one set is a subset of a second

```
julia> issubset(S3,S4)
false

julia> issubset(S4,S3)
true
```

Elements can be added to a set using the push!() function.
Recall the '!' implies that the data structure is altered.

```
# This works
julia> push!(S0,"Malcolm")
Set(Any["Malcolm"])
```

```
# But this does NOT
push!(S1,"Malcolm")
ERROR: MethodError: Cannot `convert` an object of type String to an object
of type Int64
```

Other Data Structures

The package DataStructures implements a ***rich*** bag of data structures including deques, queues, stacks, heaps, ordered sets, linked lists, digital trees etc.

For a full discussion of ALL of these see the following URL:

`https://github.com/JuliaCollections/DataStructures.jl`

As an illustration lets conclude this chapter by look at the Deque type.
This is a double-ended queue with allows insertion and removal of elements at both ends of a sequence. The Stack and Queue types are based on the a Deque type and provide interfaces for FILO and FIFO access respectively. Deques expose push!(), pop!() shift!() and unshift!() functions.

Consider the following simple example to illustrate using stacks and queues:

```
using DataStructures

julia> S = Stack{Char}(100);  typeof(S)
Stack{Char}

julia> Q = Queue{Char}(100); typeof(Q)
Queue{Char}
```

A stack will use push!() and pop!() to add and retrieve data, a queue will use push!() and unshift!(); queues also encapsulate the latter two processes as enqueue!() and dequeue!()

Stacks are FILOs (last in.first out) while queues are FIFOs (first in, first out) as the following demonstrates:

```
julia> greet = "Here's looking at you kid!";
julia> for i = 1:lastindex(greet)
 push!(S,greet[i])
 enqueue!(Q,greet[i])
 end

 julia> for i = 1:lastindex(greet) print(pop!(S)) end
```

```
    !dik uoy ta gnikool s'ereH

    julia> for i = 1:lastindex(greet) print(dequeue!(Q)) end
    Here's looking at you kid!
```

Summary

In this chapter we started on a more in-depth look at Julia with a more detailed discussion of various scalar, vector and matrix data types comprising integer, real numbers, characters and strings as well as the operations acting on them.

We then moved on to data types such as rational numbers, big integers and floats and complex numbers.

Finally we looked at some complex data structures such as data array and data frames, dictionaries and sets, stacks and queues.

The next chapter follows on by considering the type system in greater detail: defining composite data structures and the use of parametrisation.

Index