A. Introduction to Julia

References: The Julia Documentation, The Julia–Matlab–Python Cheatsheet, Think Julia

These notes give an overview of Julia. In these notes we focus on the aspects of Julia and computing that are essential to numerical computing:

1. Integers: We discuss briefly how to create and manipulate integers, and how to see the underlying bit representation.
2. Strings and parsing: We discuss how to create and manipulate strings and characters, and how we can convert a string

of 0's and 1's to an integer or other type. 3. Vectors and matrices: We discuss how to build and manipulate vectors and matrices (which are both types of *arrays*). Later lectures will discuss linear algebra. 4. Types: In Julia everything has a type, which plays a similar role to classes in Python. Here we discuss how to make new types, for example, a complex number in radial format. 5. Loops and branches: We discuss `if`, `for` and `while`, which work similar to Python. 6. Functions: We discuss the construction of named and anonymous functions. Julia allows overloading functions for different types, for example, we can overload `*` for our radial complex type. 7. Modules, Packages, and Plotting: We discuss how to load external packages, in particular, for plotting.

1. Integers

Julia uses a math-like syntax for manipulating integers:

In [1]:
```julia
1 + 1 # Addition
```
Out[1]:
```
2
```

In [2]:
```julia
2 * 3 # Multiplication
```
Out[2]:
```
6
```

In [3]:
```julia
2 / 3 # Division
```
Out[3]:
```
0.6666666666666666
```

In [4]:
```julia
x = 5; # semicolon is optional but supresses output if used in the last line
x^2 # Powers
```
Out[4]:
```
25
```

In Julia everything has a type. This is similar in spirit to a class in Python, but much more lightweight. An integer defaults to a type `Int`, which is either 32-bit (`Int32`) or 64-

bit (`Int64`) depending on the processor of the machine. There are also 8-bit (`Int8`), 16-bit (`Int16`), and 128-bit (`Int128`) integer types, which we can construct by converting an `Int`, e.g. `Int8(3)`.

These are all "primitive types", instances of the type are stored in memory as a fixed length sequence of bits. We can find the type of a variable as follows:

In [5]:
```
typeof(x)
```

Out[5]:
```
Int64
```

For a primitive type we can see the bits using the function `bitstring`:

In [6]:
```
bitstring(Int8(1))
```

Out[6]:
```
"00000001"
```

Negative numbers may be surprising:

In [7]:
```
bitstring(-Int8(1))
```

Out[7]:
```
"11111111"
```

This is explained in detail in Chapter Numbers.

There are other primitive integer types: `UInt8`, `UInt16`, `UInt32`, and `UInt64` are unsigned integers, e.g., we do not interpret the number as negative if the first bit is `1`. As they tend to be used to represent bit sequences they are displayed in hexadecimal, that is base-16, using digits 0-9a-c, e.g., $12 = (c)_{16}$:

In [8]:
```
UInt16(12)
```

Out[8]:
```
0x000c
```

A non-primitive type is `BigInt` which allows arbitrary length integers (using an arbitrary amount of memory):

In [9]:
```
factorial(big(100))^10
```

Out[9]:
```
501229411781623769749660913457440438241423232012200322468893211086126013704
894539443344309163974791659034852365995250806062011686578246599786735232198
821804597039806782364137104356302622509999458512471445459730019563426771428
142169285986787068214579295521341498939492118275642554858413261194143086490
268931209191751598129642166571081867269800866623596546422054167388335563209
506282983082667791283399575989899227202499266538194820953638625853344859790
670876861351789336077269493142900753157564688168114708204886421032550156933
567286643950190106682511408121306047648604161917328707343138398784542423497
604981394493598166286048081979508907005988879132386487437206698804316246051
907147850704889031422801542480308658883031446705268143162836428270481946917
616560029942842623910155313531513785228134356266366120116543326234048654779
501332706634928129188477486388847112098299317511538535401423273161204152693
099733507852570701555778059731473275144712539224416914644770413696741547032
650605385759271986528123972264597981517180893649589642619662794059376815997
114926751678156467705271005657886203103034886745869986816763168755424650630
543268271783299008211100347741675970542806368972994314611089910916303943706
258492662397699718485205856883926544847011250572559721527493331235100895744
451398327234919981678691884155485959440954156071175484497209589760000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000
00000
```

2. Strings and parsing

We have seen that `bitstring` returns a string of bits. Strings can be created with quotation marks

In [10]:
```
str = "hello world 😬"
```

Out[10]:
```
"hello world 😬"
```

We can access characters of a string with brackets:

In [11]:
```
str[1], str[13]
```

Out[11]:
```
('h', '😬')
```

Each character is a primitive type, in this case using 32 bits/4 bytes:

In [12]:
```
typeof(str[6]), length(bitstring(str[6]))
```

Out[12]:
```
(Char, 32)
```

Strings are not primitive types, but rather point to the start of a sequence of `Char` s in memory. In this case, there are $32 * 13 = 416$ bits/52 bytes in memory.

Strings are *immutable*: once created they cannot be changed. But a new string can be created that modifies an existing string. The simplest example is `*`, which concatenates two strings:

```julia
In [13]: "hi" * "bye"
```

```
Out[13]: "hibye"
```

(Why `*`? Because concatenation is non-commutive.) We can combine this with indexing to, for example, create a new string with a different last character:

```julia
In [14]: str[1:end-1] * "😂"
```

```
Out[14]: "hello world 😂"
```

Parsing strings

We can use the command `parse` to turn a string into an integer:

```julia
In [15]: parse(Int, "123")
```

```
Out[15]: 123
```

We can specify base 2 as an optional argument:

```julia
In [16]: parse(Int, "-101"; base=2)
```

```
Out[16]: -5
```

If we are specifying bits its safer to parse as an `UInt32`, otherwise the first bit is not recognised as a sign:

```julia
In [17]: bts = "11110000100111111001100110001010"
         x = parse(UInt32, bts; base=2)
```

```
Out[17]: 0xf09f998a
```

The function `reinterpret` allows us to reinterpret the resulting sequence of 32 bits as a different type. For example, we can reinterpret as an `Int32` in which case the first bit is taken to be the sign bit and we get a negative number:

```julia
In [18]: reinterpret(Int32, x)
```

```
Out[18]: -257975926
```

We can also reinterpret as a `Char`:

```julia
In [19]: reinterpret(Char, x)
```

```
Out[19]: '🙊': Unicode U+1F64A (category So: Symbol, other)
```

We will use `parse` and `reinterpret` as it allows one to easily manipulate bits. This is not actually how one should do it as it is slow.

Bitwise operations (non-examinable)

In practice, one should manipulate bits using bitwise operations. These will not be required in this course and are not examinable, but are valuable to know if you have a career involving high performance computing. The `p << k` shifts the bits of `p` to the left `k` times inserting zeros, while `p >> k` shifts to the right:

In [20]:
```
println(bitstring(23));
println(bitstring(23 << 2));
println(bitstring(23 >> 2));
```

```
0000000000000000000000000000000000000000000000000000000000010111
0000000000000000000000000000000000000000000000000000000001011100
0000000000000000000000000000000000000000000000000000000000000101
```

The operations `&`, `|` and `⊻` do bitwise and, or, and xor.

3. Vectors, Matrices, and Arrays

We can create a vector using brackets:

In [21]:
```
v = [11, 24, 32]
```

Out[21]:
```
3-element Vector{Int64}:
 11
 24
 32
```

Like a string, elements are accessed via brackets. Julia uses 1-based indexing (like Matlab and Mathematica, unlike Python and C which use 0-based indexing):

In [22]:
```
v[1], v[3]
```

Out[22]:
```
(11, 32)
```

Accessing outside the range gives an error:

In [23]:
```
v[4]
```

```
BoundsError: attempt to access 3-element Vector{Int64} at index [4]

Stacktrace:
 [1] getindex(A::Vector{Int64}, i1::Int64)
   @ Base ./array.jl:924
 [2] top-level scope
   @ In[23]:1
```

Vectors can be made with different types, for example, here is a vector of three 8-bit integers:

In [24]:
```
v = [Int8(11), Int8(24), Int8(32)]
```

Out[24]:
```
3-element Vector{Int8}:
 11
 24
 32
```

Just like strings, Vectors are not primitive types, but rather point to the start of sequence of bits in memory that are interpreted in the corresponding type. In this last case, there are $3 * 8 = 24$ bits/3 bytes in memory.

The easiest way to create a vector is to use `zeros` to create a zero `Vector` and then modify its entries:

In [25]:
```
v = zeros(Int, 5)
v[2] = 3
v
```

Out[25]:
```
5-element Vector{Int64}:
 0
 3
 0
 0
 0
```

Note: we can't assign a non-integer floating point number to an integer vector:

In [26]:
```
v[2] = 3.5
```

```
InexactError: Int64(3.5)

Stacktrace:
 [1] Int64
   @ ./float.jl:788 [inlined]
 [2] convert
   @ ./number.jl:7 [inlined]
 [3] setindex!(A::Vector{Int64}, x::Float64, i1::Int64)
   @ Base ./array.jl:966
 [4] top-level scope
   @ In[26]:1
```

We can also create vectors with `ones` (a vector of all ones), `rand` (a vector of random numbers between `0` and `1`) and `randn` (a vector of samples of normal distributed quasi-random numbers).

When we create a vector whose entries are of different types, they are mapped to a type that can represent every entry. For example, here we input a list of one `Int32` followed by three `Int64`s, which are automatically converted to all be `Int64`:

In [27]:
```
[Int32(1), 2, 3, 4]
```

Out[27]:
```
4-element Vector{Int64}:
 1
 2
 3
 4
```

In the event that the types cannot automatically be converted, it defaults to an `Any` vector, which is similar to a Python list. This is bad performancewise as it does not know how many bits each element will need, so should be avoided.

In [28]:
```julia
[1.0, 1, "1"]
```

Out[28]:
```
3-element Vector{Any}:
 1.0
 1
  "1"
```

We can also specify the type of the Vector explicitly by writing the desired type before the first bracket:

In [29]:
```julia
Int32[1, 2, 3]
```

Out[29]:
```
3-element Vector{Int32}:
 1
 2
 3
```

We can also create an array using comprehensions:

In [30]:
```julia
[k^2 for k = 1:5]
```

Out[30]:
```
5-element Vector{Int64}:
  1
  4
  9
 16
 25
```

Matrices are created similar to vectors, but by specifying two dimensions instead of one. Again, the simplest way is to use `zeros` to create a matrix of all zeros:

In [31]:
```julia
zeros(Int, 4, 5) # creates a 4 × 5 matrix of Int zeros
```

Out[31]:
```
4×5 Matrix{Int64}:
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  0  0
```

We can also create matrices by hand. Here, spaces delimit the columns and semicolons delimit the rows:

In [32]:
```julia
A = [1 2; 3 4; 5 6]
```

Out[32]:
```
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6
```

We can also create matrices using brackets, a formula, and a `for` command:

In [33]:
```
[k^2+j for k=1:4, j=1:5]
```

Out[33]:
```
4×5 Matrix{Int64}:
  2   3   4   5   6
  5   6   7   8   9
 10  11  12  13  14
 17  18  19  20  21
```

Matrices are really vectors in disguise. They are still stored in memory in a consecutive sequence of bits. We can see the underlying vector using the `vec` command:

In [34]:
```
vec(A)
```

Out[34]:
```
6-element Vector{Int64}:
 1
 3
 5
 2
 4
 6
```

The only difference between matrices and vectors from the computers perspective is that they have a `size` which changes the interpretation of whats stored in memory:

In [35]:
```
size(A)
```

Out[35]:
```
(3, 2)
```

Matrices can be manipulated easily on a computer. We can multiply a matrix times vector:

In [36]:
```
x = [8; 9]
A * x
```

Out[36]:
```
3-element Vector{Int64}:
 26
 60
 94
```

or a matrix times matrix:

In [37]:
```
A * [4 5; 6 7]
```

Out[37]:
```
3×2 Matrix{Int64}:
 16  19
 36  43
 56  67
```

If you use `.*`, it does entrywise multiplication:

In [38]:
```
[1 2; 3 4] .* [4 5; 6 7]
```

Out[38]:
```
2×2 Matrix{Int64}:
  4  10
 18  28
```

We can take the transpose of a real vector as follows:

In [39]:
```
a = [1, 2, 3]
a'
```

Out[39]:
```
1×3 adjoint(::Vector{Int64}) with eltype Int64:
 1  2  3
```

Note for complex-valued vectors this is the conjugate-transpose, and so one may need to use `transpose(a)`. Both `a'` and `transpose(a)` should be thought of as "dual-vectors", and so multiplcation with a transposed vector with a normal vector gives a constant:

In [40]:
```
b = [4, 5, 6]
a' * b
```

Out[40]:
```
32
```

One important note: a vector is not the same as an `n x 1` matrix, and a transposed vector is not the same as a `1 × n` matrix.

Accessing and altering subsections of arrays

We will use the following notation to get at the columns and rows of matrices:

```
A[a:b,k]    # returns the a-th through b-th rows of the k-th column of A as a Vector of length (b-a+1)
A[k,a:b]    # returns the a-th through b-th columns of the k-th row of A as a Vector of length (b-a+1)
A[:,k]      # returns all rows of the k-th column of A as a Vector of length size(A,1)
A[k,:]      # returns all columns of the k-th row of A as a Vector of length size(A,2)
A[a:b,c:d]  # returns the a-th through b-th rows and c-th through d-th columns of A
            # as a (b-a+1) x (d-c+1) Matrix
```

The ranges `a:b` and `c:d` can be replaced by any `AbstractVector{Int}`. For example:

In [41]:
```
A = [1 2 3; 4 5 6; 7 8 9; 10 11 12]
A[[1,3,4],2]  # returns the 1st, 3rd and 4th rows of the 2nd column of A
```

Out[41]:
```
3-element Vector{Int64}:
  2
  8
 11
```

Exercise Can you guess what `A[2,[1,3,4]]` returns, using the definition of `A` as above?
What about `A[1:2,[1,3]]`? And `A[1,B[1:2,1]]`? And `vec(A[1,B[1:2,1]])`?

We can also use this notation to modify entries of the matrix. For example, we can set the `1:2` x `2:3` subblock of A to `[1 2; 3 4]` as follows:

In [42]:
```
A[1:2,2:3] = [1 2; 3 4]
A
```

Out[42]:
```
4×3 Matrix{Int64}:
  1   1   2
  4   3   4
  7   8   9
 10  11  12
```

Broadcasting

It often is necessary to apply a function to every entry of a vector. By adding `.` to the end of a function we "broadcast" the function over a vector:

In [43]:
```
x = [1,2,3]
cos.(x) # equivalent to [cos(1), cos(2), cos(3)]
```

Out[43]:
```
3-element Vector{Float64}:
  0.5403023058681398
 -0.4161468365471424
 -0.9899924966004454
```

Broadcasting has some interesting behaviour for matrices. If one dimension of a matrix (or vector) is 1, it automatically repeats the matrix (or vector) to match the size of another example.

Example

In [44]:
```
[1,2,3] .* [4,5]'
```

Out[44]:
```
3×2 Matrix{Int64}:
  4   5
  8  10
 12  15
```

Since `size([1,2,3],2) == 1` it repeats the same vector to match the size `size([4,5]',2) == 2`. Similarly, `[4,5]'` is repeated 3 times. So the above is equivalent to:

In [45]:
```
[1 1; 2 2; 3 3] .* [4 5; 4 5; 4 5]
```

Out[45]:
```
3×2 Matrix{Int64}:
  4   5
  8  10
 12  15
```

Note we can also use broadcasting with our own functions (construction discussed later):

In [46]:
```
f = (x,y) -> cos(x + 2y)
f.([1,2,3], [4,5]')
```

Out[46]:
```
3×2 Matrix{Float64}:
 -0.91113    0.0044257
 -0.839072   0.843854
  0.0044257  0.907447
```

Ranges

We have already seen that we can represent a range of integers via `a:b`. Note we can convert it to a `Vector` as follows:

In [47]:
```
Vector(2:6)
```

Out[47]:
```
5-element Vector{Int64}:
 2
 3
 4
 5
 6
```

We can also specify a step:

In [48]:
```
Vector(2:2:6), Vector(6:-1:2)
```

Out[48]:
```
([2, 4, 6], [6, 5, 4, 3, 2])
```

Finally, the `range` function gives more functionality, for example, we can create 4 evenly spaced points between `-1` and `1`:

In [49]:
```
Vector(range(-1, 1; length=4))
```

Out[49]:
```
4-element Vector{Float64}:
 -1.0
 -0.3333333333333333
  0.3333333333333333
  1.0
```

Note that `Vector` is mutable but a range is not:

In [50]:
```julia
r = 2:6
r[2] = 3   # Not allowed
```

```
CanonicalIndexError: setindex! not defined for UnitRange{Int64}

Stacktrace:
 [1] error_if_canonical_setindex(#unused#::IndexLinear, A::UnitRange{Int6
4}, #unused#::Int64)
   @ Base ./abstractarray.jl:1352
 [2] setindex!(A::UnitRange{Int64}, v::Int64, I::Int64)
   @ Base ./abstractarray.jl:1343
 [3] top-level scope
   @ In[50]:2
```

4. Types

Julia has two different kinds of types: primitive types (like `Int64`, `Int32`, `UInt32` and `Char`) and composite types. Here is an example of an in-built composite type representing complex numbers, for example, $z = 1 + 2\mathrm{i}$:

In [51]:
```julia
z = 1 + 2im
typeof(z)
```

Out[51]:
```
Complex{Int64}
```

A complex number consists of two fields: a real part (denoted `re`) and an imaginary part (denoted `im`). Fields of a type can be accessed using the `.` notation:

In [52]:
```julia
z.re, z.im
```

Out[52]:
```
(1, 2)
```

We can make our own types. Let's make a type to represent complex numbers in the format

$$z = r\exp(\mathrm{i}\theta)$$

That is, we want to create a type with two fields: `r` and `θ`. This is done using the `struct` syntax, followed by a list of names for the fields, and finally the keyword `end`.

In [53]:
```julia
struct RadialComplex
    r
    θ
```

```
end
z = RadialComplex(1,0.1)
```

Out[53]: RadialComplex(1, 0.1)

We can access the fields using `.`:

In [54]:
```
z.r, z.θ
```

Out[54]: (1, 0.1)

Note that the fields are immutable: we can create a new `RadialComplex` but we cannot modify an existing one. To make a mutable type we use the command `mutable struct`:

In [55]:
```
mutable struct MutableRadialComplex
    r
    θ
end

z = MutableRadialComplex(1,2)
z.r = 2
z.θ = 3
z
```

Out[55]: MutableRadialComplex(2, 3)

Abstract types

Every type is a sub-type of an *abstract type*, which can never be instantiated on its own. For example, every integer and floating point number is a real number. Therefore, there is an abstract type `Real`, which encapsulates many other types, including `Float64`, `Float32`, `Int64` and `Int32`.

We can test if type `T` is part of an abstract type `V` using the sytax `T <: V`:

In [56]:
```
Float64 <: Real, Float32 <: Real, Int64 <: Real
```

Out[56]: (true, true, true)

Every type has one and only one super type, which is *always* an abstract type. The function `supertype` applied to a type returns its super type:

In [57]:
```
supertype(Int32)  # returns Signed, which represents all signed integers.
```

Out[57]: Signed

In [58]:
```
supertype(Float32) # returns `AbstractFloat`, which is a subtype of `Real`
```

Out[58]: AbstractFloat

An abstract type also has a super type:

In [59]:
```
supertype(Real)
```

Out[59]:
```
Number
```

Type annotation and templating

The types `RadialComplex` and `MutableRadialComplex` won't be efficient as we have not told the compiler the type of `r` and `θ`. For the purposes of this module, this is fine as we are not focussing on high performance computing. However, it may be of interest how to rectify this.

We can impose a type on the field name with `::`:

In [60]:
```
struct FastRadialComplex
    r::Float64
    θ::Float64
end
z = FastRadialComplex(1,0.1)
z.r, z.θ
```

Out[60]:
```
(1.0, 0.1)
```

In this case `z` is stored using precisely 128-bits.

Sometimes we want to support multiple types. For example, we may wish to support 32-bit floats. This can be done as follows:

In [61]:
```
struct TemplatedRadialComplex{T}
    r::T
    θ::T
end
z = TemplatedRadialComplex(1f0,0.1f0) # f0 creates a `Float32`
```

Out[61]:
```
TemplatedRadialComplex{Float32}(1.0f0, 0.1f0)
```

This is stored in precisely 64-bits.

Relationship with C structs, heap and stack (advanced)

For those familiar with C, a `struct` in Julia whose fields are primitive types or composite types built from primitive types, is exactly equivalent to a `struct` C, and can in fact be passed to C functions without any performance cost. Behind the scenes Julia uses the LLVM compiler and so C and Julia can be freely mixed.

Another thing to note is that from a programmers perspective there are three types of memory: registers, the stack and the heap. Registers only live on the CPU and are extremely fast. The stack lives in memory and has fixed memory length and is much faster than the heap as it avoids dynamic allocation and deallocation of memory. So an instance of a type with a known fixed length (like `FastRadialComplex`) will typically live either in registers or in the stack (the compiler sorts out the details) and be much faster than an instance of a type with unknown or variable length (like `RadialComplex` or `Vector`), which must be on the heap.

5. Loops and branches

Loops such as `for` work essentially the same as in Python. The one caveat is to remember we are using 1-based indexing, e.g., `1:5` is a range consisting of `[1,2,3,4,5]`:

In [62]:
```
for k = 1:5
    println(k^2)
end
```

```
1
4
9
16
25
```

There are also `while` loops:

In [63]:
```
x = 1
while x < 5
    println("x is $x which is less than 5, incrementing!")
    x += 1
end
x
```

```
x is 1 which is less than 5, incrementing!
x is 2 which is less than 5, incrementing!
x is 3 which is less than 5, incrementing!
x is 4 which is less than 5, incrementing!
```

Out[63]:
```
5
```

If-elseif-else statements look like:

In [64]:
```
x = 5
if isodd(x)
    println("it's odd")
elseif x == 2
    println("it's 2")
else
    println("it's even")
end
```

```
it's odd
```

6. Functions

Functions are created in a number of ways. The most standard way is using the keyword `function`, followed by a name for the function, and in parentheses a list of arguments. Let's make a function that takes in a single number x and returns x^2.

In [65]:
```
function sq(x)
   x^2
end
sq(2), sq(3)
```

Out[65]:
```
(4, 9)
```

There is also a convenient syntax for defining functions on one line, e.g., we can also write

In [66]:
```
sq(x) = x^2
```

Out[66]:
```
sq (generic function with 1 method)
```

Multiple arguments to the function can be included with `,`.
Here's a function that takes in 3 arguments and returns the average.
(We write it on 3 lines only to show that functions can take multiple lines.)

In [67]:
```
function av(x, y, z)
    ret = x + y
    ret = ret + z
    ret/3
end
av(1, 2, 3)
```

Out[67]:
```
2.0
```

Variables live in different scopes. In the previous example, `x`, `y`, `z` and `ret` are *local variables*: they only exist inside of `av`.
So this means `x` and `z` are *not* the same as our complex number `x` and `z` defined above.

Warning: if you reference variables not defined inside the function, they will use the outer scope definition.
The following example shows that if we mistype the first argument as `xx`, then it takes on the outer scope definition `x`, which is a complex number:

In [68]:
```
function av2(xx, y, z)
    (x + y + z)/3
end
```

Out[68]:
```
av2 (generic function with 1 method)
```

You should almost never use this feature!!
We should ideally be able to predict the output of a function from knowing just the

inputs.

Example Let's create a function that calculates the average of the entries of a vector.

In [69]:
```
function vecaverage(v)
    ret=0
    for k = 1:length(v)
        ret = ret + v[k]
    end
    ret/length(v)
end
vecaverage([1,5,2,3,8,2])
```

Out[69]: 3.5

Julia has an inbuilt `sum` command that we can use to check our code:

In [70]:
```
sum([1,5,2,3,8,2])/6
```

Out[70]: 3.5

Functions with type signatures

functions can be defined only for specific types using `::` after the variable name. The same function name can be used with different type signatures.

The following defines a function `mydot` that calculates the dot product, with a definition changing depending on whether it is an `Integer` or a `Vector`. Note that `Integer` is an abstract type that includes all integer types: `mydot` is defined for pairs of `Int64`'s, `Int32`'s, etc.

In [71]:
```
function mydot(a::Integer, b::Integer)
    a*b
end

function mydot(a::AbstractVector, b::AbstractVector)
    # we assume length(a)  == length(b)
    ret = 0
    for k = 1:length(a)
        ret = ret + a[k]*b[k]
    end
    ret
end

mydot(5, 6)  # calls the first definition
```

Out[71]: 30

In [72]:
```
mydot(Int8(5), Int8(6))   # also calls the first definition
```

Out[72]: 30

In [73]:
```
mydot(1:3, [4,5,6])     # calls the second definition
```

Out[73]:
```
32
```

We should actually check that the lengths of `a` and `b` match.
Let's rewrite `mydot` using an `if`, `else` statement. The following code only does the for loop if the length of a is equal to the length of b, otherwise, it throws an error.

If we name something with the exact same signature (name, and argument types), previous definitions get overriden. Here we correct the implementation of `mydot` to throw an error if the lengths of the inputs do not match:

In [74]:
```
function mydot(a::AbstractVector, b::AbstractVector)
    ret=0
    if length(a) == length(b)
        for k = 1:length(a)
            ret = ret + a[k]*b[k]
        end
    else
        error("arguments have different lengths")
    end
    ret
end
mydot([1,2,3], [5,6,7,8])
```

```
arguments have different lengths

Stacktrace:
 [1] error(s::String)
   @ Base ./error.jl:35
 [2] mydot(a::Vector{Int64}, b::Vector{Int64})
   @ Main ./In[74]:8
 [3] top-level scope
   @ In[74]:12
```

Anonymous (lambda) functions

Just like Python it is possible to make anonymous functions, with two variants on syntax:

In [75]:
```
f = x -> x^2
g = function(x)
    x^2
end
```

Out[75]:
```
#9 (generic function with 1 method)
```

There is not much difference between named and anonymous functions, both are compiled in the same manner. The only difference is named functions are in a sense "permanent". One can essentially think of named functions as "global constant anonymous functions".

Tuples

`Tuple` s are similar to vectors but written with the notation `(x,y,z)` instead of `[x,y,z]` . They allow the storage of *different types*. For example:

In [76]:
```
t = (1,2.0,"hi")
```

Out[76]:
```
(1, 2.0, "hi")
```

On the surface, this is very similar to a `Vector{Any}` :

In [77]:
```
v=[1,2.0,"hi"]
```

Out[77]:
```
3-element Vector{Any}:
 1
 2.0
  "hi"
```

The main difference is that a `Tuple` knows the type of its arguments:

In [78]:
```
typeof(t)
```

Out[78]:
```
Tuple{Int64, Float64, String}
```

The main benefit of tuples for us is that they provide a convenient way to return multiple arguments from a function. For example, the following returns both `cos(x)` and `x^2` from a single function:

In [79]:
```
function mytuplereturn(x)
    (cos(x), x^2)
end
mytuplereturn(5)
```

Out[79]:
```
(0.28366218546322625, 25)
```

We can also employ the convenient syntax to create two variables at once:

In [80]:
```
x,y = mytuplereturn(5)
```

Out[80]:
```
(0.28366218546322625, 25)
```

Modules, Packages, and Plotting

Julia, like Python, has modules and packages. For example to load support for linear algebra functionality like `norm` and `det` , we need to load the `LinearAlgebra` module:

In [81]:
```
using LinearAlgebra
norm([1,2,3]), det([1 2; 3 4])
```

Out[81]: (3.7416573867739413, -2.0)

It is fairly straightword to create ones own modules and packages, however, we will not need modules in this....module.

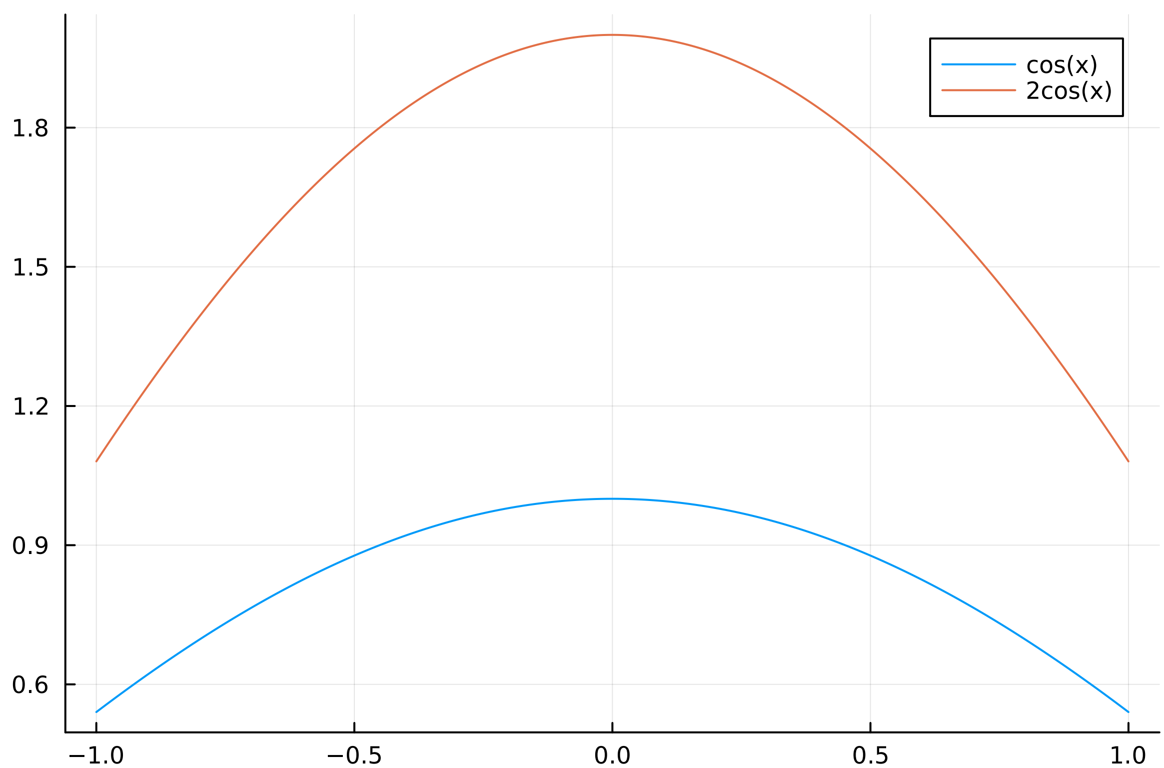
Plotting

Some important functionality such as plotting requires non-built in packages. There are many packages such as PyPlot.jl, which wraps Python's matplotlib and Makie.jl, which is a state-of-the-art GPU based 3D plotting package. We will use Plots.jl, which is an umbrella package that supports different backends.

For example, we can plot a simple function as follows:

In [82]:
```
using Plots
x = range(-1, 1; length=1000) # Create a range of a 1000 evenly spaced numbe
y = cos.(x) # Create a new vector with `cos` applied to each entry of `x`
plot(x, y; label="cos(x)")
plot!(x, 2y; label="2cos(x)")
```

Out[82]:



Note the `!` is just a convention: any function that modifies its input or global state should have `!` at the end of its name.

Installing packages (advanced)

If you choose to use Julia on your own machine, you may need to install packages. This can be done by typing the following, either in Jupyter or in the REPL: `] add Plots`.