Types

Julia has two different kinds of types: bittypes (like `Int64`, `Int32`, `UInt32` and `Char`) and composite types.

Here is an example of an inbuilt composite type representing complex numbers, for example,

$$x = 1 + i$$

In [110]:

```
x=1+2im
```

Out[110]:

```
1 + 2im
```

In [111]:

```
typeof(x)
```

Out[111]:

```
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 [112]:

```
x.re
```

Out[112]:

```
1
```

In [113]:

```
x.im
```

Out[113]:

```
2
```

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

$$z = r exp(i\theta)$$

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

In [101]:

```
type MyComplex
    r
    θ
end
```

In [102]:

```
z=MyComplex(1,0.1)
```

Out[102]:

```
MyComplex(1,0.1)
```

We can access fields for our new type using `.r` and `.θ`:

In [103]:

```
z.r,z.θ
```

Out[103]:

```
(1,0.1)
```

Functions

Functions are created 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 [104]:

```
function sq(x)
   x^2
end
```

Out[104]:

```
sq (generic function with 1 method)
```

In [105]:

```
sq(2),sq(3)
```

Out[105]:

```
(4,9)
```

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 [108]:

```
function av(x,y,z)
    ret=x+y
    ret=ret+z
    ret/3
end
```

Out[108]:

```
av (generic function with 1 method)
```

In [109]:

```
av(1,2,3)
```

Out[109]:

```
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 [114]:

```
function av2(xx,y,z)
    (x+y+z)/3
end
```

Out[114]:

```
av2 (generic function with 1 method)
```

In [115]:

```
av2(5,6,7)
```

Out[115]:

```
4.666666666666667 + 0.6666666666666666im
```

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

Functions of Vectors

We can define functions for other types, for example vectors. Let's create a function that calculates the average of the entries of a vector. We want the function to work for general length vectors, so let's create vectors `v` and `w` of different lengths:

In [116]:

```
v=rand(Int,5)
w=rand(Int,10)

length(v),length(w)
```

Out[116]:

```
(5,10)
```

To implement the function, we need to use a for loop, using the `for` keyword. The following syntax evaluates the body of the for loop (between the lines after the `for` and before the `end`) one by one for `k` equal to every number in the range `1:10`.

In [118]:

```
for k=1:5
    k2=k^2
    println(k2)
end
```

```
1
4
9
16
25
```

This is exactly the same as the following block of text, but without having to write it out explicitely

In [119]:

```
k=1
k2=k^2
println(k2)


k=2
k2=k^2
println(k2)

k=3
k2=k^2
println(k2)

k=4
k2=k^2
println(k2)

k=5
k2=k^2
println(k2)
```

```
1
4
9
16
```

```
25
```

We can use a for loop to step `k` through every index of a vector. The following calculates the sum of the entries of the vector, printing out the current value for each value of k

In [120]:

```
v=[1,5,6,3]
ret=0
for k=1:length(v)
    ret=ret+v[k]
    println("At step $k, the current sum is $ret")
end
ret
```

```
At step 1, the current sum is 1
```

Out[120]:

```
15
```

```
At step 2, the current sum is 6
At step 3, the current sum is 12
At step 4, the current sum is 15
```

We are now ready to write a function that calculates the average of the entries of a vector:

In [39]:

```
function vecav(v)
    ret=0

    for k=1:length(v)
        ret=ret+v[k]
    end
    ret/length(v)
end
```

Out[39]:

```
vecav (generic function with 1 method)
```

In [122]:

```
vecav([1,5,2,3,8,2])
```

Out[122]:

```
3.5
```

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

In [123]:

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

Out[123]:

```
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` means any kind of integer: `mydot` is defined for pairs of Int64's, Int32's, etc.

In [124]:

```
function mydot(a::Integer,b::Integer)
    a*b
end

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

Out[124]:

```
mydot (generic function with 2 methods)
```

In [125]:

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

Out[125]:

```
30
```

In [126]:

```
mydot(Int8(5),Int8(6))   # also calls the first definition
```

Out[126]:

```
30
```

In [127]:

```
mydot([1,2,3],[4,5,6])    # calls the second definition
```

Out[127]:

```
32
```

In [128]:

```
mydot([1,2,3,4],[4,5,6])     # an error is thrown because length(a) > length
(b)
```

```
LoadError: BoundsError: attempt to access 3-element Array{Int64,1
}:
 4
 5
 6
  at index [4]
while loading In[128], in expression starting on line 1

 in mydot at In[124]:9
```

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.

Note that == checks if two quantities are equal. This is *not the same* as =, which assigns the value of one quantity to the other

If we name something with the exact same signature (name, and argument types), previous definitions get overriden.

In [129]:

```
function mydot(a::Vector,b::Vector)
    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
```

Out[129]:

```
mydot (generic function with 2 methods)
```

In [130]:

```
mydot([1,2,3,4],[5,6,7,8])
```

Out[130]:

```
70
```

In [131]:

```
mydot([1,2,3,4],[5,6,7])
```

```
LoadError: arguments have different lengths
while loading In[131], in expression starting on line 1

 in mydot at In[129]:8
```

The fields in types point to locations in memory

Let's return to the example we started last lecture:

In [57]:

```
r=Ref(1)
r.x
```

Out[57]:

```
1
```

`Ref` is just a composite type with a single field called `x`. We can make our own version of `Ref` called `MyRef`:

In [132]:

```
type MyRef
    x
end
```

The function call `MyRef(52)` creates a new `MyRef`, with `x` initialized as 52:

In [133]:

```
myref=MyRef(52)
```

Out[133]:

```
MyRef(52)
```

In [134]:

```
myref.x
```

Out[134]:

```
52
```

we can create another variable `n` that is equal to `myref`.

In [135]:

```
n=myref
```

Out[135]:

```
MyRef(52)
```

Unlike bittypes, the fields of composite types point to locations in memory that store the values. In this example, `myref.x` lives somewhere in memory, let's say at address 1543. But setting `n=myref` has the property that all the fields of `n` also point to the same location in memory. This means `n.x` also points to address 1543.

So if we change the value of `myref.x` to 6, this changes the value living in address 1543 to 6, and so `n.x` is also automatically 6:

In [136]:

```
myref.x=6
```

Out[136]:

```
6
```

In [137]:

```
n.x
```

Out[137]:

```
6
```

This is very different from bittypes. Here, `myrefx` and `nx` are in two different locations in memory, let's say 1765 and 1987, and the = copies the value 52 from `myrefx`'s address 1765 to `nx`'s address 1987. Then calling `myrefx=6` actually creates a new address in memory, let's say 2076, with the value of 6. But `nx` still corresponds to 1987, and is still 52.

In [138]:

```
myrefx=52
nx=myrefx
myrefx=6
nx
```

Out[138]:

```
52
```

Here is another example. Let's return to the composite type set-up:

In [139]:

```
myref=MyRef(52)
myref.x
n=myref
```

Out[139]:

```
MyRef(52)
```

If instead of calling `myref.x=6` we call `myref=MyRef(6)`, this creates a brand new `MyRef`, with the new `myref.x` pointing to a new address in memory (let's say 6543) initialized with the value 6. Whereas `n.x` still points to the same address in memory as the old `myref.x`, so is still 52:

In [67]:

```
myref=MyRef(6)
```

Out[67]:

```
MyRef(6)
```

In [68]:

```
n.x
```

Out[68]:

```
52
```

Vectors work like composite types, not bittypes

Vectors behave like composite types, where they point to an address in memory, and = copies the address in memory:

In [141]:

```
v=[1,2,3,4]

w=v
```

Out[141]:

```
4-element Array{Int64,1}:
 1
 2
 3
 4
```

You can change values of a vector using brackets and =:

In [142]:

```
v[2]=52
```

Out[142]:

```
52
```

This has changed v:

In [72]:

```
v
```

Out[72]:

```
4-element Array{Int64,1}:
  1
 52
  3
  4
```

But it's also changed w, since w points to the same location in memory as v:

In [143]:

```
w
```

Out[143]:

```
4-element Array{Int64,1}:
  1
 52
  3
  4
```

If we assign v to a new vector, w still points to the old location in memory, so is unchanged.

Makes sure it is clear the difference between `v=`, which reassigns the variable v to a new value, and `v[1]=`, which leaves v the same, but modifies a value in memory.

In [74]:

```
v=[6,75]
```

Out[74]:

```
2-element Array{Int64,1}:
  6
 75
```

In [75]:

```
w
```

Out[75]:

```
4-element Array{Int64,1}:
  1
 52
  3
  4
```

If you actually want to copy the entries of a vector, without pointing to a new vector, use `copy`:

In [147]:

```
v=[6,75]
w=v
w2=copy(v)
```

Out[147]:

```
2-element Array{Int64,1}:
  6
 75
```

In [148]:

```
v[1]=2
```

Out[148]:

```
2
```

In [150]:

```
w
```

Out[150]:

```
2-element Array{Int64,1}:
  2
 75
```

In [149]:

```
w2
```

Out[149]:

```
2-element Array{Int64,1}:
  6
 75
```

On to floating point numbers

Floating point numbers represent real numbers. They are also a bitstype, by default `Float64` which uses 64 bits, but not we interpret the bits in a different way than integers.

In [153]:

```
x=1/2
typeof(x)
```

Out[153]:

```
Float64
```

In [154]:

```
y=1/3
typeof(y)


bits(y)
```

Out[154]:

```
"0011111111010101010101010101010101010101010101010101010101010101"
```

We can create floats by adding .0 to the end. The following creates a Float64 to represent the integer 1:

In [155]:

```
x=1.0

bits(x)
```

Out[155]:

```
"0011111111110000000000000000000000000000000000000000000000000000"
```