Types (and variables)

Motivating question

What is `1/2` ?

Seems obvious?

What if I told you that `1.0/2.0` is different to `1/2`? It is in C.

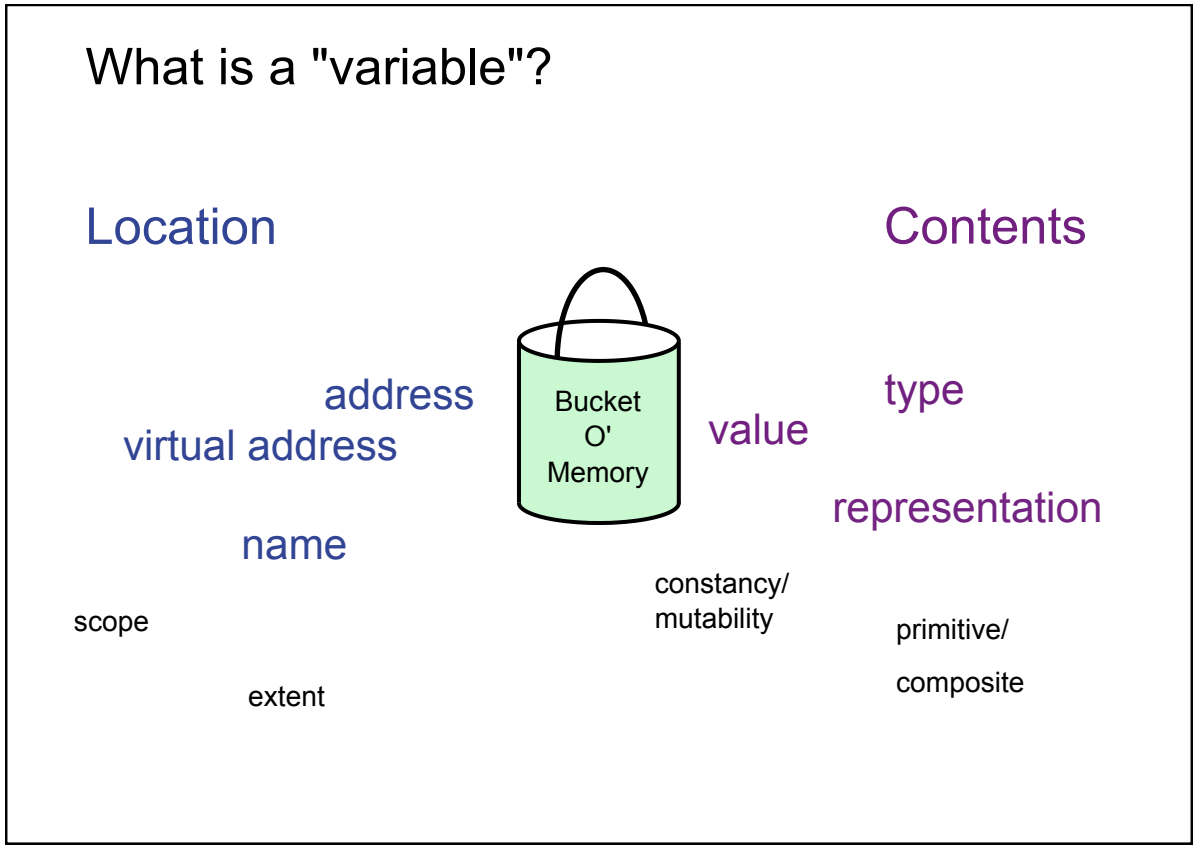
What if I noted that we were operating `mod 3`? Then 1/2 is the number that, when multiplied by 2 results in 1. That number would be 2, so `1/2 mod 3 = 2`.

Is it so obvious now? The secret is that the type of value we are working with influences the nature of the operations on the value. So, somehow, we should keep track of types. So in this short doc I will try to explain types.

Variables and types

To understand types, we need to understand variables. As is often the case, the name "variable" is dreadful. In mathematics we are familiar with variables, *e.g.*, x, which are abstract containers that only take on values as solutions. In programming, most variables are likewise containers for values, but the values are rarely abstract. So despite the name, we need to redefine what we are working with.

But in programming a *variable* is not actually precisely defined. Different programming languages take a grab at different sets of concepts when constructing their notion of *variable*.



A variable has three key concepts:

- a bucket of memory (usually allocated in fixed size blocks, *e.g.*, 32 or 64 bits),
- which is at a location, and
- which (at some point in time) contains a value.

The bucket-of-memory is allocated via multiple layers of the system: through the operating system and programming tool, and in most (modern) cases you have very little control over where it is. So to keep track of it you need some other information. You could try to record its location as an address (really a virtual location) but that isn't easy (pointers, which store locations are actually variables themselves). So we usually reference a variable by a *name*. Modern languages make that so easy, it seems fundamental, but it isn't.

> *Note that a variable can have more than one name, i.e., more than one way to get to the same value.*

The useful part of a variable is its ability to store a value, but once again there isn't too much that is completely standard about this part. In order for a value to be stored on a computer it has to be represented using binary digits, i.e., 1s and 0s. The *type* of a variable tells the computer what the representation will be. The representation/type determines what "type" of value can be stored. Hence the name.

Likewise, although most variables are intended to store values (but not all – can you think of an example?), the time at which the value is allocated to the variable may change, and some variables might (at least mistakenly) never be allocated a value. One of the main divisions between compiled languages such as C and FORTRAN and just-in-time (JIT) compiled languages such as Python, R and Julia is that in compiled languages variables are created in two steps: (i) they are defined (and given a type), and (ii) a value is assigned to the variable (possibly to be changed later). In JIT languages the two steps are often combined (they can be combined in some compiled languages, but in JIT languages the process is more streamlined.)

Another of the divisions between compiled languages and just-in-time (JIT) compiled languages is that a common facility provided in JIT languages is dynamic types. In an old-school language like C, the compiler has to determine the type of all variables before they can be used. So the type of a variable is designated when it is defined. There are ways to get around this, but they often cause more bugs than they fix. In modern JIT languages the type of a variable can be determined as the program runs. That creates a great deal of flexibility, at some cost.

Some languages (for instance Julia) allow both in an optimised way. Sometimes this is called *gradual typing*.

Types are attached to values but we often think of them as part of the variable because in old-school "statically-typed languages" a variable could not change its type. In modern, dynamically-typed languages like Julia a variables type can change so we think of the type as being part of the value.

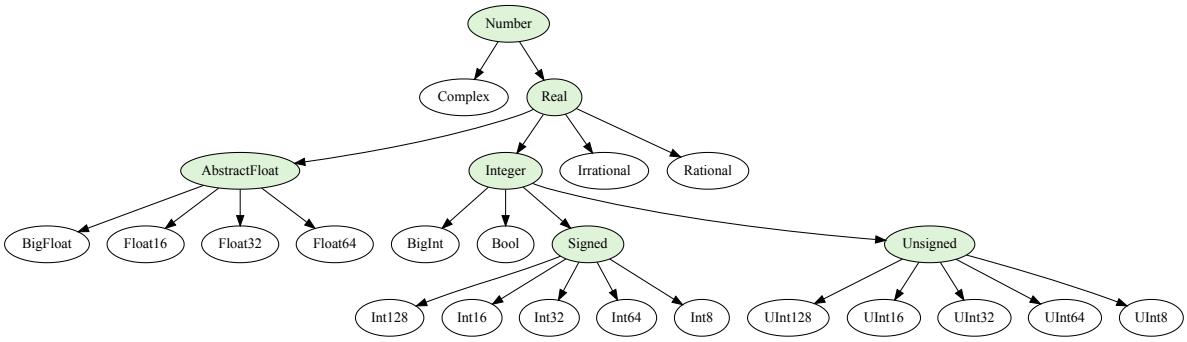
Note also that every computed value (even if it is not associated with a named variable) must have a type.

Common types

The most common types are:

- Numbers
- Characters (singular or in "strings")

But there are many others, and there are even many sub-types of numbers and characters. The following figure illustrates Julia's number types.



The green nodes are *abstract* types. They aren't used to create variables, but rather as containers that group together a set of related types. The other types are *concrete* and we can create variables with those types. The number, as in the 32 in `Int32`, says how big the container is. In that case 32 bits.

The tree of numbers illustrates many of the most common types used in programming languages:

- **Integers:** represent whole numbers like -23, 0 and 1235322311. There are a small number of fairly standard ways that computers use to map a decimal number (such as the ones above) into binary. Look up 1's complement and 2's-complement and Gray codes. 2's complement is probably most common.

  https://en.wikipedia.org/wiki/Integer_(computer_science)

  In many languages the size of an "int" is left to the compiler/system. In Julia it is explicit, which is much more reliable IMHO.

- **Floating point:** is the most common way to represent an **approximation** of a real number. You convert the number into a form like 123×10^{231} and then store two integers (123 and 231). Floating point is a *very* complicated topic, but these days many modern languages use a fairly standard representation called IEEE 754. Its supported by hardware in most modern computers addresses some of the problems and so has come to dominate.

There are a few others number types in this picture but I will leave them for the moment.

The other most commonly used types involve characters and groups of characters. Julia has two main ways to represent these:

- **Char**: represents a single (Unicode) character. Unicode is a modern but fairly complex way to represent characters in binary that allows a MUCH larger set of characters (all the languages of the world and then some) than the old ASCII standard, which was entirely anglocentric.
- **String**: represents a "string" of characters, e.g., "abc". In some languages a string is just an array of character and indeed the element type of a string in Julia is Char. However a string in Julia is actually a byte array, and some Unicode characters may take more than one byte.

We also use the **Bool** type a lot. Bools can take the values `true` and `false` and follow the rules of Boolean algebra. Even if you never declare a Boolean, you will be using them, for instance in the conditional statements you use in if-then-else statements.

A variable can be built up from other variables, perhaps the most common example being an array. For instance, we might want to store the vector $x = (4, 2, 1)$. Most modern languages provide a shortcut to allow you do do this as a block and access the individual elements by *indexing* into the array. For example, given x above we have $x[1] = 4$. The elements in this array are all integers, but in principle an array can store many different types, but with some loss of efficiency. Underlying it, the array is just a block of memory with some fancy syntax to access its elements.

Number types in more detail

There are a large number of possible number types (see the figure). You can get the one you want in a number of ways, perhaps the most common being to use special shorthand, e.g.:

```
julia> 1       # Int64 (this is the default on a 64-bit computer)
julia> local x::UInt8 = 1 # assign 1 to a UInt8
julia> local x::Int32 = 1 # only local variables can use '::' syntax
julia> local x::Int32 = 1 # assign 1.0 to a Float16
julia> 1 // 3  # a rational number (stored exactly!)
julia>
julia> 0b10110 # binary literal (but will be converted to an integer)
julia> 0xff    # hexadecimal literal (but will be converted to an int)
julia> 0x1     # UInt8
julia> 0x10000 # UInt32, because this is too big for UInt8
julia> 1.0     # Float64 (this is the default on a 64-bit computer)
julia> 1 + 1im # complex (Int)
julia> 1.0 + 1.0im # complex (Float)
 Irrational{:π}
julia> setprecision(BigFloat, 1024) do
          BigFloat(π)
       end
julia> import Base: @irrational
julia> @irrational twoπ   6.2831853071795864769 big(π) * 2.0
```

https://julialang.org/blog/2017/03/piday/

Hint: use `Sys.WORD_SIZE` to find out your operating systems default integer size.

Converting between types

I just mentioned converting between types. In programming jargon this is called *casting* or *type coercion*. In compiled languages, (statically-typed languages in general) where the type is attached to the name/variable (not the value) this process requires assigning the output to a new variable (that's not quite true: in some languages like C you can work directly on a block of memory, and treat it like whatever type you like through casting). In dynamically-typed languages its a lot easier.

In Julia I can convert using the function `convert`, for instance

```
julia> convert(Int64, 'a')
97
```

Thus I have found the integer value of the character 'a'.

However, it is complicated by the fact that in either one can *cast* a value implicitly by performing certain operations. In many languages 'a' + 0 will cast the result to the integer value of the character 'a'. Its used as a common trick to convert ASCII characters into their numerical values. But casting rules are often subtle and complicated and a source of many tears before breakfast.

Julia's designers chose to minimise implicit casting. The result is some tricks don't work

```
julia> 'a' + 0
'a': ASCII/Unicode U+0061 (category Ll: Letter, lowercase)
```

Overall that is good. Errors are more likely to be explicitly noted (and hence easily fixed) rather than the code apparently working but doing something stupid. I would rather lose a few tricks and spend less time debugging.

However, some casting is needed. When we add any forms of integers we want to get an output (not an error) so some casting is needed. For instance

```
julia> 1 + 0x1
2
```

That is, when we add 1 (a 64-bit integer by default) and 0x1 (an unsigned integer) we get 2. In Julia this is called *promotion*. A less comprehensive type is promoted to the more comprehensive type when they are combined together by an operator. Julia has means to create your own promotion rules, but I wouldn't recommend it until you have some experience.

One more note, Julia functions don't need to specify the output type, but by convention, if the function takes one type as input, you probably should return the same type if it makes sense to do so. That is, functions should not accidentally do type conversion.

Why should I care

There are a few languages these days that do their darndest to hide types from you. They choose default types, convert between them freely, and keep track of the mess. This seems to make your life easier. In the short run that is true, but when you are writing serious code it can catch up with you. Let me tell you why.

Once upon a time memory was *expensive*. My first computer had 1,850 bytes of RAM and a large part of that was for storing the program. It had very little space for variables. So little that even the names were limited to two letters. You could not waste memory on machines like this, so if you only needed the numbers 0-255 then you only needed an 8-bit integer, not the full 32 bits or even 64 that a modern programming language might give you by default.

"So what?" you say. "My computer has 16 GB of RAM!" Well, we live in the days of big data. Using this data effectively requires careful memory management, not just in order to store the data (in memory or on disk) but also because reading and writing it can be faster if you are more efficient. Languages that "help" you with types are part of the reason "big" data is big.

In addition, converting between types is potentially quite slow. It can create a bottleneck in your code that just isn't necessary.

However the key reason to use types is safety. Using types well prevents many hard to debug errors. Take my word for it!

The main reason is that bugs appear at compile time (and are explicitly described in a good language like Julia), rather than being hidden bugs at run time. Finding bugs where the code "works" but produces the wrong results is MUCH harder.

All those factors together provide a rationale for being able to control types.

Multiple dispatch

Another concept strongly linked to types is the idea of a set of methods (or functions or procedures) that are related to that type. Object-oriented languages go so far as to incorporate these methods into the type.

Julia uses a different approach. It allows you to specify the types of input arguments to a function, and the function can only be called if they match.

Multiple dispatch is useful because we can overload the same function name with different meanings depending on what it is operating on. A good example is the commonly defined `show` function.

One thing you might not know is that multiple dispatch can't make a decision based on value. That is, the type of the input arguments is used, not the value. However, you can create types that mimic values if you need to do that. The `Distributions.jl` package uses that extensively: *e.g.:* it creates a hierarchy of types of distributions (Univariate, Multivariate and Discrete and Continuous) and subtypes for each distribution.

```
julia> using Distributions
julia> c = Chernoff()
julia> typeof( c )
Chernoff

julia> mean(c)
0.0
```

Here the type `Chernoff` of the variable is telling us which function `mean` to call. Other types in `Distributions` also contain (parameter) values, but using a type without a value as a signal is neat trick.

We could say much, much more about multiple dispatch, but will leave that for another day.

Creating your own types: objects/structure/classes

Julia allows you to create your own types in several ways illustrated below. There is much more to say about this, but let's go with a few examples for the moment.

```
julia> abstract type MyReal <: Number end
julia> primitive type MyInt8 <: Signed 8 end
julia> struct Foo # composite type
           bar
           baz::Int
           qux::Float64
       end
julia> mutable struct Bar # mutable composite type
           baz
           qux::Float64
       end
julia> IntOrString = Union{Int,AbstractString}
julia> struct Point{T} # parametric type
           x::T
           y::T
       end
julia> Point{Float64}
julia> struct NoFields # singletone type
       end
julia> struct WrapType{T} # wrapper type
       value::T
       end
julia> if Int === Int64 # type aliases
          const UInt = UInt64
       else
          const UInt = UInt32
       end
```

```
julia>
```

And if you want to create a fully-fledged type you will want to create all the methods that go with it, e.g., any operators like +, *, but also you will want to extend `Base.show()` and `Base.hash()`.

Functions and operators for working with types

Julia has a suite of functions and operators just for working with types:

```
T <: S         # test whether T is a subtype of S
subtypes(T)    # list the subtypes of type T
supertype(T)   # list the parent type of type T
typeof(X)      # give the type of a variable X
eltype(A)      # the type of the elements of A
sizeof(X)      # the size (in bits) of variable X
sizeof(T)      # the size (in bits) of type T
typemax(T)     # the maximum possible value of type T
typemin(T)     # the minimum possible value of type T
typeassert(X,T) # throw an error if X is not type T
varinfo()      # list variables and types
isa(X, T)      # test if X is of type T
```

Type hints: you can get Julia to test that the type you want is the type you get. Try:

```
1::Int64
1::Int32
```

Also try `varinfo(Base)`.

Parametric types

The complex number type is interesting because it is *parametric*. That is, it is the complex type uses another type as a parameter. For instance, you can have complex numbers formed from integers, or from floats, e.g. try

```
julia> typeof(1 + 1im)
julia> typeof(1.0 + 1im)
```

Many types are parametric, *e.g.*, Arrays, Dictionaries, …

You can define your own parametric types, but I will leave learning that trick to you.

Mutability

By default Julia `structs` are immutable. They cannot be changed after being created. The contained objects may be mutable (for instance arrays), so you might change the values inside a an object internal to the `struct` but not the `struct` itself. If the compiler knows the struct can't change form, it can create more efficient machine code. For instance, you only need one such variable if it has the same values.

```
julia> struct X
           a::Int
           b::Float64
       end
julia> x1 = X(1,2)
julia> x2 = X(1,2)
julia> x1 === x2
true
julia> x1.a = 2
ERROR: setfield! immutable struct of type X cannot be changed
```

You can also create mutable `structs` when you need to be able to change the structure.

Other variables can also be made immutable, but the default for most is mutability.

Symbols

We typically think of variables as data, *i.e.*, the numbers and other values that our program will work on.

However code is also stored in memory. In a very real sense we can think of a function or any other chunk of code as a bucket-of-memory with a location containing a value with some representation, *i.e.*, a variable.

Julia is *homoiconic* which means, loosely, that the language's code can be represented as a data structure in the language itself. This means Julia program can write Julia programs. From there it is turtles all the way down.

https://www.expressionsofchange.org/homoiconicity-revisited/

Homoiconicity is confusing (at best). But one aspect of it is that Julia has a type designed to hold code. Symbols are *interned* strings. An interned string is immutable, so you only need one copy. Thus comparison are fast and memory usage is reduced.

https://en.wikipedia.org/wiki/String_interning

Symbols are created using a single colon, *e.g.,*

```
julia> s = :foo
julia> typeof(s)
Symbol
```

Or using an explicit constructor (which you can use if the symbol has a space or other punctuation in it)

```
julia> s = Symbol("foo bar")
julia> typeof(s)
Symbol
```

Symbols can be used directly as a convenient syntactic sugar as they are in Data Frames in Julia. However their real purpose is to allow *metaprogramming*. Symbols can be use to indicate access to a variable when evaluating an expression. This goes way beyond this tutorial though.

Some more types you should know

- Any
- Missing
- Enum
- DataFrames
- Union (try `IntOrString = Union{Int,AbstractString}`)
- Tuple
- DataType (try `typeof(Int)`)
- Arbitrary precision integers (BigInt)
- Dates
- Sparse arrays
- Distributions
- Random number generators

The mathematics of types

There is a deep theory of types.

The maths looks like the most obscure pure maths.

Its very cool but beyond my pay grade.

Other variable concepts

- **Primitive::** types are just a block of bits.
- **Composite:** types (records or structs or objects) are build up of sub-blocks, and often each sub-block has its own name.
- **Literals:** are values hard-wired into the code, *e.g.*, in the statement `x=11` the value 11 is a literal.

- **Scope:** of a name is the part of the program that can see that name. *Global* variable (names) are visibly everywhere.
- **Extent:** is the time (in the program) at which a variable exists. Usually it is linked to scope, but a *static* variable can exist in between calls to the function in which it is defined and so as extent greater than its scope. Think of extent and scope as time and space.

Links

In no particular order:

- https://en.wikibooks.org/wiki/Introducing_Julia/Types
- https://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node43.html
- https://www.geeksforgeeks.org/types-in-julia-set-2/
- https://en.wikipedia.org/wiki/Type_system
- https://julia-ylwu.readthedocs.io/en/latest/manual/integers-and-floating-point-numbers.html
- https://scls.gitbooks.io/ljthw/content/_chapters/06-ex3.html
- https://docs.julialang.org/en/v1/devdocs/types/
- https://en.wikibooks.org/wiki/Introducing_Julia/Types
- https://julia.quantecon.org/getting_started_julia/introduction_to_types.html
- https://docs.julialang.org/en/v1/manual/metaprogramming/